Functional Components

Using COM components in Haskell

Daan Leijen, master thesis

August 10, 1998

Contents

Acknowledgements

Who else to thank first than the one who showed me the beauty of the lambda. Almost two years ago, Jon Mountjoy guided my first steps on the functional path. After a few months, Simon Peyton Jones accepted me as a student and I went to Portland for half a year. The energy and commitment of Simon probably makes him one of the best supervisors a student can have. Erik Meijer became my semi-supervisor and not only helped me to make this project succeed but also taught me the Zen of functional programming.

Of course more is needed than two excellent supervisors; Lillian and Dino are responsible for the american workout two times a week, my climb and dive buddies Mark and Roger took me to the most beautiful spots of Oregon and my friends Julie, Tanya and Laszlo made sure that I was never unhappy. Everyone at the OGI, thank you for the wonderful stay!

A special word of thanks to my supervisor in Amsterdam, Marcel Beemster. His way of working has been very inspiring and I learned a lot from him. Last, but not least, I want to thank my girlfriend Toos for always being there for me.

Daan Leijen, March 1998.

Chapter 1

Introduction

”The physical realization of a functional component is not, in some sense, its essence. Rather, what makes a functional component the type it is, is characterized in terms of its role in relating inputs to outputs and its relations to other functional components.” [1]

Software component architectures break the existing barriers between different programs by defining a framework where different components can interact which each other in a seamless way. There have been many attempts in defining such a framework: Toolbus [Kli93], SOM, COM [Mic92]and the CORBA standard [Sie96, OMG93]. We developed a system for interfacing the language Haskell to the COM architecture.

COM is a language, machine and operating system independent component architecture developed by Microsoft [Mic92]. It is an ‘open’ architecture and is (with additional support) compatible with the CORBA standard. COM defines a binary interaction mechanism, which makes it language and hardware independent. A good book about the internals of COM is [Rog97] but ‘Inside OLE’ by Kraig BrockSchmidt is the ultimate reference on COM technology [Bro95].

Haskell is a non-strict, purely functional language and has features as polymorphism, overloading and higher-order functions [Pet97, HF92]. Haskell uses monadic IO [GH95, JL95] to encapsulate side effects within a pure functional language. I believe that Haskell’s powerful abstraction mechanisms gives an expressiveness which make the language ideal as a ‘glue’ language for software components. However, using Haskell used to be an all or nothing approach. By making Haskell connect in a seamless way with other software components, it becomes possible to develop systems using the power of functional programming while retaining legacy software.

The work described in this Thesis contributes in three ways to the goal of making Haskell suitable for writing component software:

- A strongly typed interface to the component model of COM within the functional paradigm is developed (chapter 3).
- A compiler, called Redcard, is built to automatically translate any COM component specification to a suitable Haskell module (chapter 4).

[1] dept. of philosophy, washington university,
`http://artsci.wustl.edu/ philos/MindDict/functionalism.html`

- It is shown how the expressiveness of Haskell can be used to program software components in a very powerful way (chapter 5).

Chapter 2 will describe the COM architecture in detail. The next chapter shows how to use COM from Haskell. Chapter 4 defines a formal translation from a component specification to Haskell. Chapter 5 gives an extensive example of programming a 'real world' component and shows how the functional paradigm can be used to good effect when programming components. Chapter 6 describes how Haskell interfaces with the Automation technology. This is an important extension of COM which is used in interpreted languages and highly dynamic environments. The last chapter discusses future developments.

We have only developed a system for calling COM components, but not for creating COM components in Haskell. However there is already a a prototype system that runs Haskell components (chapter 7).

Readers not familiar with Haskell are referred to [HF92] and the Haskell home page: `http://www.haskell.org`. The monadic IO system is explained in [GH95, JW93, Wad92]. All the code examples, software and futher information can be found on `http://www.haskell.org/active/activehaskell.html`.

Chapter 2

The COM architecture

In this chapter the COM architecture will be explained from a 'clients' point of view. Only those parts important for calling COM components are discussed.

A software component is a reuseable piece of software with a well defined functionality and interface. The traditional approach to implementing software components used to be a library implementing an abstract data type. This approach has many shortcomings though. The main problem is the tight coupling between client and component. For example, function calls are resolved at link time. Each update in the component forces at least a relinking of the program.

Dynamic link libraries (DLL) bind function calls at run time. It is possible to update a component without updating the client code. Dynamic linking provided a nice way for developers to distribute updates of their software without having to relink client code. An example is the Windows operating system whose programming interface, implemented as a DLL, has been updated many times.

A DLL is still tightly coupled to the programming language used. A language has its own calling conventions and data structure layout. There was no way of describing the interface or data structures of your components in a language indepenent way. A client could easily call a function in a DLL and use the wrong number of arguments, a different calling convention, different alignment etc.

The Interface Description Language (IDL) [OMG93], which is defined as part of the CORBA standard, is a language for describing interfaces *and* associated data structures. An IDL compiler is used to translate the IDL specification to the the appropiate definitions and stubs for a specific target language. This effectively makes the interface indepenent of implementation and client language.

However there are still limitations that we wouldn't expect from a true component framework:

- DLL's are identified at run time with their file name; besides portability problems, it could give rise to ambiguities when different vendors ship a DLL with the same name.

- DLL's have to be called in the same process context as the client: an in-process context. If a component is in another process or even a remote machine, the call and all arguments need to be transferred across the process boundary. Besides being very complicated, it forced the application to have three different views of calling a component.

- There is no robust way of handling failures or sharing of DLL's across pro-

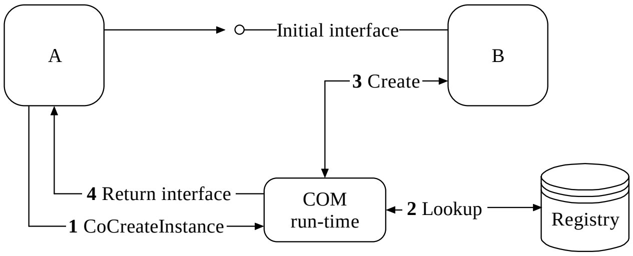


Figure 2.1: *Creation of a COM component*

cesses. A DLL for example can never know if it is still referenced and if it is safe to unload itself.

The COM framework provides solutions to the above problems. The naming problem is solved by associating a globally unique class identifier (CLSID) to each component. Clients identify their component with these CLSID's and COM takes care of the association between a CLSID and the file that provides its implementation on that computer. Different vendors can now ship components with exactly the same name because each one will be identified with their own unique CLSID. Equivalently, each set of functions gets a unique interface identifier (IID). Whenever there is an incompatible change in a function, the IID changes, so a client can never connect to an incompatible set of functions. This unique naming mechanism is essentially a 'global' type checker.

The out-of-process calling mechanism is implemented by using indirect calls to methods of the interface. It enables COM to intercept calls across process- or computer boundaries by providing a COM generated stub that marshalls the call and arguments in an invisible way. Marshalling is the process of translating data to a different format. It is needed when arguments are sent across the network. First the data has to be packaged into the network defined format and unmarshalled again at the reciever. The same process is needed when the native language data doesn't match the binary format of COM. This happens for example in Haskell. Haskell data needs to marshalled to the COM format and COM data needs to be unmarshalled to Haskell format. Chapter 3 describes the IDL compiler for Haskell which generates this marshalling code automatically. Luckily, a client of COM never needs to know about the marshalling process. A client uses all components as if they are in-process and written in the same language.

To handle the problem of resource allocation, in the presence of failure and sharing, COM introduces reference counting. Every COM compatible component is required to support this mechanism so it knows when to release itself.

COM consists of a specification ([Mic92]) which describes how COM components, should behave and be implemented, and of a supporting run-time system. The run-time system is used for creating objects, performing remote process calls and for memory allocation. The next sections discuss the COM implementation and describe the solutions to the above items in detail.

2.1 Creating an object

When a COM object is created, a pointer is returned to one of its interfaces (not to the object itself). COM provides the `CoCreateInstance` function to create an object:

```
> typedef unsigned long HRESULT;
>
> HRESULT CoCreateInstance( [in] CLSID* clsid,
>                           [in] IUnknown* outer,
>                           [in] CLSCTX context,
>                           [in] IID* iid,
>                           [out] void** iface );
```

The HRESULT value is the standard way in COM to return a success or failure value [Mic92]. The clsid argument specifies the class from which we want to create an instance. The outer parameter is used for aggregation which will be ignored in this paper [Bro95, page 101-105] and can safely be assumed NULL. The context parameter specifies if the object is run in-process, in another process or even on a remote machine. The iid parameter specifies the initial interface that is returned and the iface argument receives the pointer to the interface on success.

Figure 2.1 illustrates the creation of a COM component. It starts with a call to CoCreateInstance. The COM run-time will lookup the executable or dynamic link library that implements components of that CLSID in its database. COM will invoke the executable or library to create an instance of the object and query for the initial interface which is returned to the client. All further interaction between client and component is done via the interfaces of the component.

2.1.1 GUID's

The CLSID and IID are both a globally unique identifier (GUID). This is also equivalent to the universally unique interface identifier (uuid) defined by IDL and CORBA. A GUID is a 128 bit number. This unique number will never be reused for any other purpose in this universe. Each COM implementation should provide a way to generate these numbers[1].

The GUID design allows for coexistence of different allocation technologies but the one most commonly used incorporates a 48 bit machine unique identifier (if there is a network card) together with the current UTC time and some persistent backing store to guard against retrograde clock motion. The algorithm is capable of generating unique GUID's at a rate of 10.000.000 per second per machine for the next 3240 years to come. This is why it is quite safe to call them 'globally unique'.

2.1.2 CLSID and IID

Whenever a class is defined, it is given a CLSID. This uniquely names the component. Whenever a user installs a component, the CLSID is put into the COM database[2] together with the path to the code that implements your component. Whenever CoCreateInstance is called COM will look in the registry and associate it with the code for the component.

When an interface is defined, it gets an IID. The IID uniquely types the interface. It is the name for the total signature of the interface, including calling conventions, method order, and anything special about interface that is documented. Whenever the interface changes in an incompatible way, it should get a different IID.

[1] The program guidgen on windows platforms or the CoCreateGuid function of COM.

[2] The registry on windows platforms

2.1.3 Process contexts

The process context specifies in what kind of process the object will be created. COM takes care of hiding all the details of remote procedure calls, every component seems to run in-process for the client. The process context can have the following values:

`INPROC_SERVER`: Load the in-process code (DLL) which creates and manages objects of this class.

`INPROC_HANDLER`: Load the in-process code (DLL) which implements the client side structures of this class while instances of it are accessed remotely. An object handler generally implements object functionality which can only be implemented in-process, relying on a local server for the rest of the implementation.

`LOCAL_SERVER`: Launch a seperate process (EXE) which creates and manages objects of this class.

`REMOTE_SERVER`: Launch a separate process (EXE) on another machine which creates and manages objects of this class.

`SERVER`: All the server variants.

`INPROC`: All the in-process variants.

`ALL`: All of the above.

2.1.4 IDL

IDL is used describe the interface of a component. As an example, the IDL specification of the media control will be given. The media control is part of DirectX; a component framework for doing (fast) graphics and sound. It is build using COM and is used in games and other demanding graphics applications. The media control is one component of DirectX and can be used to play sound. The class that implements the media contol interface is called “filGraphManager”. The IDL specification reads:

```
> [uuid(56A868B1-0AD4-11CE-B03A-0020AF0BA770)]
> interface IMediaControl : IDispatch {
>    HRESULT Run();
>    HRESULT RenderFile([in] BSTR strFilename);
>    ...
> };
>
> [uuid(E436EBB3-524F-11CE-9F53-0020AF0BA770)]
> coclass FilgraphManager {
>    [default] interface IMediaControl;
>    interface IMediaEvent;
>    ...
> };
```

Items between brackets are called attributes in IDL. They give extra information about the declarations. The `uuid` attribute specifies the GUID. For the interface, this is the IID and for the coclass it is the CLSID. The interface `IMediaControl` inherits[3] from another interface `IDispatch` and exposes two methods. Both methods

[3] Type inheritance

can fail and therefore return a HRESULT value. The RenderFile takes an argument of type BSTR. This is a unicode string with length information. The in attribute makes this into an input argument. The out attribute can be used for arguments that receive a value.

The coclass declaration defines a class. It also specifies which interfaces this class supports at least. As we shall see later, this information is not necessary since it should be queried for at run-time. An IDL compiler translates this specification to a target language. The MIDL[4] compiler can translate IDL to C or C++. The following C++ program creates an instance of the media control:

```
> #include <objbase.h>                // basic com functions
> #include "media.h"                  // MIDL generated definitions
>                                     // for this component
> void main() {
>   IMediaControl* media  = NULL;
>   HRESULT        hr     = S_OK;
>
>   hr = CoInitialize(NULL);          //initialize COM
>   if (FAILED(hr)) {...};
>
>   hr = CoCreateInstance(            //create instance
>             CLSID_FilGraphManager   //the class
>             NULL,
>             CLSCTX_INPROC_SERVER    //process kind
>             IID_IMediaControl       //initial interface
>             &media  );              //the result
>
>   ...
> }
```

The MIDL compiler translates the IDL definitions in appropiate C++ declarations. For example, the uuid attribute of the coclass is translated to the constant CLSID_FilGraphManager. Without such a compiler, using COM would be extremely cumbersome since all definitions would have to be written by hand. I have made an IDL compiler for Haskell, called Redcard. Later chapters will describe this in detail. Redcard will automatically generate all the definitions needed to use a component from Haskell.

2.2 Interfaces

When an object is created, only one initial interface is returned. The IUnknown interface gives access to all other interfaces that an object supports. Since every COM interface is required to inherit from this interface, this functionality is accessable from every interface.

2.2.1 The IUnknown interface

The IUnknown interface is defined as ([Bro95, page 82]:

```
> typedef unsigned long ULONG;
>
> [uuid(00000000-0000-0000-C000-000000000046)]
```

[4] Microsoft IDL compiler.

```
> interface IUnknown {
>   HRESULT QueryInterface( [in] IID* iid,
>                           [out] void** iface );
>   ULONG   AddRef(void);
>   ULONG   Release(void);
> };
```

The most important function is the QueryInterface function. It takes an IID of an interface and if the object supports this interface, it returns a pointer to that interface in the iface argument. This method is used to query the functionality of an object at run-time.

2.2.2 Identity and equality

It is required that any query for the interface IUnknown on an object always returns the same actual pointer value in iface. Two interfaces can now be tested for equality: QueryInterface is called on both interfaces with the IID for IUnknown and the results are compared. Since this requirement only mentions the IUnknown interface, this still allows for sophisticated implementations which build method tables on the fly for other interfaces.

2.2.3 Consistency

Queries for interfaces on an object should behave consistent, they either always fail or succeed. This requirement allows for a setup phase in an application where all the interfaces are queried and saved for later use.

2.2.4 Equivalence

Furthermore it is required that QueryInterface is symmetric, reflexive and transitive with respect to the supported set of interfaces. If a given object (ObjectABC) for example supports the interfaces IA,IB and IC and we have the following C++ program:

```
> CoCreateInstance( CLSID_ObjectABC, NULL, CLSCTX_ALL, IID_IA, &ifaceA );
```

Now because of reflexivity the following call must succeed:

```
> ifaceA->QueryInterface( IID_IA, &ifaceA );
```

Symetricity will make the following calls succeed:

```
> ifaceA->QueryInterface( IID_IB, &ifaceB );
> ifaceB->QueryInterface( IID_IA, &ifaceA );
```

and transitivity will makes these calls succeed:

```
> ifaceA->QueryInterface( IID_IB, &ifaceB );
> ifaceB->QueryInterface( IID_IC, &ifaceC );
> ifaceA->QueryInterface( IID_IC, &ifaceC );
```

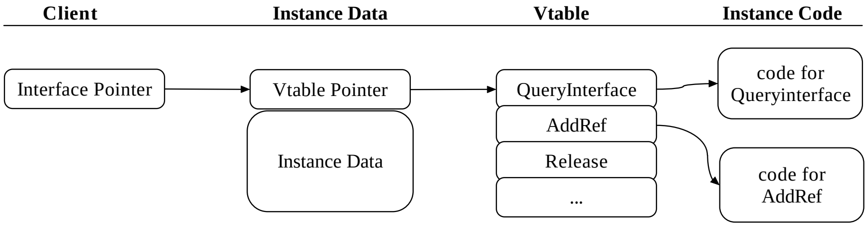


Figure 2.2: *Calling a method on an interface*

2.2.5 Reference counting

IUnknown supports 2 other methods: AddRef and Release. COM uses a reference counting mechanism to manage the life time of the objects. AddRef increments the reference count and Release decrements it. If the reference count drops to zero the object itself is responsible for releasing its resources and uninitializing itself. This mechanism is necessary because one object can service many different clients possibly running in different processes or machines.

In practice, reference counting puts quite a burden on the programmer to be sure to match each call to AddRef with a call to Release, [Bro95, page 83-90] devotes many pages to the subject. In Haskell, the garbage collector is used to take care of reference counting automatically (see section 3.5).

2.3 Calling a method

An object can be created and all its interface can be queried. How is an actual method call performed ?

An interface is at run time nothing more than (deep breath): a pointer to a pointer which points to a table of funcion pointers. A method is called using an indirect call on this table. See figure 2.2 which illustrates the layout of an interface at run time. Suppose we want to call the QueryInterface method. First the interface pointer is dereferenced, which gives the *virtual method table* (Vtbl) pointer. Dereference this pointer with the right offset (0 for QueryInterface) which gives the code address of the method. The first parameter to a method is always the current interface pointer so a method of an object has access to its own function table and, more important, the objects local data directly following the method table pointer. Note that a client can't use this local data directly. Coded in C, a method call would look like:

```
> typedef HRESULT (*QueryInterface)( void* this,
>                                    IID* iid, void** obj);
>
> HRESULT hr = (QueryInterface)(*iface)[0])( iface, iid2, &iface2 );
```

Note that any language able to call a function through indirection is able to do this: COM is language independent.

2.3.1 Calling conventions, data layout

When you make a call as in the previous example, you have to be sure that the server also uses the C calling convention. Part of your interface signature

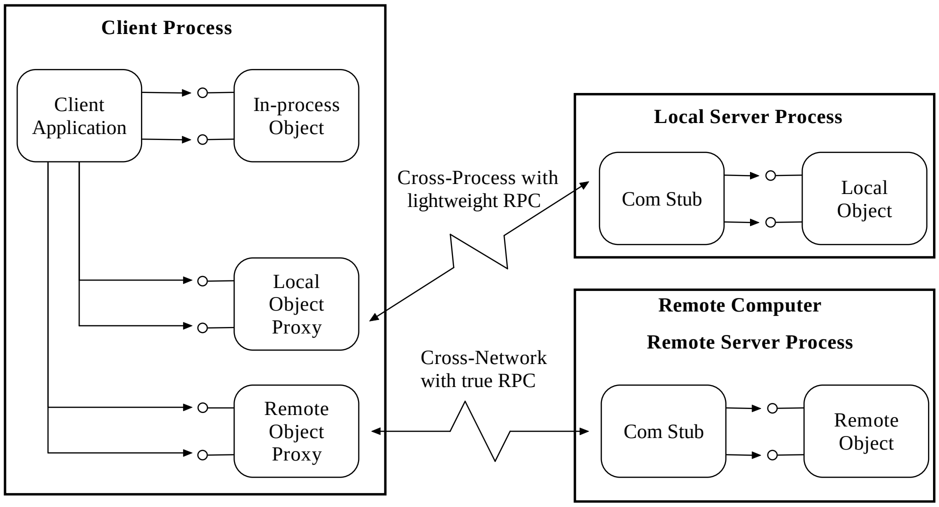


Figure 2.3: *A client always calls in-process code, COM povides a transparent RPC.*

is the calling convention and data layout. IDL allows you to specify the calling convention and data structures used by your interface. In theory they could be anything as long as the client and server agree, but in practice they will almost always match your C++ compiler conventions.

2.3.2 Encapsulation.

Since COM provides no access whatsoever other than a method call to the object, it enforces encapsulation. It can also be seen as a consequence of the type inheritance used by COM (in contrast with implementation inheritance) [Wil90]. The implementation of a component can always be updated without affecting the clients (as long as the semantics of the methods is preserved).

2.3.3 Polymorphism.

Since methods are called indirectly via a method table, different code specific for that object could be invoked (ad-hoc polymorphism, [WB89]). For example, a list of objects, each supporting the OLE interface `IViewObject`([Bro95, page 539-551]), will invoke a different `Draw` method, specific to drawing that particular object, if each objects `Draw` method is called.

2.3.4 Remote transparency.

It is the indirection through a method table which allows COM to intercept inter process (IPC) and remote process (RPC) calls. When a COM component is actually running on a different machine or process, COM will create a proxy method table. The table will contain pointers to special marshalling code and make a IPC/RPC call. This way the whole IPC/RPC mechanism is totally transparent to the client and all calls seem to be in-process. Figure 2.3 shows a client that connects to three components in different contexts: an in-process, a local-process and a remote process.

2.3.5 Memory management.

Components will share data by calling each other through interface functions. This raises the question of who is responsible for allocating and releasing complex data structures. This problem is solved by the **in** and **out** attributes in the IDL language. Each parameter in a method will be annotated with [**in**]-put parameter, [**out**]-put parameter or a [**in,out**] parameter. The following rules must be obeyed:

[**in**]: Allocated and released by the caller.

[**out**]: Allocated by the callee, released by the caller.

[**in,out**]: Allocated by the caller, then released and reallocated by the callee if necessary, and the caller is again responsible for releasing the final result.

Note that the caller is always responsible for releasing the resource. In the latter two cases both callee and caller have to agree on the memory allocator being used. COM provides a standard memory allocator for these cases. Whenever ownership of a piece of memory is being passed through a COM interface or between client and COM library, this memory allocator has to be used.

2.4 Example in two languages

We have now collected enough knowledge to put theory into practice and give a little example. Using the previous media control and MIDL, we can write the following program to play a nice tune:

```
> #include <objbase.h>                  //basic COM functionality
> #include <stdio.h>
> #include <conio.h>
> #include "media.h"                    //a MIDL generated header
>                                       // file for this object
>
> //the CLSID for the object, we get this from the IDL source
> const CLSID CLSID_FilGraphManager
>    = {0xE436EBB3, 0x524F, 0x11CE,
>                       {0x9F,0x53,0x00,0x20,0xAF,0x0B,0xA7,0x70}};
>
> void main() {
>       IMediaControl* media = NULL;
>       BSTR bstr            = NULL;
>       HRESULT hr           = S_OK;
>
>       hr = CoInitialize(NULL);                //intialize COM
>       if (FAILED(hr)) return;                 //check result
>
>       hr = CoCreateInstance(                  //Create an instance
>               CLSID_FilGraphManager,          //The class
>               NULL,
>               CLSCTX_INPROC_SERVER,           //what kind of process ?
>               IID_IMediaControl,              //initial interface ?
>               (void**)&media );               //the result
>       if (SUCCEEDED(hr)) {
>
>       bstr = SysAllocString( "media.wav" );   //alloc a system string
>       if (bstr != NULL) {
>
>       hr = media->RenderFile( bstr );         //load the sound file
>       SysFreeString( bstr );                  //free the string
>       if (SUCCEEDED(hr)) {
>
>       hr = media->Run();                      //play the sound
>
>       printf("press key to quit...");
>       getch();                                //wait for a key
>       }}}
>
>       if (media) media->Release();            //release the interface
>
>       CoUninitialize();                       //uninitialize COM
>       return;
>  }
```

COM maps nicely to C++ since the invokation mechanism of COM is exactly the same as for C++ virtual methods. The `->` operator can be used for method invokation. However, at the same time, many low level details are exposed to the C++ programmer like explicit handling of errors, calling the `Release` method and allocation of system strings. The same example in Java is already a lot simpler, and in Haskell even more. Using the Redcard compiler, a Haskell module is generated which makes the component available from Haskell. The following program is equivalent to the one in C++:

```
> module Main where
> import Com                          -- import basic COM functionality
> import Media                        -- import Redcard generated module
>                                     -- for the component
>
> main  = comRun $                    -- 'comRun' does COM (un)initialize
>       do media <- comCreateInstance
>                      clsidFilgraphManager     -- the class
>                       Nothing
>                      InProcess           -- process context
>                      iidIMediaControl    -- intial interface
>
>          media # renderFile "media.wav"  -- load the sound file
>          media # run                     -- play the sound
>
>          putStr "press a key to quit"
>          getChar                         -- wait for a key
```

The `$` operator denotes function application and can be used to replace parenthesis: `comRun $ do ...` is equal to `comRun (do ...)`.

The `comRun` method does all the COM initialization and finalization (section 3.5). `comCreateInstance` is the Haskell equivalent of the primitive `CoCreateInstance` call. The next chapter describes all these functions in detail.

Since Redcard also generates marshalling code for arguments, the `"media.wav"` string is automatically converted to a `BSTR` type (see section 4.4.2). Objects will automatically be garbage collected (section 3.5) and no calls to `Release` are neccessary. Redcard automatically uses the `IO` exception mechanism to test on `HRESULT` values.

With all this support, the haskell example is a lot more consise than the one in C++. By carefull design, our 'support' is written in Haskell itself and therefore completely extensible and customizable, in contrast with VB or MS Java where all the support is built into the compiler itself. I hope to show in chapter 5 that this a great advantage when programming COM components.

The next chapter describes all the details of using Redcard and the associated libraries.

Chapter 3

Redcard

I developed a tool called *redcard* (a successor to green-card [Sim97]). Redcard automatically generates a suitable Haskell module from an interface specification. This generated Haskell module takes care of the following items:

- It marshalls all data types from Haskell world to COM and back;
- It maps the defined IDL interfaces and data types to a suitable Haskell definition;
- It creates the primitive functions needed to call methods of the interfaces.

Redcard is completely written in Haskell and since it uses COM components, it is used to bootstrap itself. Redcard consists of both a compiler to Haskell modules and a set of Haskell modules to give run-time support. See figure 3.1 for an overview. This figure gives the compilation steps taken for the example given in section 2.4. The process starts with a specification of an interface using IDL, the interface description langage. However, Redcard does not translate IDL files directly, it uses a type library[1].

3.1 The type library

A type library is a binary description of an IDL file[2]. It is actually a persistent COM component itself. Type libraries are used instead of IDL for the following reasons:

- Almost every component ships with a type library but without its IDL source.
- It gave experience with complex COM objects.
- It saved the work of writing an IDL parser.
- Redcard would use a complex COM component itself and is actually bootstrapped with itself, which created a very good test environment.

[1] At the time of writing, Sigbjorn Finne has made a real IDL compiler for Haskell called H/Direct [Sig98]

[2] Use `oleview.exe` to view the type libraries on your system

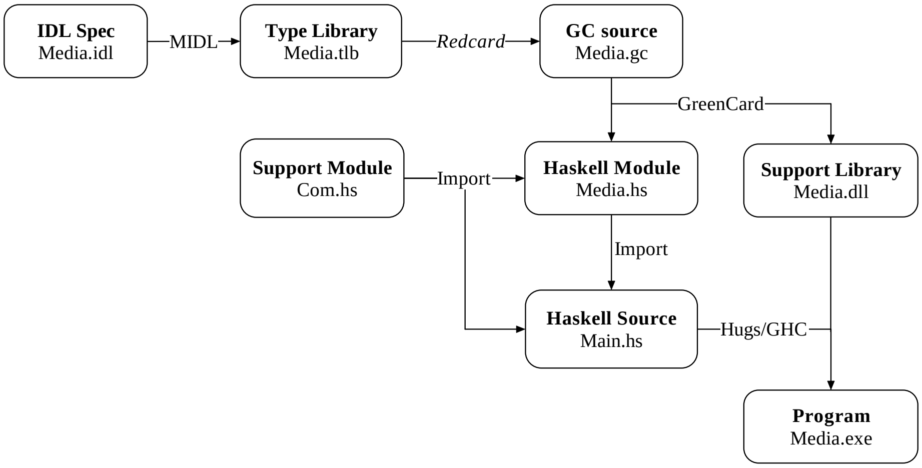


Figure 3.1: *Compilation scheme with Redcard*

If a component ships with an IDL file without a type library, the IDL needs to be compiled to a type library. On windows platforms the programs `mktyplib` and `midl` both are able to translate IDL to a type library. For our running example:

```
> midl /tlb media.tlb media.idl
```

3.2 Running the compiler

Redcard translates a type library to a Haskell module containing green-card source. It expects as the first argument the type library[3] and as a second argument the name of the Haskell file to generate. It recognises the following options:

-auto: Translates all definitions in such a way that they use automation instead of pure COM, this is discussed in chapter 6. Modules compiled this way consist just of Haskell code and do **not** need a separate green-card run or support library.

-oldgc: This instructs redcard to generate code for the 'old' greencard. The current Hugs version (1.4 beta) uses the old greencard. Since the new greencard still changes a lot, this option is recommended.

If the file name extension is not specified, redcard appends `.ss` for old green-card files, `.ghs` for new green-card files and `.hs` for automation modules.

The following command generates an automatic green-card module `explorer.ghs` for the internet explorer object model (version 3).

```
> redcard  c:\windows\system\shdocvw.dll explorer
```

[3] The library is allowed to be embedded in another file, for example in `*.exe`, `*.dll` or `*.ocx` files

3.3 Running green-card

Green-card tranlates a green-card module to some compiler specific code which interfaces to C. For the old green-card[4] redcard provides a standard makefile (`com.mak`) to translate your module. Greencard will generate both a Haskell module and a supporting DLL to link with your program.

3.4 Using the module

After running green-card, the generated Haskell module can be used to access the COM component. Import the support module `Com` and the generated module and the component can be used in Haskell (see section 2.4). The programmer using the component needs to know the Haskell representation of a component and the COM services provided by Haskell. The next sections will explain COM as seen by the Haskell programmer, while the next chapter will give all the details of the translation from IDL to Haskell.

3.5 The Com module

`Com` is the main module used by a Haskell programmer. Only this module needs to be imported to use COM components. It provides all the basic funtions needed to use COM components. The `(#)` operator is used to call methods on objects. It has the same function as the `.` in Java or the `->` in C++. Using the example from section 2.4:

```
> media # renderFile "media.wav"
> media # run
```

The operator is simply defined as:

```
> (#) :: a -> (a -> b) -> b
> obj # method  = method obj
```

In Haskell, methods are therefore just functions that take the this/self pointer as their last argument. The previous example could be written as:

```
> renderFile "media.wav" media
> run media
```

A nice property of the definition of `(#)` is its generality. It is not specific to COM objects and could be applied to many functions to write code in an object oriented style. It is possible to define a data type with associated functions in Haskell and use it as if it was a COM component (this is done in chapter 6 to deal with Automation interfaces). In retrospect, applying the this-pointer as the last argument is obvious for a functional language but it took quite some false starts to see this, maybe because all other (first order) languages take the this-pointer as an implicit first argument.

Since `(#)` is a first class value (unlike `.` or `->`) it is easy to provide extensions. The Com module provides the `withObject` function to apply a list of methods to an object:

[4] This section is specific for the Hugs 1.4 version 180797.

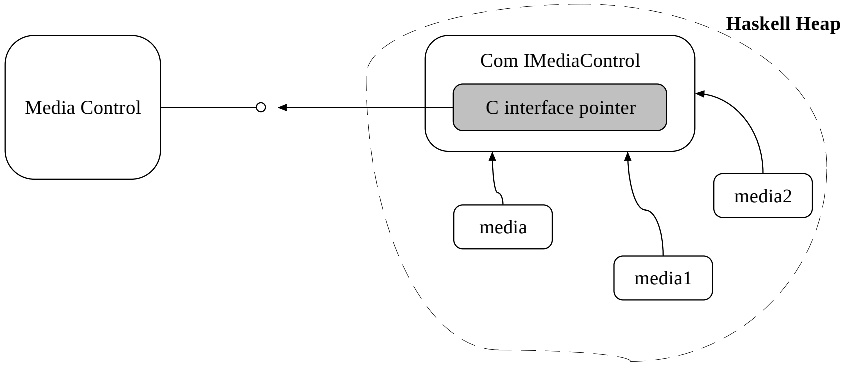


Figure 3.2: *One reference outside, multiple references within Haskell*

```
> withObject obj methods        = sequence (map (obj#) methods)
```

For example:

```
> withObject media
>        [ renderFile "media.wav"
>        , run
>        ]
```

Actual interface pointers, like `media`, are represented in Haskell by a value of type `Com a`, where `a` is the type of the interface. The methods of the media control therefore have the following signatures:

```
> run        :: Com IMediaControl -> IO ()
> renderFile :: String -> Com IMediaControl -> IO ()
```

The `IMediaControl` interface type and the methods are generated by Redcard. The methods are strongly typed since you can only apply an interface of type `IMediaControl` to them.

The `Com` abstract data type is implemented using so called foreign objects. Any interface pointer will automatically be garbage collected without an explicit call to `Release`. In Haskell we never have to deal with the reference counting mechanisms of COM, a great advantage to languages as C++ or Visual Basic. This is 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. The idea is that you can have many references to an interface inside Haskell, but there will be just one true reference to the interface outside Haskell (see figure 3.2). Whenever all references within Haskell are gone, the garbage collector will remove the outside reference after calling `Release` on the object.

The media control also supports the `IDispatch` interface. One of the methods of `IDispatch` is:

```
> getTypeInfoCount :: Com IDispatch -> IO Int
```

However, this method can't be called using `media` since its type is `Com IMediaControl`. The `QueryInterface` function is needed to get an interface pointer to the `IDispatch` interface. `QueryInterfae` takes the the IID of the requested interface and returns a pointer to the requested interface. In C++ or Java, the result has to be cast to the right interface type since the type system views the IID as just a number and can't know the type of the result. In Haskell, IID's are packed into the data type `Interface a` where `a` denotes the type of the interface. Redcard automatically generates functions with the name of the interface of this type:

```
> iidIMediaControl :: Interface IMediaControl
> iidIDispatch     :: Interface IDispatch
```

These functions name and type an interfaces in Haskell. There is a function `iid` to get the IID of an interface:

```
> iid :: Interface a -> IID
```

Since IID's are now strongly typed, it is possible to write a wrapper around `QueryInterface` (called `queryInterface`), which creates a typed connection between the interface type and the actual pointer to the interface:

```
> queryInterface ::  Interface a -> Com b -> IO (Com a)
```

`queryInterface` calls `QueryInterface` on any interface pointer `Com b` [5] with the IID contained in `Interface a` and returns an interface pointer `Com a`:

```
> dispatch <- media # query iDispatch
> count    <- dispatch # getTypeInfoCount
```

In the above example, `query` is used in the following type context:

```
> queryInterface :: Interface IDispatch -> Com IMediaControl -> IO (Com IDispatch)
```

This unusual use of polymorphism elegantly captures the type of `QueryInterface` without having to resort to typecasts. To my knowledge, Haskell is the first language with this feature. During the design of redcard I first tried to achieve this behaviour using type classes (without success), while overlooking this simple and elegant solution.

Since `Com a` is an abstract data type, we need a function to create an initial interface pointer. The function `CoCreateInstance` returns an intial interface pointer in COM. It needs a CLSID, a process context and the IID of the initial interface (see the previous chapter). A CLSID is encapsulated in the data type `CLSID`. Redcard automatically generates functions with the name of a class that return a `CLSID`. The function `clsid` is provided to extract the CLSID from a `CLSID`. The component implementing the `IMediaControl` interface is called `FilGraphManager` and Redcard will generate a function of type:

```
> clsidFilGraphManager :: CLSID
```

Process contexts are encapsulated in the type `CLSCTX`:

```
> data CLSCTX   = CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER | ...
>                 LocalProcess | InProcess | ServerProcess | AnyProcess
```

`CoCreateInstance` is now elegantly encapsulated in the function `comCreateInstance`:

```
> comCreateInstance :: CLSID -> Maybe (Com b) -> CLSCTX -> Interface a -> IO (Com a)
```

[5] since every interface inherits from `IUnknown`

Again, polymorphism is used to create a typed connection between the passed IID (`Interface a`) and the returned interface pointer (`Com a`). A media control is now created with:

```
> media <- comCreateInstance clsidFilGraphManager Nothing LocalProcess iidIMediaControl
```

I have also defined some more friendly creation functions that can take a ProgID or VersionIndependentProgID [Bro95, page 70–72]. A programmatic identifier or ProgID identifies a class with a human readable name. The format of a ProgID is `vendor.component.version`. This format is reasonably unique and used to create objects in environments as Visual Basic instead of the ugly CLSID. A VersionIndependentProgID is the same as the ProgID but suprisingly without a version number.

The function `comCreateObject`:

```
> comCreateObject :: String -> Interface a -> IO (Com a)
```

can be used to create the Internet Explorer component using its VersionIndependentProgID:

```
> explorer <- comCreateObject "InternetExplorer.Application" iidIWebBrowser
```

The ProgID's can be viewed using the `ole2view` tool on windows platforms. Other, more exotic, creation functions are:

```
> comGetActiveObject  :: CLSID -> Interface a -> IO (Com a)
> comGetObject        :: String -> Interface a -> IO (Com a)
```

`comGetActiveObject` tries to connect to an already running object [Bro95, page 665-667]. `comGetObject` takes a ProgID, VersionIndependentProgID or a file name and tries to connect to the object or creates a fresh instance. In all cases the `CLSCTCX` is `AnyProcess`.

Since COM methods can be side effecting (or become side effecting) all COM calls have the IO type. The IO monad encapsulates side effects and exceptions within a pure functional language as Haskell. For an introduction about monadic IO in Haskell see [GH95]. In one way the IO monad restricts our expressiveness because of the need to sequentialize our expressions but it also gives us exception handling for free (compare C++ in section 2.4). COM requires all methods to return a HRESULT which is used to flag errors. Redcard automatically tests this value and raises an exception on error. The library function `catch` is used to handle exceptions:

```
> render fname media =
>   media # renderFile fname
>    `catch` \err -> media # renderFile "default file"
```

The following functions deal with COM errors:

```
> comFail :: String -> IO a
> isComError :: IOError -> Bool
> comGetErrorString :: IOError -> String
```

`comFail` generates an COM exception. `isComError` tests the error argument in a `catch` function if it is a COM exception and `comGetErrorString` retrieves the error message, also in case of an IO or User error. For example:

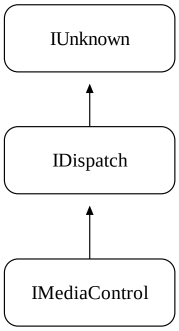


Figure 3.3: *Inheritance tree for the media control*

```
>   comFail "hello world"
>    `catch` \err -> if (isComError err)
>                    then  putMessage (comGetErrorString err)
>                    else  fail err
```

If no catch blocks are installed, all COM errors are eventually caught by the function comRun which presents the user a nice message box with error information. The function comRun also does a garbage collect to free COM objects still alive and takes care of COM initialisation and cleanup. The function comRun should always encapsulate your program if you use COM and is generally called right after main:

```
> comRun :: IO a -> IO ()
>
> main  = comRun $
>         do media <- comCreateObject "filGraphManager" iidIDispatch
>            ...
```

This wraps up the discussion of the Com module. The next chapter will deal with the exact mapping from IDL to Haskell done by Redcard and discuss the implementation of the abstract data types and generated methods. I will end this chapter with some ideas for future work.

3.6 About inheritance

With the polymorphic type Com a it is not possible to express inheritance relationships. Take for example the inheritance tree in figure 3.3. It would be safe to pass a Com IMediaControl value to a funcion expecting a Com IDispatch value, but the type system can not infer that. In Haskell you always need to query for an interface even if it is in the same inheritance tree. One exception is the IUnknown member functions. Since every object inherits from this interface, the methods can accept a Com a instead of a Com IUnknown but this is a special case and just used within the query function. Allthough sometimes inconvenient to query for an interface down in the inheritance tree, it doesn't restrict our expressiveness.

Type classes ([Jon93]) introduce an interesting way of expressing inheritance with constraints. Every method would take a Com a pointer with a constaint on a to the interface where it belongs:

```
> addRef           :: Unknown a => Com a -> IO ()
> getTypeInfoCount :: Dispatch a => Com a -> IO Int
> run              :: MediaControl a => Com a -> IO ()
```

The inheritance relation is expressed with class constraints using empty classes:

```
> class Unknown a
> class Unknown a => Dispatch a
> class Dispatch a => MediaControl a
```

The function `createInstance` has type:

```
> createInstance :: Unknown a => ClassId -> CLSCTX -> IO (Com a)
```

We would now like to write:

```
> media <- createInstance filGraphManager LocalProcess
> media # getTypeInfoCount
> media # run
```

The type system could infer the type `media :: Media a => Com a` from the calls to `run` and `getTypeInfoCount`. However it would also complain about an ambiguous type `Media a => IO ()` for this program. The second problem is that `createInstance` needs the constraint at run-time to know the initial interface to query for. I think that both problems can be resolved by allowing explicit access to the constraints at run time (the dictionaries) by making instance declarations first class values in Haskell. This would allow the type system to automatically infer types at compile time and to automatically query at run-time if the component satisfies the constraint (provides the required interface). The language Mondrian ([Eri97]) does seem to offer the desired expressiveness, but alas, there is not yet an implementation to work with.

Chapter 4

The mapping

This chapter describes the mapping from IDL to Haskell. A lazy functional language like Haskell has a completely different language model than the object oriented IDL model. The translation to Haskell is therefore far less trivial than the mapping to conventional languages like C++ or VB.

Data types in languages as C++ map almost immediately to COM data types, for example the `char*` or `int` type and even user defined types as `struct`. This is not the case for a pure functional language as Haskell that uses polymorphic algebraic data types like `type String = [Char]` and `data Bool = True | False`. For every COM call, all arguments need to be marshalled from Haskell to COM and every result (and arguments passed by reference) needs to be unmarshalled.

Besides having to marshall the values, there has to be a mapping from the IDL types like `char*` to suitable Haskell types as `String`. Some IDL types are not easy to represent in a strongly typed functional language (most notably interface types). The previous chapter already showed that this requires a non-obvious (but satisfying) translation.

The calling mechanism of COM matches the more conventional system languages. The lazy evaluation strategy doesn't even 'call' functions. From a Haskell point of view, COM functions are first-order, impure and strict. Such functions can be used within the Haskell language by giving them the IO type. However, we still nead to deal with building a proper call stack and calling a method by offset in the vtable. The current solution is to use green-card. Green-card allows the Haskell programmer to specify C functions within Haskell and translates these C functions to Haskell functions that execute the C code [Sim97]. Redcard now enables COM calls by generating some C code that makes the actual call to a method. Chapter 6 deals with dynamic method invokation, called automation. Automation modules do not need separate green-card code but can be used directly from Haskell.

4.1 A translation example

We will start with an example of the translation done by Redcard. Our running example, the media control, is described in IDL as:

```
> [uuid(56A868B0-0AD4-11CE-B03A-0020AF0BA770)]
> library QuartzTypeLib
> {
> importlib("stdole32.tlb");
>
> [uuid(56A868B1-0AD4-11CE-B03A-0020AF0BA770)]
> interface IMediaControl : IDispatch {
>    HRESULT Run();
>    HRESULT RenderFile([in] BSTR strFilename);
>    ...
> };
>
> [uuid(E436EBB3-524F-11CE-9F53-0020AF0BA770)]
> coclass FilgraphManager {
>    [default] interface IMediaControl;
>    interface IMediaEvent;
>    ...
> };
>
> ...
> };
```

If there is no type library, the IDL source needs to be translated to a type library first; The IDL `library` declaration is used to denote a type library. The media control has its type library included in the `quartz.dll` module[1] and can directly be translated by Redcard:

```
> redcard c:\windows\system\quartz.dll media.ghs
```

Redcard will generate the following `media.ghs` file:

```
> ------------------------------------------------------------
> -- redcard, version 0.1, (c) 1997 Daan Leijen
> -- This file is automatically generated from a type library
> ------------------------------------------------------------
> module Media (  filgraphManager
>               , IMediaControl
>               , iMediaControl
>               , run
>               , renderFile
> ...
>
> import OleMarshall
> import OleTypes
> import StdDis
>
> ------------------------------------------------------------
> -- coclass filgraphManager
> ------------------------------------------------------------
> clsidFilgraphManager   = ClassId (newGuid (-466162765) 21071 4558
>                                   [159 ,83 ,0 ,32 ,175 ,11 ,167 ,112])
>
> ------------------------------------------------------------
> -- interface IMediaControl
> ------------------------------------------------------------
> data IMediaControl = IMediaControl
>
> iidIMediaControl :: Interface IMediaControl
> iidIMediaControl   = Interface (newGuid 1453877425 2772 4558
>                                   [176 ,58 ,0 ,32 ,175 ,11 ,167 ,112])
>
```

[1] Use `oleview.exe` to get this information.

```
> ------------------------------------------------------------
> -- method IMediaControl.run
> ------------------------------------------------------------
> run :: Com IMediaControl -> IO ()
> run self   = do {  xself <- marshallSelf self
>                  ; res <- primRun xself
>                  ; checkHRESULT res
>                  ; return ()}
>
> %fun primRun :: Ptr -> IO Int
> ...
>
> ------------------------------------------------------------
> -- method IMediaControl.renderFile
> ------------------------------------------------------------
> renderFile :: BSTR -> Com IMediaControl -> IO ()
> renderFile strFilename self
>         = do {  xstrFilename <- marshallBSTR strFilename
>               ; xself <- marshallSelf self
>               ; res <- primRenderFile xself xstrFilename
>               ; checkHRESULT res
>               ; releaseBSTR xstrFilename
>               ; return ()}
>
> %fun primRenderFile :: Ptr -> Ptr -> IO Int
> %call (ptr self) (ptr strFilename)
> %code
> % typedef int (__stdcall * _method)(void* self ,wchar_t* strFilename);
> % _method primRenderFile;
> % primRenderFile = (*((_method*)(*((char**)self)+ 44)));
> % res = primRenderFile(self ,strFilename);
> %result (int res)
```

This is fairly typical for a Redcard generated module. It also shows why you definitely don't want to write it yourself. From top to bottom, we see that the IDL `library` declaration is translated to a Haskell module with the same name. The `coclass` declaration just generates the `CLSID` function as described in the previous chapter. The `interface` declaration is translated to both a new Haskell type and a function that returns the IID in Haskell.

The methods have a more complex translation. The `run` method first marshalls the self-pointer to an IDL pointer (`marshallSelf`). It calls the primitive `primRun` function and automatically performs a test on the `HRESULT` to raise a possible exception. The `renderFile` method also needs to marshall its `BSTR` argument. A `BSTR` is in Haskell just a type synonym for `String` but in COM it is defined as a unicode string prefixed with length information:

```
> typedef struct _BSTR {
>       int length;
>       [sizeis(length)] wchar_t data[];
> } *BSTR;
```

`renderFile` first marshall the string to this format before calling the primitive function and releases the store held by the BSTR afterwards. The primitive function `primRenderFile` is defined using green-card directives. The generated C code calls the `renderFile` method at offset 44 in the vtable.

A programmer using COM never needs to know the details of the marshalling process. To use a component, a Haskell programmer just needs to know the interface and the Haskell model of COM, which was presented in the previous chapter. The following table gives a general idea of the mapping done by Redcard as seen by the Haskell programmer:

library: A module with the same name of the library (or the same as the user specified file name).

interface: 1) An abstract interface type with the name of the interface. 2) A constructor function with the name of the interface prefixed with `iid` of type `Interface a` where `a` is the interface type. 3) The set of methods in the interface taking as last argument the object itself of type `Com a` where `a` is the the interface type.

coclass: An constructor function with the same name as the com class prefixed with `clsid` of type `CLSID`.

enum: A data type with the name of the enumeration and a set of constructors corresponding with the enumeration values.

struct: A record type and constructor with the same name and field names as the structure.

union: A data type with name of union and its fields as constructors.

typedef: A haskell `type` definition.

In the literature, the translation is normally given ‘by example’. Maybe this is appropiate for conventional languages that closely match the COM model, but I hope to give a more formal definition of the translation process for Haskell. In the following sections the precise translation is defined, starting with identifiers and types, up to interfaces and methods.

4.2 Name translation

Because Haskell is more restrictive on its identifiers than IDL, IDL identifiers need to be translated into Haskell identifiers. In the case of a type identifier:

```
> typeID :: String -> String
> typeID (x:xs)         | isuppercase(x)  = x:xs
>                       | islowercase(x)  = (touppercase x):xs
>                       | otherwise       = ’X’:x:xs
```

and in the case of a variable/function identifier:

```
> varID :: String -> String
> varID (x:xs)          | islowercase(x)  = x:xs
>                       | x == ’_’        = x:xs
>                       | isuppercase(x)  = (tolowercase x):xs
>                       | otherwise       = ’x’:x:xs
```

In contrast to IDL, Haskell requires field and method names to be unique in the same module. Redcard therefore renames conflicting names by appending an integer (starting at 1) to a duplicate name. This allows for an easy search and replace to give the identifiers more meaningfull names.

4.3 Translation functions

The translation to Haskell is formalized using eight translation functions. For each IDL type, two translation schemes are defined which map an IDL type to a Haskell type:

$$\mathbf{T}[\![\mathrm{t}]\!], \mathbf{B}[\![\mathrm{t}]\!] :: \mathrm{IDL.type} \rightarrow \mathrm{Haskell.type}$$

The **T** (type) function maps an IDL type to a full Haskell type that shows up in the signature of a method. For example $\mathbf{T}[\![\mathrm{int}]\!] = \mathrm{Int}$ and $\mathbf{T}[\![\mathrm{IUnknown*}]\!] = \mathrm{Com\ \ IUnknown}$. The **B** (basic type) function maps an IDL type to a Haskell type that is actually passed to the primitive green-card function. The only types passed this way are an `Int`, `Float`, `Double` and `Ptr`. $\mathbf{B}[\![\mathrm{int}]\!] = \mathrm{Int}$ and $\mathbf{B}[\![\mathrm{IUnknown*}]\!] = \mathrm{Ptr}$. The **S** (sizeof) function maps types to their (COM world) size: $\mathbf{S}[\![\mathrm{int}]\!] = 4$ and $\mathbf{S}[\![\mathrm{IUnknown*}]\!] = 4$ on win32 systems. This function is needed when allocating memory to store results. All three functions are used at compile time and are part of the Redcard compiler. Of course, functions taking a type as argument (polytypic functions [JJ97]) using the $[\![t]\!]$ notation can not be defined in Haskell. Within Redcard these functions work on a data type representing types. For example the $\mathbf{T}[\![\mathrm{t}]\!]$ function could be defined in Redcard as:

```
> haskellType :: IDLType -> String
> haskellType TpInt              = "Int"
> haskellType (TpInterface name)  = "Com " ++ name
> ...
```

The functions **M** (marshall) and **U** (unmarshall) are generated for each user defined type:

$$\mathbf{M}[\![\mathrm{t}]\!] :: \mathbf{T}[\![\mathrm{t}]\!] \rightarrow \mathrm{IO}\ \mathbf{B}[\![\mathrm{t}]\!]$$
$$\mathbf{U}[\![\mathrm{t}]\!] :: \mathbf{B}[\![\mathrm{t}]\!] \rightarrow \mathrm{IO}\ \mathbf{T}[\![\mathrm{t}]\!]$$

The **M** function marshalls its argument to a value of a basic Haskell type that can be passed to the green-card function. The **U** function unmarshalls a value of a basic type to a value of a full Haskell type again. **M** and **U** are mutually inverse:

$$\mathbf{do}\ \{\ t\ \leftarrow \mathbf{U}[\![\mathrm{t}]\!]\ b;\ \ \mathbf{M}[\![\mathrm{t}]\!]\ t\ \}\ \equiv\ \mathbf{return}\ b$$
$$\mathbf{do}\ \{\ b\ \leftarrow \mathbf{M}[\![\mathrm{t}]\!]\ t;\ \ \mathbf{U}[\![\mathrm{t}]\!]\ b\ \}\ \equiv\ \mathbf{return}\ t$$

Since these functions are called at run-time to (un)marshall arguments, these functions are part of the module generated by Redcard. For example, for an integer passed as a reference, the function $\mathbf{M}[\![\mathrm{int*}]\!]$ has type $:: \mathrm{Int}\ \ \rightarrow \mathrm{IO\ \ Ptr}$. The code will allocate memory for the size $\mathbf{S}[\![\mathrm{int}]\!]$, pack the integer in the memory and return a pointer of type $:: \mathrm{Ptr}$ to the memory:

$$\mathrm{marshallIntPtr}\ \ i = \mathbf{do}\ \{\ p\ \leftarrow \mathrm{alloc}\ 4;\ \ \mathrm{packInt}\ p\ i;\ \ \mathbf{return}\ p\ \}$$

The functions **W** (write) and **R** (read) are used to write or read a Haskell value from a specific memory location. These functions are needed in situations as the previous example (`packInt`) and when array or structure fields need to be (un)marshalled. The types are:

$$\mathbf{W}[\![\mathrm{t}]\!] :: \mathrm{Ptr} \rightarrow \mathbf{T}[\![\mathrm{t}]\!] \rightarrow \mathrm{IO\ Ptr}$$
$$\mathbf{R}[\![\mathrm{t}]\!] :: \mathrm{Ptr} \rightarrow \mathrm{IO}\ \mathbf{T}[\![\mathrm{t}]\!]$$

These functions are also mutually inverse:

$$\begin{aligned}\textbf{do}\ \{\ p\ \leftarrow \mathbf{W}[\![\mathrm{t}]\!]\ p\ x;\ \ \mathbf{R}[\![\mathrm{t}]\!]\ p\ \} &\equiv \textbf{return}\ x\\ \textbf{do}\ \{\ x\ \leftarrow \mathbf{R}[\![\mathrm{t}]\!]\ p;\ \ \mathbf{W}[\![\mathrm{t}]\!]\ p\ x\ \} &\equiv \textbf{return}\ p\end{aligned}$$

For all the basic types, the above four functions are already defined by Redcard and part of the OleMarshall module. Of course, type information is not available any more and all the functions are instantiated to a specific type in Haskell. For example the function $\mathbf{W}[\![\mathrm{int}]\!]$ is defined as:

```
> packInt :: Ptr -> Int -> IO Ptr
> packInt       = primPackInt32
```

where `primPackInt32` is defined in OlePrim as a primitive C function.

Redcard will generate specialized versions of the **W** and **R** functions for composed data structures to write and read each of their fields.

A function that is also generated for these types is the **F** (free) function. This functions calls a finalization function for each of the fields in a data structure. For example, if the memory occupied by an array of BSTR strings is released, the **F** function for the array will first free all the elements of the array.

$$\mathbf{F}[\![\mathrm{t}]\!] :: \mathrm{Ptr}\ \to \mathrm{IO\ Ptr}$$

The **F** function needs to operate on the real memory pointer in order to allow specific system routines to be called by the free routine. For example, a BSTR type needs to be freed with a special system call `SysFreeString`. Note that a **F** function just takes care of releasing fields within a data structure, the memory occupied by the data is released later with a call to the `free` function of the COM heap. This is done this way since structures can be part of other structures: the fields need to be finialized but the memory is released as part of the parent structure.

The last and most important function needed is the $\mapsto$ function that translates an IDL declaration to a Haskell declaration:

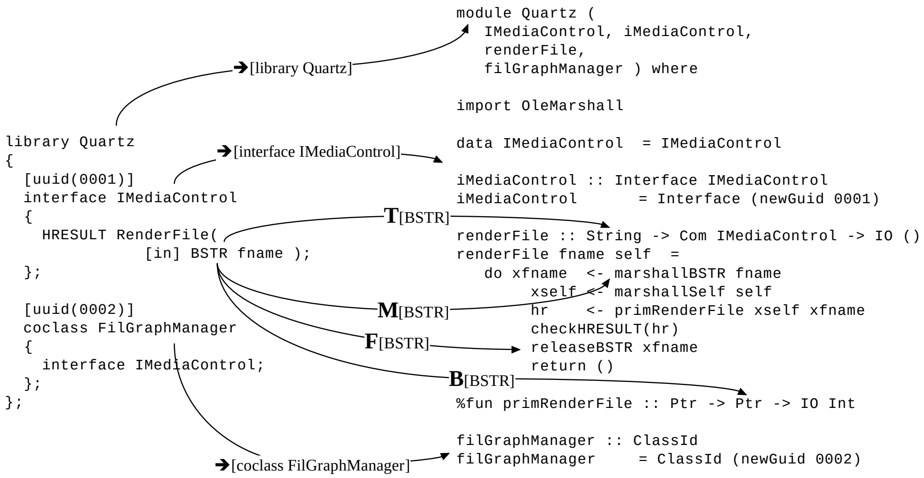
$$\mapsto\ :: \mathrm{IDL.decl} \to \mathrm{Haskell.decl}$$

For example, a `typedef` declaration in IDL is mapped to a `type` declaration in Haskell by this function. This function is therefore the actual mapping function from IDL to Haskell and it uses all other functions in this process. An example of the translation functions in action, is given in figure 4.1. It shows how some of the declarations and types are mapped from the media control to Haskell using the translation functions.

The following sections will define a translation for all basic types of IDL in terms of the above functions. After this, the *mapsto* function is defined for each IDL declaration ending with the translation of a COM method, which uses all of the defined functions to correctly perform a call from Haskell to COM.

4.4 Basic types

The basic types of IDL are already provided with a **B**, **T** and **S** function by Redcard. Figure 4.2 defines these functions for all basic types.



```
library Quartz
{
  [uuid(0001)]
  interface IMediaControl
  {
    HRESULT RenderFile(
              [in] BSTR fname );
  };

  [uuid(0002)]
  coclass FilGraphManager
  {
    interface IMediaControl;
  };
};
```

```
module Quartz (
   IMediaControl, iMediaControl,
   renderFile,
   filGraphManager ) where

import OleMarshall

data IMediaControl  = IMediaControl

iMediaControl :: Interface IMediaControl
iMediaControl       = Interface (newGuid 0001)

renderFile :: String -> Com IMediaControl -> IO ()
renderFile fname self  =
   do xfname  <- marshallBSTR fname
        xself <- marshallSelf self
        hr    <- primRenderFile xself xfname
        checkHRESULT(hr)
        releaseBSTR xfname
        return ()
%fun primRenderFile :: Ptr -> Ptr -> IO Int

filGraphManager :: ClassId
filGraphManager     = ClassId (newGuid 0002)
```

Figure 4.1: *Some translation functions applied to the media control*

B⟦octet⟧	= Byte	**S**⟦octet⟧	= 1
B⟦char⟧	= Char	**S**⟦char⟧	= 1
B⟦wchar_t⟧	= Short	**S**⟦wchar_t⟧	= 2
B⟦boolean⟧	= Bool	**S**⟦boolean⟧	= 1
B⟦short⟧	= Short	**S**⟦short⟧	= 2
B⟦int⟧	= Int	**S**⟦int⟧	= 4
B⟦long⟧	= Long	**S**⟦long⟧	= 4
B⟦unsigned char⟧	= Byte	**S**⟦unsigned char⟧	= 1
B⟦unsigned short⟧	= UShort	**S**⟦unsigned short⟧	= 2
B⟦unsigned int⟧	= UInt	**S**⟦unsigned int⟧	= 4
B⟦unsigned long⟧	= ULong	**S**⟦unsigned long⟧	= 4
B⟦void*⟧	= Ptr	**S**⟦void*⟧	= 4
B⟦HRESULT⟧	= HRESULT	**S**⟦HRESULT⟧	= 4
B⟦float⟧	= Float	**S**⟦float⟧	= 4
B⟦double⟧	= Double	**S**⟦double⟧	= 8
B⟦long double⟧	= Double	**S**⟦long double⟧	= 8
T⟦wchar_t⟧	= Char		
T⟦t⟧	= **B**⟦t⟧		

Figure 4.2: *The basic type, the Haskell type and the sizeof functions for primitive IDL types*

The **B** function is exactly the same for the **T** function for these basic types, except for a wide character, `wchar_t`, which is represented as a `Short` in the COM world. The marshall functions don't do anything except to translate wide characters:

$$\begin{aligned}
&\mathbf{M}\llbracket \text{wchar\_t} \rrbracket && = \mathbf{return}\ \cdot \text{ansitouni} \\
&\mathbf{M}\llbracket \text{t} \rrbracket && = \mathbf{return} \\
\\
&\mathbf{U}\llbracket \text{wchar\_t} \rrbracket && = \mathbf{return}\ \cdot \text{unitoansi} \\
&\mathbf{U}\llbracket \text{t} \rrbracket && = \mathbf{return}
\end{aligned}$$

The **W** and **R** functions are primitive functions defined in OleMarshall and OlePrim (see `packInt` in the previous section). None of these types need any finalization and Redcard will not generate any call to the **F** function for these types.

4.4.1 Pointers and arrays

Arrays are treated as values. A pointer to an array is explicitly represented by a pointer to an array in Redcard. Do not confuse these two notions:

```
> int arrayValue[3];
> int arrayPtr[];
```

The first array is a sequence of three integers in memory, but the second is a pointer to an sequence of integers somewhere in memory. The first has type `int [3]` while the second array gets the type `int*` in the Redcard translation. An array of type t and size n is represented in haskell as a list $[t]$ and its COM size is n times the COM size of its elements t:

$$\begin{aligned}
&\mathbf{T}\llbracket \text{t[n]} \rrbracket && = [\mathbf{T}\llbracket \text{t} \rrbracket] \\
&\mathbf{S}\llbracket \text{t[n]} \rrbracket && = n * \mathbf{S}\llbracket \text{t} \rrbracket
\end{aligned}$$

If an array is part of another data structure, the **R** and **W** functions are used to pack and unpack the array. The $\mathbf{R}\llbracket \text{t[n]} \rrbracket$ takes the adress and applies $\mathbf{R}\llbracket \text{t} \rrbracket$ to each of the n elements and builds a list of them. The **W** is the inverse operation. The $\mathbf{F}\llbracket \text{t[n]} \rrbracket$ function applies $\mathbf{F}\llbracket \text{t} \rrbracket$ to all the elements. The **U**,**M** and **B** functions are just needed when an array is passed as an argument to a function. Since arrays cannot be passed by value, these functions can stay undefined since Redcard never needs them. However, arrays can be passed by reference if a pointer to an array is used;

For pointers $t*$:

$$\begin{aligned}
&\mathbf{B}\llbracket \text{t*} \rrbracket && = \text{Ptr} \\
&\mathbf{T}\llbracket \text{t*} \rrbracket && = [\mathbf{T}\llbracket \text{t} \rrbracket] \\
&\mathbf{S}\llbracket \text{t*} \rrbracket && = 4
\end{aligned}$$

It is assumed that a pointer to a type points to an array of that type. IDL actually has attributes to define if a pointer points to an array, just one value and if the pointer can be `NULL`. Sadly, this information is lost in the type library which Redcard uses as its source. Redcard therefore defaults to the most general case and assume that a pointer always points to one or more elements and uses the empty list for the `NULL` case.

The **U**, **M** and **F** functions for a pointer type, are defined in terms of these functions for array values. An object of type $t*$ is unmarshalled by first applying $\mathbf{R}\llbracket \text{t[n]} \rrbracket$ to the pointer to read the elements, calling $\mathbf{F}\llbracket \text{t[n]} \rrbracket$ to finalize the elements and to free the memory with a call to free $:: \text{Int} \ \to \text{Ptr} \ \to \text{IO} \ ()$:

$$\mathbf{U}\llbracket \text{t*} \rrbracket\ p = \mathbf{do}\ \{\ xs\ \leftarrow \mathbf{R}\llbracket \text{t[n]} \rrbracket\ p;\ \ \mathbf{F}\llbracket \text{t[n]} \rrbracket\ p;\ \ \text{free}\ \mathbf{S}\llbracket \text{t[n]} \rrbracket\ p;\ \ \mathbf{return}\ xs\ \}$$

The marshall function allocates memory using alloc :: Int → IO Ptr , and calls $\mathbf{W}\llbracket\mathrm{t[n]}\rrbracket$ on the allocated memory to marshall the elements:

$$\mathbf{M}\llbracket\mathrm{t}^*\rrbracket\; xs = \mathbf{do}\ \{\ p\ \leftarrow \mathrm{alloc}\ \mathbf{S}\llbracket\mathrm{t[n]}\rrbracket;\ \ \mathbf{W}\llbracket\mathrm{t[n]}\rrbracket\; p\; xs;\ \ \mathbf{return}\; p\ \}$$

The free function first reads the array pointer, calls $\mathbf{F}\llbracket\mathrm{t[n]}\rrbracket$ to finalize its elements and frees the occupied memory:

$$\mathbf{F}\llbracket\mathrm{t}^*\rrbracket\; p = \mathbf{do}\ \{\ p'\ \leftarrow \mathbf{R}\llbracket\mathrm{void}^*\rrbracket\; p;\ \ \mathbf{F}\llbracket\mathrm{t[n]}\rrbracket\; p';\ \ \mathrm{free}\ \mathbf{S}\llbracket\mathrm{t[n]}\rrbracket\; p\ \}$$

These functions all use the number of elements n the pointer points to. IDL has attributes (`size_is`, `length_is`) to compute this number at run-time but this information is again not available in the type library. Redcard uses a heuristic to guess the size. In the case of **F**, the size argument is ignored since the size of the allocated block is already known to the allocator. In case **M** the length of the argument list is used: $n = \mathrm{length}\ xs$. Only the **U** case can not be solved properly, since there is simply not enough information to deduce the length at run-time. Redcard will assume a one element array. Redcard issues a warning in this case and it is easy to handpatch the generated code to unmarshall correctly. It seems that the situation doesn't occur very often in practice, but in `Oletypes.hs` are a few examples of this situation.

This leaves the **R** and **W** functions. These functions can be defined in terms of **U** and **M**:

$$\begin{aligned}
\mathbf{R}\llbracket\mathrm{t}^*\rrbracket\; p \quad &= \mathbf{do}\ \{\ p'\ \leftarrow \mathbf{R}\llbracket\mathrm{void}^*\rrbracket\; p;\ \ \mathbf{U}\llbracket\mathrm{t}^*\rrbracket p'\ \}\\
\mathbf{W}\llbracket\mathrm{t}^*\rrbracket\; p\; x \quad &= \mathbf{do}\ \{\ p'\ \leftarrow \mathbf{M}\llbracket\mathrm{t}^*\rrbracket\; x;\ \ \mathbf{W}\llbracket\mathrm{void}^*\rrbracket\; p\, p'\ \}
\end{aligned}$$

The **R** function simply reads the pointer value and applies the **U** to it. The **W** function uses the same technique. The **R** function has the same problem as the **U** function above: The length of the array is unknown. As an example of how to handpatch the generated source, a structure from the OleTypes module is used:

```
> struct BLOB {
>       int len;
>       [sizeis(length)] int* blob;
> };
```

The $\mathbf{T}\llbracket\mathrm{struct\ BLOB}\rrbracket$ function is generated as:

```
> data BLOB     = BLOB { len :: Int, blob :: [Int] }
```

The $\mathbf{R}\llbracket\mathrm{struct\ BLOB}\rrbracket$ function of this structure will be generated as:

```
> unpackBLOB p  = do { len   <- unpackInt p
>                    ; blob  <- unpackArray 1 4 (p *+ 4)
>                    ; return (BLOB { len, blob }) }
```

The exact translation of structures is described in more detail in the next section. The `*+ :: Ptr -> Int -> Ptr` function increments a pointer value. The `unpackArray` function is the Redcard primitive for the $\mathbf{R}\llbracket\mathrm{t[n]}\rrbracket$ function. Its first argument is the size of the array, the second the size of its elements. Redcard will warn that the size of the array is unknown, since the `sizeis` attribute information is lost in the type library:

```
> warning: OleTypes.BLOB.blob: array size unknown,
>                              assuming single element array
```

The generated code needs to be handpatched by replacing the 1 with the length of the array:

```
> unpackBLOB p  = do { len   <- unpackInt p
>                    ; blob  <- unpackArray len 4 (p *+ 4)
>                    ; return (BLOB { len, blob }) }
```

4.4.2 Strings

IDL defines two types of strings: normal 8-bit ascii strings of type `char*` and 16-bit wide character or unicode strings of type `wchar_t*`. COM adds the BSTR type to this list. Redcard maps all these kinds of string to the Haskell `String` type. The Haskell programmer is completely protected from any character string conversions! From my experience with C++, this is a great feature.

To differentiate between an array of characters and a string which is zero terminated, the `string` attribute needs to be used in the IDL specification.

$$
\begin{aligned}
&\mathbf{B}[\![\text{[string] char*}]\!] &&= \text{Ptr}\\
&\mathbf{B}[\![\text{[string] wchar\_t*}]\!] &&= \text{Ptr}\\
&\mathbf{B}[\![\text{BSTR}]\!] &&= \text{Ptr}\\
\\
&\mathbf{T}[\![\text{[string] char*}]\!] &&= \text{String}\\
&\mathbf{T}[\![\text{[string] wchar\_t*}]\!] &&= \text{String}\\
&\mathbf{T}[\![\text{BSTR}]\!] &&= \text{String}
\end{aligned}
$$

The length of a string is known at run-time by looking for the zero, terminating the string: $\mathbf{S}[\![\text{[string] t*}]\!]\ x = (\text{strlen}\ x) * \mathbf{S}[\![\text{t}]\!]$. For ascii and unicode strings, Redcard uses the normal array translation (but with the correct length information). A `BSTR` uses special allocation functions provided by COM. The **M**, **U** and **F** functions for a `BSTR` object are provided in the module OleMarshall as primitives.

For a `BSTR` and unicode string, the unicode characters need to be mapped to ansi characters and back. This is done by applying: $(\text{map}_{io}\ \mathit{toansi}) :: \text{IO [Short]} \rightarrow \text{IO String}$ to the **R** and **U** functions while applying: $\mathit{touni} :: \text{String } \rightarrow \text{[Short]}$ to the on the **W** and **M** functions.

4.5 Mapping of declarations

All the basic IDL types can now be translated to Haskell types together with their marshalling functions. The following sections define the actual map function $\mapsto$, which is defined in terms of the other translation functions. We end with the method declaration, which uses all the previously defined functionality to perform a call to a COM component. Important information for a user of a component is boxed, other information gives the fine details of the translation.

4.5.1 Libraries

$$\boxed{\textbf{library}\ x\ \textbf{\{}\quad \{\ \mathit{definition}\ d_i\ ;\ \}^n\quad \textbf{\}}}$$

$$\mapsto$$

$$\boxed{\textbf{module}\ x\ \textbf{(}\ d_1\ ,\ \ldots\ ,\ d_n\ \textbf{)}\ \textbf{where}}$$

import OleMarshall

Redcard will generate a module containing the definitions inside the library. The module name is the same as the library except when the user specifies a filename in which case the module name is the same as that of the file. The library will export all public definitions from the IDL file. $\mapsto$ is called for each IDL declaration in the library to generate the corresponding Haskell definition.

Another design decision could be to generate one module for each declaration and just generate the main library module to export all these declaration modules. This would solve the name clashing problems since we could now control the name spaces with the module system. However, there will be mutually dependent modules since different declarations in an IDL file can refer to each other. Since both GHC and Hugs don't like mutually dependent modules, we chose not to do this and use the renaming mechanism instead.

4.5.2 Type definitions

$$\boxed{\textbf{typedef}\ t\ x\ ;}$$

$$\mapsto$$

$$\boxed{\textbf{type}\ x = \mathbf{T}[\![\mathrm{t}]\!]}$$

Type definitions are directly translated to a type synonym in Haskell. We define:

$$\begin{aligned}
\mathbf{B}[\![\mathrm{x}]\!] &= \mathbf{B}[\![\mathrm{t}]\!]\\
\mathbf{T}[\![\mathrm{x}]\!] &= x\\
\mathbf{S}[\![\mathrm{x}]\!] &= \mathbf{S}[\![\mathrm{t}]\!]\\
\mathbf{F}[\![\mathrm{x}]\!] &= \mathbf{F}[\![\mathrm{t}]\!]\\
\mathbf{R}[\![\mathrm{x}]\!] &= \mathbf{R}[\![\mathrm{t}]\!]\\
\mathbf{W}[\![\mathrm{x}]\!] &= \mathbf{W}[\![\mathrm{t}]\!]\\
\mathbf{M}[\![\mathrm{x}]\!] &= \mathbf{M}[\![\mathrm{t}]\!]\\
\mathbf{U}[\![\mathrm{x}]\!] &= \mathbf{U}[\![\mathrm{t}]\!]
\end{aligned}$$

The **T** function generates the typedef name to get readable type signatures. Other than that, a type definition has no effect on the translation process.

4.5.3 Enumerations

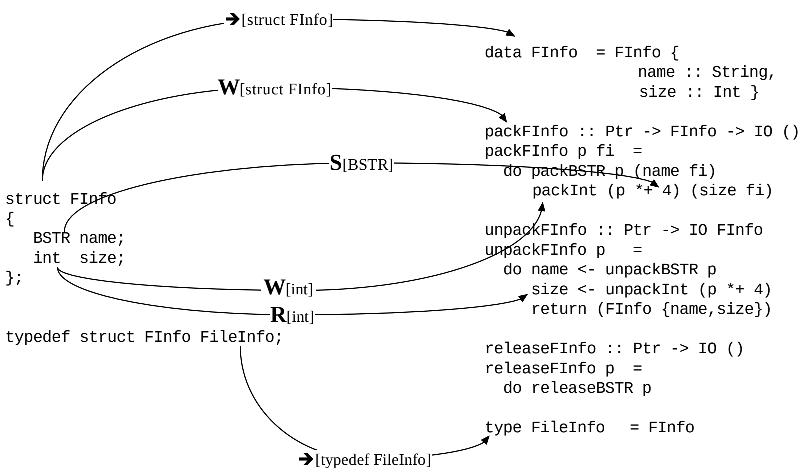
$$\boxed{\textbf{enum}\ x\ \pmb{\{}\quad \{\ e_i\ [= k_i]\ \ ,\ \ \}^n\quad \pmb{\}}}$$

$$\mapsto$$

$$\boxed{\textbf{data}\ x = e_1 \mid ... \mid e_n \quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show}\ [, \mathit{Enum}])}$$

$$\begin{aligned}
&\textbf{instance Enum}\ x\ \textbf{where}\\
&\qquad \mathrm{fromEnum}\ x = \textbf{case}\ x\ \textbf{of}\ \{\ e_i \to k_i\ \}^n\\
&\qquad \mathrm{toEnum}\ k \quad\ = \textbf{case}\ k\ \textbf{of}\\
&\qquad\qquad\qquad\qquad \{\ k_i \to e_i\ \}^n\\
&\qquad\qquad\qquad\qquad \_ \to \textbf{error}\ \text{“invalid enum value”}
\end{aligned}$$

Enumerations are translated to a algebraic data type with each of the enumeration values as a constructor. The generic `toEnum` and `fromEnum` functions can be used to translate the constructors to COM integer values. The translation for enumerations is defined using the translation functions for the `int` type:



```
struct FInfo
{
   BSTR name;
   int  size;
};

typedef struct FInfo FileInfo;
```

```
data FInfo  = FInfo {
                 name :: String,
                 size :: Int }

packFInfo :: Ptr -> FInfo -> IO ()
packFInfo p fi  =
  do packBSTR p (name fi)
     packInt (p *+ 4) (size fi)

unpackFInfo :: Ptr -> IO FInfo
unpackFInfo p   =
  do name <- unpackBSTR p
     size <- unpackInt (p *+ 4)
     return (FInfo {name,size})

releaseFInfo :: Ptr -> IO ()
releaseFInfo p  =
  do releaseBSTR p

type FileInfo   = FInfo
```

Figure 4.3: *Translation functions used for a structure declaration*

$$
\begin{array}{ll}
\mathbf{B}\llbracket \mathrm{x} \rrbracket & = \mathrm{Int} \\
\mathbf{T}\llbracket \mathrm{x} \rrbracket & = x \\
\mathbf{S}\llbracket \mathrm{x} \rrbracket & = \mathbf{S}\llbracket \mathrm{Int} \rrbracket \\
\mathbf{R}\llbracket \mathrm{x} \rrbracket & = \mathrm{map}_{io}\ \mathrm{toEnum} \cdot \mathbf{R}\llbracket \mathrm{Int} \rrbracket \\
\mathbf{W}\llbracket \mathrm{x} \rrbracket p & = \mathbf{W}\llbracket \mathrm{Int} \rrbracket\ p \cdot \mathrm{fromEnum} \\
\mathbf{U}\llbracket \mathrm{x} \rrbracket & = \mathrm{map}_{io}\ \mathrm{toEnum} \cdot \mathbf{U}\llbracket \mathrm{Int} \rrbracket \\
\mathbf{M}\llbracket \mathrm{x} \rrbracket & = \mathbf{M}\llbracket \mathrm{Int} \rrbracket \cdot \mathrm{fromEnum}
\end{array}
$$

Redcard supports the Microsoft IDL extension where enumeration values can have a specific integer value in which case we generate a specific `Enum` instance. For example:

```
> enum Foo { bar, baz = 2 };
```

translates to:

```
> data Foo = Bar | Baz    deriving (Eq,Ord,Show)
>
> instance Enum Foo where
>    toEnum i   = case i of
>                   0  -> Bar
>                   2  -> Baz
>                   _  -> error ("Foo.toEnum: unknown enum value"
>                                  ++ show i)
>    fromEnum x = case x of
>                   Bar -> 0
>                   Baz -> 2
```

4.5.4 Structures

$$\boxed{\textbf{struct}\ x\ \textbf{\{}\quad \{\ type_i\ f_i\ \textbf{;}\ \}^n \quad \textbf{\}}}$$

$$\mapsto$$

$$\boxed{\textbf{data}\ x = x\ \textbf{\{}\quad f_1 :: \mathbf{T}\llbracket type_1 \rrbracket \mid \ldots \mid f_n :: \mathbf{T}\llbracket type_n \rrbracket \quad \textbf{\}}}$$

Structures map to a Haskell record with a constructor of the same name as its type.

$$\begin{array}{ll} \mathbf{T}\llbracket \mathrm{x} \rrbracket & = x \\ \mathbf{S}\llbracket \mathrm{x} \rrbracket & = \sum_{i=0}^{n} \mathbf{S}\llbracket \mathit{type}_i \rrbracket \end{array}$$

The $\mathbf{F}\llbracket \mathrm{x} \rrbracket$ function applies $\mathbf{F}\llbracket \mathit{type}_i \rrbracket$ to each field. The **R** and **W** functions do the same.

$$\begin{array}{lll} \mathbf{F}\llbracket \mathrm{x} \rrbracket\ p & = \mathbf{do}\ \{ & \{\ \mathbf{F}\llbracket \mathit{type}_i \rrbracket\ \ (p \star\!+ \mathrm{ofs}(\ f_i\ ))\ ;\ \}^n \quad \} \\ \mathbf{R}\llbracket \mathrm{x} \rrbracket\ p & = \mathbf{do}\ \{ & \{\ f_i\ \leftarrow \mathbf{R}\llbracket \mathit{type}_i \rrbracket\ \ (p \star\!+ \mathrm{ofs}(\ f_i\ ))\ ;\ \}^n \\ & & \mathbf{return}\ (x\ \{f_1, ..., f_n\}) \quad \} \\ \mathbf{W}\llbracket \mathrm{x} \rrbracket\ p\ s & = \mathbf{do}\ \{ & \{\ \mathbf{W}\llbracket \mathit{type}_i \rrbracket\ \ (p \star\!+ \mathrm{ofs}(\ f_i\ ))\ (f_i\ s)\ ;\ \}^n \\ & & \mathbf{return}\ p \quad \} \end{array}$$

As with arrays, structures are never passed by value so the **U**, **M** and **B** functions don't need to be defined.

(Actually, Redcard does handle structures passed by value by simulating that it is passed by reference and push it later by value on the C-stack.)

Figure 4.3 shows for an example structure how some of the translation functions are used and how the **W**, **R** and **F** functions are generated.

4.5.5 Unions

$$\boxed{\mathbf{union}\ x\ \{\quad \{\ \mathit{type}_i\ f_i\ ;\ \}^n \quad \}}$$

$$\mapsto$$

$$\boxed{\mathbf{data}\ x = f_1\ \mathbf{T}\llbracket \mathit{type}_1 \rrbracket \mid ... \mid f_n\ \mathbf{T}\llbracket \mathit{type}_n \rrbracket}$$

A union is translated to an algebraic data type with the field names as constructors. Just as with structures we define:

$$\begin{array}{ll} \mathbf{T}\llbracket \mathrm{x} \rrbracket & = x \\ \mathbf{S}\llbracket \mathrm{x} \rrbracket & = \max(\mathbf{S}\llbracket \mathit{type}_i \rrbracket) \end{array}$$

Since a union is never passed by value, the **U**,**M** and **B** functions are unnecessary for unions.

The **W** function does a case on the data type and applies the right $\mathbf{W}\llbracket \mathit{type}_i \rrbracket$ function:

$$\mathbf{W}\llbracket \mathrm{x} \rrbracket\ p\ u\ =\ \mathbf{case}\ u\ \mathbf{of}\ \{\ (f_i\ x)\ \rightarrow\ \mathbf{W}\llbracket \mathit{type}_i \rrbracket\ \ (p \star\!+ \mathrm{ofs}(\ f_i\ ))\ x\ \}^n$$

The **R** or **F** function can not do case analysis but needs to know the valid field at run-time to read or free. IDL provides attributes that allow the valid field to be chosen at run-time but this information is lost in the type library. The generated code will therefore take an integer i that functions as the selector. **R** and **F** now call $\mathbf{R}\llbracket \mathit{type}_i \rrbracket$ and $\mathbf{F}\llbracket \mathit{type}_i \rrbracket$ respectively:

$$\begin{array}{lll} \mathbf{F}\llbracket \mathrm{x} \rrbracket\ p\ i & = & \mathbf{F}\llbracket \mathit{type}_i \rrbracket\ \ (p \star\!+ \mathrm{ofs}(\ f_i\ )) \\ \mathbf{R}\llbracket \mathrm{x} \rrbracket\ p\ i & = & \mathbf{do}\ \{\ \ x\ \leftarrow \mathbf{R}\llbracket \mathit{type}_i \rrbracket\ \ (p \star\!+ \mathrm{ofs}(\ f_i\ ))\ ; \\ & & \qquad \mathbf{return}\ (f_i\ x) \quad \} \end{array}$$

The generated code needs to be handpatched to provide the valid selector at run-time (which will default to an invalid value). Redcard issues a warning when this situation occurs. For example, suppose there is a structure like:

```
> struct IntFloat {
```

```
>  boolean isInt;
>  union tagValue {
>    int i;
>    float f;
>  }       value;
> };
```

The relevant generated code will be like:

```
> data IntFloat      = IntFloat { isInt :: Int, value :: TagValue }
> data TagValue      = I Int | F Float

> unpackIntFloat p   = do isInt <- unpackBool p
>                          value <- unpackTagValue (p *+ 1) 0
>                          IntFloat { isInt, value }

> unpackTagValue p i =
>       case i of 1 -> do { i <- unpackInt p; return (I i) }
>                 2 -> do { f <- unpackFloat p; return (F f) }
>                 _ -> comFail "TagValue: unpacking invalid union field"
```

The selector passed as `0` in `unpackTagValue` needs to be adjusted by hand to:

```
> unpackIntFloat p   =
>       do isInt <- unpackBool p
>          value <- unpackTagValue (p *+ 1) (if isInt then 1 else 2)
>          IntFloat { isInt, value }
```

4.5.6 Com classes

$[\textbf{uuid}(guid_x)]$

$\boxed{\textbf{coclass}\ \mathrm{x}\ \{\ \ \ldots\ \ \}}$

$$\mapsto$$

$\boxed{\mathrm{clsid}x :: \mathrm{CLSID}}$

$\mathrm{clsid}x = \mathrm{mkCLSID}\ (\mathrm{newGuid}\ guid_x)$

Each COM class is translated to a function of type `CLSID`. Since a class is not a first-class value, there is no need to define any translation function. The only data needed at run-time is the `CLSID` to create an instance of a specific class.

4.5.7 Interfaces

$[\textbf{uuid}(guid_x)]$

$\boxed{\textbf{interface}\ \mathrm{x}\ \{\ \ \ldots\ \ \}}$

$$\mapsto$$

$\boxed{\textbf{data}\ x = x}$

$\boxed{\mathrm{iid}x :: \mathrm{Interface}\ x}$

$\mathrm{iid}x = \textbf{Interface}\ (\mathrm{newGuid}\ guid_x)$

Interfaces are translated into two items. The first is an abstract data type x that is the equivalent of the interface type. This type is used both to identify its `IID` of type `Interface x` and to identify a run-time pointer to this interface of type `Com x`. The second item is a function with name iid$x$ which returns a value of type `Interface x`, the Haskell equivalent of an `IID`. See the previous chapter for an explanation of these types.

We define:

$$\begin{aligned}
&\mathbf{B}[\![\text{x*}]\!] && = \text{Ptr}\\
&\mathbf{T}[\![\text{x*}]\!] && = \text{Com x}\\
&\mathbf{S}[\![\text{x*}]\!] && = \mathbf{S}[\![\text{void*}]\!]
\end{aligned}$$

Since an interface is represented at run-time by a simple pointer, it can be treated as the `void*` type. However we need to maintain the correct reference count. The function `primRelease :: Ptr -> IO ()` calls the `Release` method of a component, `addRef :: Com a -> IO Int` increments the reference count and `finalize :: Ptr -> Com a` instantiates a `Com` type with a foreign pointer. The finalization routine of this foreign pointer will call `primRelease` if garbage collected. We can now define:

$$\begin{aligned}
&\mathbf{F}[\![\text{x*}]\!]\,p && = \mathbf{do}\ \{\ p' \leftarrow \mathbf{R}[\![\text{void*}]\!]\,p;\ \text{primRelease}\,p'\ \}\\
&\mathbf{R}[\![\text{x*}]\!]\,p && = \mathbf{do}\ \{\ p' \leftarrow \mathbf{R}[\![\text{void*}]\!]\,p;\ x \leftarrow \text{finalize}\,p';\ x \mathbin{\#} \text{addRef}\ \}\\
&\mathbf{W}[\![\text{x*}]\!]\,p\,x && = \mathbf{do}\ \{\ x \mathbin{\#} \text{addRef}\ ;\ \mathbf{W}[\![\text{void*}]\!]\,p\,(\text{comToPtr}\ x)\ \}\\
&\mathbf{U}[\![\text{x*}]\!]\,p && = \mathbf{do}\ \{\ p' \leftarrow \mathbf{U}[\![\text{void*}]\!]\,p;\ \text{finalize}\,p'\ \}\\
&\mathbf{M}[\![\text{x*}]\!]\,x && = \mathbf{do}\ \{\ x \mathbin{\#} \text{addRef}\ ;\ \mathbf{M}[\![\text{void*}]\!]\,(\text{comToPtr}\ x)\ \}
\end{aligned}$$

The COM standard states that the reference count should only be incremented whenever a copy is made of an interface pointer. The reference count has to be decremented if the pointer is not live anymore [Bro95, page 83-90]. The above definitions satisfy these constraints. It is a bit tricky to see, since there can be an instance of the pointer both in COM world and in Haskell world. The next section will show that the reference count of an argument will stay correct during a method call using these definitions.

4.5.8 Member functions

Last but not least, the member functions or methods. It is here where the actual marshalling takes place and where all the machinery developed in the previous sections is necessary.

$$\boxed{\begin{array}{ll}
\mathbf{interface}\ x\ \{ & \{\ \mathit{type}_m\ \mathit{method}_m\ (\{[\mathbf{in}]\ \mathit{itype}_i\ \mathit{in}_i\}^I,\\
& \qquad\qquad\qquad \{[\mathbf{inout}]\ \mathit{iotype}_j\ \mathit{io}_j\}^J,\\
& \qquad\qquad\qquad \{[\mathbf{out}]\ \mathit{otype}_k\ \mathit{out}_k\}^K\ )\ \}^M \quad \mathbf{\}}
\end{array}}$$

$\mapsto$

$$\boxed{\begin{array}{l}
\mathit{method}_m :: \mathbf{T}[\![\mathit{itype}_1]\!] \rightarrow \ldots \rightarrow \mathbf{T}[\![\mathit{itype}_I]\!]\\
\qquad \rightarrow \mathbf{OT}[\![\mathit{iotype}_1]\!] \rightarrow \ldots \rightarrow \mathbf{OT}[\![\mathit{iotype}_J]\!]\\
\qquad \rightarrow \text{Com}\ x \rightarrow \text{IO}\ ([\mathbf{T}[\![\mathit{type}_m]\!]]\ \{,\mathbf{OT}[\![\mathit{iotype}_j]\!]\}^J\ \{,\mathbf{OT}[\![\mathit{otype}_k]\!]\}^K)
\end{array}}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{method}_m\ \mathit{in}_1 \ldots \mathit{in}_I\ \mathit{io}_1 \ldots \mathit{io}_J\ \ \text{self}}\\
= \mathbf{do}\ \{ & \{\ \mathit{in}'_i \leftarrow \mathbf{M}[\![\mathit{itype}_i]\!]\ \mathit{in}_i\ ;\ \ \}^I\\
& \{\ \mathit{io}'_j \leftarrow \mathbf{OM}[\![\mathit{iotype}_j]\!]\ \mathit{io}_j\ ;\ \ \}^J\\
& \{\ \mathit{out}'_k \leftarrow \text{alloc}\ \mathbf{S}[\![\mathit{otype}_k]\!]\ ;\ \ \}^K\\
& \text{self}' \leftarrow \mathbf{M}[\![\text{void*}]\!]\ \text{self}\ ;\\
& \text{res} \leftarrow \mathbf{prim}\mathit{method}_m\,\text{self}'\ ,\mathit{in}'_1\,, \ldots\,, \mathit{in}'_i\,, \mathit{io}'_1\,, \ldots\,, \mathit{io}'_j\,, \mathit{out}'_1\,, \ldots\,, \mathit{out}'_k\\
& [\text{check res}\ ;]\\
& [\text{res} \leftarrow \mathbf{U}[\![\mathit{type}_m]\!]\,\text{res}'\ ;]\\
& \{\ \mathit{io}_j \leftarrow \mathbf{OU}[\![\mathit{iotype}_j]\!]\ \mathit{io}'_j\ ;\ \ \}^J\\
& \{\ \mathit{out}_k \leftarrow \mathbf{OU}[\![\mathit{otype}_k]\!]\ \mathit{out}'_k\ ;\ \ \}^K\\
& \{\ \mathbf{F}[\![\mathit{itype}_i]\!]\ \mathit{in}'_i\ ;\ \ \}^I
\end{array}$$

$$\textbf{return}\ ([\text{res}\ ,]io_1\ , \ldots\ , io_j\ , out_1\ , \ldots\ , out_k) \quad \}$$

%fun prim$method_m$

The above definition deals with a method $method_m$ with result type $type_m$, I input arguments in_i, J input- and output arguments io_j and K output arguments out_k. The definition given here is actually somewhat more restricted than Redcard handles. Redcard can also translate methods with arguments that are mixed `[in]` and `[out]`, but this makes the definition hopelessly more obscure. The algorithm to find the argument and result in Haskell, is to scan the method first from left to right for `in` arguments. These are the arguments for the method in Haskell. The result in Haskell is found by scanning the signature from left to right for `out` arguments including a result value.

In IDL, output arguments are always passed by reference by declaring a pointer type. In Redcard, this is simulated by translating as if it is a pointer to one-element array. These output versions of the normal translation functions are:

$$\begin{array}{ll}
\mathbf{OT}[\![\text{t*}]\!] & = \mathbf{T}[\![\text{t}]\!] \\
\mathbf{OM}[\![\text{t*}]\!]\ x & = \mathbf{M}[\![\text{t*}]\!]\ [x] \\
\mathbf{OU}[\![\text{t*}]\!]\ p & = \textbf{do}\ \{\ [x]\ \leftarrow \mathbf{U}[\![\text{t*}]\!]\ p\ ;\ \textbf{return}\ x\ \}
\end{array}$$

Input arguments are first marshalled using **M**. After that, `[in,out]` arguments are marshalled using **OM** which always returns a pointer. For `[out]` arguments, a piece of memory is allocated which receives the result value. The self argument is marshalled by value without using the normal translation functions for interfaces. This optimisation is possible here because the pointer value is just extracted to make a call into the virtual method table and a correct reference count does not have to be maintained. At this point, all the arguments are translated to their COM counterparts and the actual call is performed by some standard green-card code.

The **check** statement is inserted whenever the method returns a `HRESULT`. Redcard automatically tests for this value and it will not show up in the result.

After the call, the output arguments are unmarshalled using **OU** which also deallocates any allocated memory. The finalization function is called for each input argument, and the final result is returned.

At the moment, the COM heap is used to do memory allocation, but all `[in]` parameters, together with `[in,out]` parameters that are not reallocated, could be allocated in a stack-wise fashion. This stack would allow very fast and cheap allocation of scalar values like `Int` and could be used immediately as the call-stack for the COM call when other pointer values would be written to this stack too. A large number of arguments could probably be allocated this way. We hope to investigate this when developing the successor of Redcard; haskellDirect [Sig98]. haskellDirect will no longer use greencard but the compiler will directly implement the primitive method call without using the C compiler.

Here are some examples of the type translation done by Redcard:

```
> interface IX {
>   void     f1( [in] float f, [in] int* i, [out] short* s );
>   int      f2( [out] long* l, [in,string] char* s,
>                [in,out] double* d );
>   HRESULT  f3( [in,string] wchar_t* ws, [in,out] short** s,
>                [out] long* l );
> }
```

This will be translated to:

```
> data IX
>
> iidIX :: Interface IX
>
> f1 :: Float -> [Int] -> Com IX -> IO Short
> f2 :: String -> Double -> Com IX -> IO (Int,Long,Double)
> f3 :: String -> [Short] -> Com IX -> IO ([Short],Long)
```

Note how Redcard automatically tests the `HRESULT` and passes output parameters by reference.

4.5.9 Reference counting

The behaviour of the marshalling functions with respect to reference counting can be shown correct. The following table shows how the reference counting is done for an interface x:

$\mathbf{M}[\![\mathrm{x}]\!]$	+
$\mathbf{U}[\![\mathrm{x}]\!]$	∼
$\mathbf{W}[\![\mathrm{x}]\!]$	+
$\mathbf{R}[\![\mathrm{x}]\!]$	+,∼
$\mathbf{F}[\![\mathrm{x}]\!]$	–

A ‘+’ means a call to `addRef` and ‘–’ a call to `release` and the ’∼’ is a call to `finalize`. A ‘∼’ is used when release is called when the object is not live anymore (garbage collected).

Since the reference count should only be incremented when making a copy, only `out` parameters should have there reference count incremented. In all other cases the netto effect should be nil [Bro95, page 83-90].

For an interface pointer x as argument this means:

Argument type	Functions called	Reference effect
[in]	$\mathbf{M}[\![\mathrm{x}]\!]$,$\mathbf{F}[\![\mathrm{x}]\!]$	+, –
[in,out]	$\mathbf{M}[\![\mathrm{x}]\!]$,$\mathbf{U}[\![\mathrm{x}]\!]$	+,∼
[out]	$\mathbf{U}[\![\mathrm{x}]\!]$	∼

When the interface pointer is part of a structure of array, we get the following behaviour during a method call:

Argument type	Functions called	Reference effect
[in]	$\mathbf{W}[\![\mathrm{x}]\!]$,$\mathbf{F}[\![\mathrm{x}]\!]$	+, –
[in,out]	$\mathbf{W}[\![\mathrm{x}]\!]$,$\mathbf{R}[\![\mathrm{x}]\!]$,$\mathbf{F}[\![\mathrm{x}]\!]$	+,+,∼,–
[out]	$\mathbf{R}[\![\mathrm{x}]\!]$,$\mathbf{F}[\![\mathrm{x}]\!]$	+,–,∼

In each of the cases, the behaviour is correct with respect to the COM standard.

Chapter 5

Agents

This chapter will give a more detailed example of using COM components from Haskell. This will be done using the Microsoft Agent component. The Microsoft Agent component creates cartoon characters on the screen (see figure 5.1) which can talk, move around and react on speech input. Microsoft Agent is freely available and there is an excellent book about programming the agents [Mic97b]. Already, many companies are using the agent to guide users through web sites or as a personal assistant in an application.

Besides giving a detailed example, I also want to show how laziness and higher-order functions give Haskell an advantage over conventional languages used to program COM components [Sim98]. Haskell's powerful abstraction mechanisms make it a great scripting language. The ability to define 'domain specific combinators' can effectively yield a mini language tailored for the needs of a specific application [Wad98, EH97, Pau96]. In the case of Microsoft Agent, combinators are defined to synchronize parallel running agents.

5.1 Running 'hello world'

The agent component can be downloaded from: `http://www.microsoft.com/oledev/agent`. Using the `oleview` tool, the installed agent component can be inspected and we see that the agent type library is enclosed in `agentsvr.exe`:

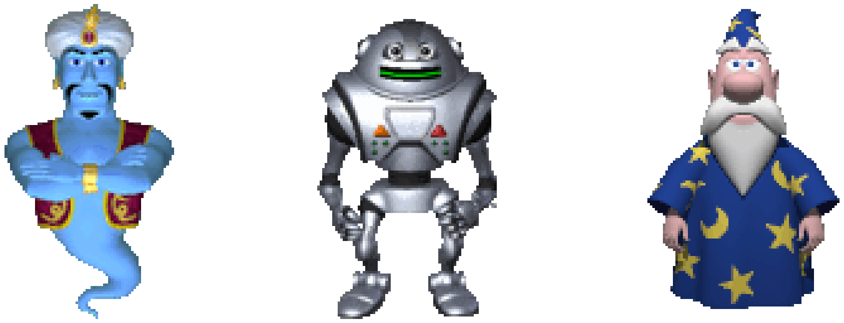


Figure 5.1: *The agents: Genie, Robby and Merlin*

```
> redcard -oldgc agentsvr.exe agent
```

Redcard generates the file `agent.ss` which is compiled using the old greencard. The resulting `agent.dll` and the `agent.hs` module let us use the component from Haskell.

The Agent component is actually a server which coordinates all character components. To create a character, a character animation file is first loaded into the server. The server can than be asked for a character interface pointer using this animation file. The next listing gives a super fancy 'hello world' program in 12 lines[1]:

```
> module Main where
>
> import Com            -- basic support
> import Agent          -- our generated module
>
> main
>   = comRun $
>     do server     <- comCreateObject "Agent.Server" iidIAgent
>        (charID,_) <- server # load (setVariant VT_BSTR
>                                              (BstrVal path))
>        genie      <- server # getCharacter charID
>        genie      <- genie  # query  iAgentCharacter
>
>        genie # showUp 0
>        genie # speak "Hello, COM World" ""
>        putStr "press enter.."
>        getChar
>
>   where
>      path = "c:\\Program Files\\Microsoft Agent\\Characters\\genie.acs"
```

The server is created with the `comCreateObject` call, of course `comCreateInstance` could also be used:

```
> server <- comCreateInstance clsidAgentServer Nothing LocalProcess iidIAgent
```

The next call uses the `setVariant` function to pass the path to the animation file. `setVariant` is exported from `OleTypes` and creates a `VARIANT` structure. This structure is part of `Automation` and will be described in the next chapter. After loading the animation, a character component is returned from the server and queried for its `IAgentCharacter` interface.

This example is not much different than the same program in C++, Java or VB. The following sections try to show how Haskell can give an advantage to more conventional languages when programming components, even though Haskell lacks an object oriented language model.

5.2 Abstraction

The main advantage of Haskell, is its powerful abstraction mechanisms. Haskell makes it easy to abstract from commonly occurring patterns in our code. Suppose we want our animation to gesture left and right in a row:

[1] Needing a 120 Mhz Pentium to run.

```
> type Action      = Com IAgentCharacter -> IO ReqID
>
> gestureLR        :: Action
> gestureLR agent  = do agent # play "GestureLeft"
>                       agent # play "GestureRight"
```

`genie # gestureLR` can now be used to gesture left and right. In C++, a function `gestureLR` can be defined but a different syntax is needed to call it:

```
> genie->speak( "hi", "");
> gestureLR( genie );
```

In Java (and also C++), normally a new class that inherits from `IAgentCharacter` is created which contains the `gestureLR` method. However a type distinction between agents that can gesture left and gesture right and agents that can gesture left and right is created. There is also no way of knowing where to stop this chain of classes. If a library is created, the user will again create a new class to add extra methods on top of the class in the library. It seems as though inheritance is a good fit for (implementing) building basic components but not for extending or combining abstract interfaces [Wil90]. In Haskell, a component gives us the primitives on which we can build our own abstractions.

The pattern `agent # play` occurs twice in the `gestureLR` method. The `withObject` function in combination with `map` can be used to write:

```
> gestureLR agent  = withObject agent (map play ["GestureLeft","GestureRight"])
```

This commonly occurring pattern is very usefull and is defined in the `Com` module:

```
> withMethod :: (arg -> obj -> IO b) -> [arg] -> obj -> IO ()
> withMethod fun args obj       = withObject obj (map fun args)
```

and `gestureLR` can now be written as:

```
> gestureLR       = withMethod play ["GestureLeft","GestureRight"]
```

A definition like this is very hard to write in conventional languages since it relies on higher-order functions (`map`) and on the fact that the method call operator (`#`) and even methods (`play`) are first class objects. In this particular example this doesn't seem to be a great advantage but even such a small definition like `withMethod` can be reused in many situations.

The next definition moves a character via a specific set of points on the screen.

```
> type Pos      = (Int,Int)
>
> moveAlong      :: [Pos] -> Action
> moveAlong      = withMethod moveToPos
```

By using our own control operator `withMethod`, this is a trivial exercise. In conventional languages we are dependent on the language designers to define every possible control structure. In VB for example, our newly found pattern needs to be repeated over and over again:

```
> Sub MoveAlong (Byref Agent, Byref Path)
>  For Each Point in Path
>   Agent.MoveTo (Point)
>  Next Point
> End Sub
```

moveAlong can now be reused to move our character along more specific figures, such as circles or squares:

```
> circle :: Pos -> Int -> [Pos]
> circle (x,y) radius
>       = [ (x + (radius * cos t),
>            y + (radius * sin t))  | t <- [0,pi/100 .. pi]]
>
> square :: Pos -> Int -> [Pos]
> square (x,y) radius
>       = [(x-radius,y-radius), (x+radius,y-radius),
>          (x+radius,y+radius), (x-radius,y+radius), (x-radius,y-radius)]
>
> demo        :: Action
> demo  = withMethod moveAlong [circle org radius, square org radius]
>       where
>         org     = (100,100)
>         radius  = 50
>
> main  = do ...
>             genie # demo
```

Functional languages have always been a great 'glue' language ([Hug89]) but I think they lacked the bricks to glue together. Now that components can be used, the bricks are found and the glue can be used: laziness and higher-order functions.

5.3 Agent combinators

A larger 'real world' problem will now be addressed. While looking at many web sites, I found that most agent demos were fairly small and simple. Especially when more agents were into view, most demos were setup sequentially with little parallel interaction. When looking more closely to the agent interfaces, the reason becomes apperent. The only way of synchronising two agents, is a very primitive method called wait. Parallel interactions between agents become very complicated with this mechanism. Suppose agent genie and merlin do something in parallel (|) followed by (·) agent robby and merlin doing something in parallel. Our intention can be written as:

$$(\text{genie } A \mid \text{merlin } B) \cdot (\text{robby } C \mid \text{merlin } D)$$

How can this be programmed in a language like Java ? Doing something in parallel is very easy. Since the agents run in different processes it is enough to call a method on an agent. The agent will put the request in its message queue and immediately return a request-id. To synchronize things sequentially, an agent needs to wait for another agent to complete its request by calling its wait method with the appropiate request-id. The above specification would be programmed in Java or VB as:

```
> reqid1 = genie.A;               //let genie and merlin perform
> reqid2 = merlin.B;
>
> robby.wait(reqid1);             //let robby wait
> robby.wait(reqid2);
> merlin.wait(reqid1);            //let merlin wait
> reqid3 = merlin.wait(reqid2);
```

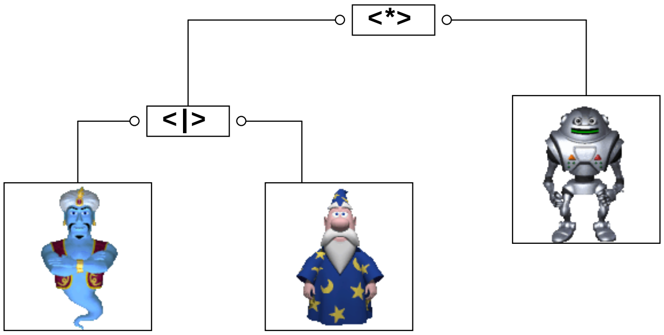


Figure 5.2: *Building animations using combinators*

```
>
> reqid4 = robby.wait(reqid3);     //synchronize robby and merlin
> merlin.wait(reqid4);
>
> robby.C;                         //let robby and merlin perform
> merlin.D;
```

The simple specification has become quite complicated. It is difficult to program these kinds of animations. All the structure of the specification is gone and it is not at all clear what exactly happens. More than half of the statements are devoted to synchronisation details. Secondly, the program contains a subtle bug. Merlin waits for itself, causing a deadlock. It is difficult to spot this class of errors. A library is written for Visual Basic which solves the problem with by maintaining an explicit synchronisation queue, but this forces the programmer to insert an explicit synchronisation request at every method call ! Allthough safer, the resulting program doesn't look much nicer.

In Haskell it is possible to define a parallel and sequential combinator which hide all the synchronisation complexity. The above specification can be written in Haskell as:

```
> (genie A <|> merlin B) <*> (robby C <|> merlin D)
```

In the next paragraphs, the definition of these combinators will be derived. The combinators do not work directly on agent components but on so-called animations, consisting of some arbitrary interaction between agents. Figure 5.2 shows how agents can be glued together using the combinators. The resulting building blocks can be combined again to create more complex animations.

To perform two animations in sequence, the second animation needs to wait for first animation to complete. Since an animation returns the request-id of the last action performed, the second animatioin can wait for that request-id to complete. To let everybody part of the second animation wait for this request-id, the *cast* of the animation needs to be maintained. An animation is therefore represented as a tuple containing the program that performs the animation (an `IO` type) and its cast (a list of agents):

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

If two animations are combined, its new cast consists of the union of the casts of the two animations. A first try is:

```
> (<*>) :: Anim -> Anim -> Anim
> (anim1,cast1) <*> (anim2,cast2)
>       = (anim, cast1 `union` cast2)
>       where
>         anim  = do reqid1 <- anim1
>                    cast2 `waitfor` reqid1
>                    anim2
```

The operation waitfor lets everybody wait for a request-id:

```
> waitfor agents reqid  = do sequence (map (# wait reqid) agents)
>                            return reqid
```

This definition of <*> ignores deadlock. The difference (\\) between cast2 and cast1 should wait for the first animation to complete. Secondly, anyone part of cast1 but not cast2 should wait for the second animation to complete in order to maintain the invariant that an animation returns the request-id that flags the completion of an animation (which implies that nobody part of the animation can begin sooner with another action).

The final definition is now:

```
> (<*>) :: Anim -> Anim -> Anim
> (anim1,cast1) <*> (anim2,cast2)
>       = (anim, cast1 `union` cast2)
>       where
>         anim  = do reqid1 <- anim1
>                    (cast2 \\ cast1) `waitfor` reqid1
>                    reqid2 <- anim2
>                    (cast1 \\ cast2) `waitfor` reqid2
```

It is now fairly easy to define the parallel combinator:

```
> (<|>) :: Anim -> Anim -> Anim
> (anim1,cast1) <|> (anim2,cast2)
>       = (anim, cast1 `union` cast2)
>       where
>         anim  = do reqid1 <- anim1
>                    reqid2 <- anim2
>                    (cast2 \\ cast1) `waitfor` reqid1
>                    (cast1 \\ cast2) `waitfor` reqid2
```

In about 20 lines of code, a very clear definition of two non-trivial combinators is written. Note that the definitions rely heavily on being able to pass computations (IO ReqID) and laziness (we don't want the whole computation in memory at one time). Since we can do equational reasoning in a functional language we can prove laws like associativity for <*>:

$$x <\!*\!> (y <\!*\!> z) = (x <\!*\!> y) <\!*\!> z$$

and associativity and reflection for <|>:

$$x <|> (y <|> z) = (x <|> y) <|> z$$
$$x <|> y = y <|> x$$

Stating laws about the combinators increases the reusability immensely. They make the intuitive notion of how these combinators should behave, precise. I hope to investigate these possibilities in a larger frameworks too, like the OleDB framework which consists of components to program databases. It would also be worthwhile to study the connection between software patterns [Eri95] and higher-order functions in Haskell.

5.4 AgentScript

Building on the above combinators a library is built with some utility functions. For example `seqAnim` which combines a list of animations:

```
> seqAnim :: [Anim] -> Anim
> seqAnim       = foldr (<*>)
```

Using the associativity law for `<*>`, it could also be defined as:

```
> seqAnim       = foldl (<*>)
```

The function `animate` creates an animation, `runAnim` runs an animation and `loadCharacter` extends the agentserver to make the loading of agents more easy:

```
> animate       :: Agent -> [Action] -> Anim
> runAnim       :: Com IAgent -> Anim -> IO ()
> loadCharacter :: String -> Com IAgent -> IO (Com IAgentCharacter)
```

Using this library, complex agent interactions can easily be written:

```
> module Demo where
>
> import AgentScript
>
> agentDemo  genie merlin robby
>   = seqAnim                   -- top level description of interaction
>       [ genie introduces,
>         merlin appears <|> robby appears,
>         (merlin sayshello <*> robby sayshello) <|> genie disappears
>       ]
>   where
>     -- desription of primitive actions
>     introduces        = [ showUp,
>                           speak "Hello, my friends will show up now" ]
>     appears           = [ showUp,
>                           play "Surprised" ]
>     sayshello         = [ speak "Hi there!" ]
>     disappears        = [ hide ]
>
>
> main  = comRun $
>         do server    <- comCreateObject "Agent.Server" iidIAgent
>            robby     <- server # loadCharacter "robby"
>            merlin    <- server # loadCharacter "merlin"
>            genie     <- server # loadCharacter "genie"
>            server # runAnim (animate genie) (animate merlin)
>                                             (animate robby)
```

The resulting library enables a whole new style of programming the agents. The combinators can be viewed as a domain specific language: AgentScript. Using

AgentScript, the interaction between agents can be described separately from the actual primitive actions performed by the agents.

The Agent component can also be embedded in web pages. They support a special set of interfaces of the ActiveX framework to make this possible. The next chapter shows how Haskell can access those components using a special COM extension, called Automation.

Chapter 6

Automation

A limitation of the virtual method table interface of COM is that any client of a COM object must bind to the methods on the basis of their location in the vtable. A call as `media->Run()` will be compiled into a call instruction from a specific offset of the vtable pointed to by `media`. While this is great for compiled code, it is not as useful for an interpreted language. Interpreted languages need to be able to construct calls to a method at run-time: dynamic binding. Automation is a technology that enables dynamic binding [Mic97a]. It does so by defining a COM interface called `IDispatch`. Any component implementing this interface supports dynamic binding. This interface allows a client to query type information about all the dynamic methods and to invoke them at run-time by name (`Run`) instead of offset (`40`).

When an untyped language as Visual Basic encounters a call to `Media.Run` it can't know the offset of the `Run` method in the vtable (since it doesn't have the IDL description). However, if the `Media` object supports Automation, VB can query the type information to check if the call is valid and than actually call the `Run` method by name. This mechanism completely avoids IDL compilation and linking.

An Automation object is somewhat richer than a COM object since it can expose properties which can be read or written, optional arguments and full type information. However since Automation is accessed using a normal COM interface, every language capable of calling COM can use Automation and all of its features. This does not say that it is particulary convenient to use Automation this way. A language like VB is designed to use Automation and hides all the complexity for the programmer. If a user of VB sets a propery of an object: `Object.Title = "Hi"`, VB takes care of calling the `IDispatch` interface with the right flags and parameters. If a language has such explicit support for using Automation interfaces, it is called an Automation controller.

Haskell can be made into an Automation controller by importing the `OleAuto` module. This module contains all the functionality needed to make Automation almost as easy as in VB. Haskell's abstraction mechanisms make it possible to hide all the complexity of the `IDispatch` interface and to expose a clean interface to the programmer.

6.1 ActiveX

Automation became one of the cornerstones of the ActiveX framework. ActiveX is an infrastructure for components in interactive and dynamic environments. It consists of a set of COM interfaces that deal with persistance, graphics, event handling and much more. [Bro95, Pla96, Den97, Cha96] are excellent books about ActiveX. If a component supports ActiveX, other applications and environments can use, display and customize it. An example is Internet Explorer or Word which can run ActiveX controls within their documents. This allows for example, Microsoft Agent to be embedded in Web pages. Development environments like Delphi and VB take ActiveX to the extreme by representing everything as an ActiveX component: buttons, sliders, windows etc.

ActiveX actually enabled a market for components. Components can not exist in a vacuum. Most components are truly useable as part of other components, ActiveX defines the protocol between the components. Technologies like CORBA still lack a component infrastructure. This makes it close to impossible to ship rich interactive and graphical components like Microsoft Agent in the CORBA framework.

Besides exposing generic functionality through the normal ActiveX interfaces (drawing, persistance, etc), ActiveX components expose their specific functionality through an Automation interface. An application like Internet Explorer uses ActiveX interfaces to draw and run components in the Web page and uses the Automation interface to program the components with embedded scripts in the web page.

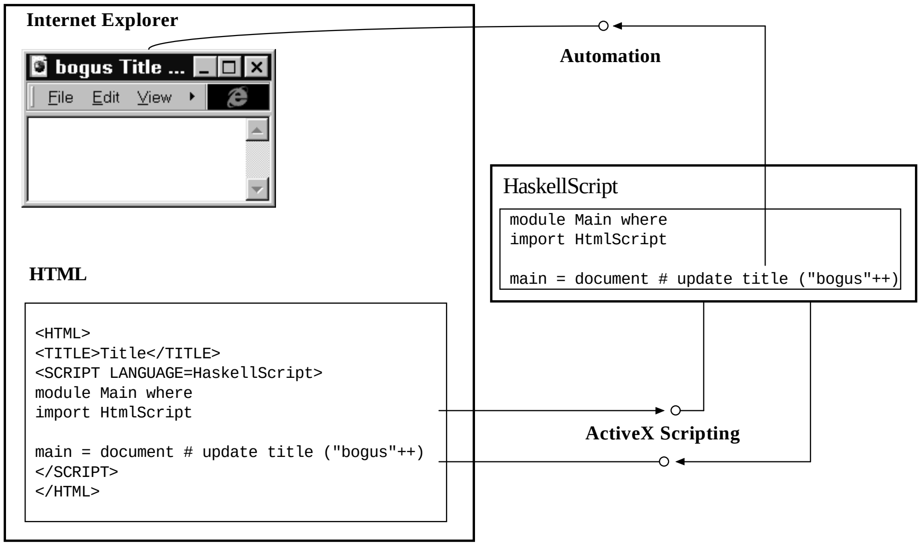
JavaScript is a standard scripting language and Automation controller. The following JavaScript program uses Automation to script the title of the Internet Explorer window:

```
> <HTML>
> <TITLE>title</TITLE>
> <BODY ONLOAD="onload()">
>
> <SCRIPT LANGUAGE="JavaScript">
>   function onload() {
>      t   = document.title;
>      yes = window.confirm( "Do you like the title " + t );
>      if (yes) document.title = "nice " + t;
>          else document.title = "bogus " + t;
>   }
> </SCRIPT>
>
> </BODY>
> </HTML>
```

window and document are two of the Automation interfaces exposed by the Internet explorer. title is a string property of the document, which is changed by this program depending on how the user likes it. Since the Automation interface is completely dynamic, no IDL is involved in running this script. All calls to the interaces are built and checked at run-time, using the IDispatch interface.

6.2 IDispatch

Every Automation object implements the IDispatch interface, that inherits directly from IUnknown. Its IDL definition is:



```
module Main where
import HtmlScript

main = document # update title ("bogus"++)
```

```
<HTML>
<TITLE>Title</TITLE>
<SCRIPT LANGUAGE=HaskellScript>
module Main where
import HtmlScript

main = document # update title ("bogus"++)
</SCRIPT>
</HTML>
```

Figure 6.1: *Using Automation and HaskellScript to program the Internet Explorer*

```
> interface IDispatch : IUnknown
> {
>   HRESULT GetTypeInfoCount( [out] unsigned int* count );
>   HRESULT GetTypeInfo( [in] unsigned int idx, [in] LCID lcid,
>                        [out] ITypeInfo** info);
>   HRESULT GetIDsOfNames( [in] IID* iid,
>                          [in,string,length_is(count)] wchar_t** names,
>                          [in] unsigned int count, [in] LCID lcid,
>                          [out,length_is(count)] DISPID* dispid );
>   HRESULT Invoke( [in] DISPID dispid, [in] IID* iid, [in] LCID lcid,
>                   [in] unsigned short flags, [in,out] DISPPARAMS* dp,
>                   [in,out] VARIANT* res, [out] EXCEPINFO* ex,
>                   [out] unsigned int* argerr );
> };
```

These four methods give you all the functionality needed to call methods by name. The first two methods give you access to the (optional) type library for the object, and the latter two allow a method to be called by name.

I have build a set of Haskell functions to hide the complexity of the automation interfaces. Automation can now be used almost as easy as in languages like VB that were specially designed to use Automation as their underlying run-time model. Haskell is now a real Automation controller ([Bro95, chapter 15]).

Since the programmer never needs to deal with the low-level `IDispatch` interface, I will not explain the interface here. A detailed explanation of this interface is given in [Bro95, chapter 14] and [Mic97a].

6.3 An example in Haskell

ActiveX provides an interface to allow any language to script an ActiveX application. By building such an interface for the Hugs interpreter, Haskell can be used to script any ActiveX application like Word and even Windows itself. This technology, called HaskellScript, is still a research item [ML98, Pla96].

The application will expose its internals using Automation interfaces. Scripts executed by HaskellScript use this Automation interface to program the application they are running in. Figure 6.1 shows how HaskellScript programs the Internet Explorer. When the Explorer encounters the `SCRIPT` tag, it will create the HaskellScript component and interact with it using the standard ActiveX scripting interfaces. The HaskellScript component will than run the script which can program the internals of IE, like the title bar, using the Automation interface.

The previous JavaScript example can be written in HaskellScript as:

```
> <HTML>
> <TITLE>title</TITLE>
>
> <SCRIPT LANGUAGE="HaskellScript">
> module Main where
> import HtmlScript
>
> main :: IO ()
> main
>   = do window   <- extern "window"
>         nm       <- window # getTitle
>        yes      <- window # confirm ("do you like the title: " ++ nm)
>        window # setTitle ((if yes then "nice " else "bogus ") ++ nm)
>
> </SCRIPT>
> </HTML>
```

The script imports the `HtmlScript` module which exports IE specific functionality and the `Automation` module which enables the use of Automation in Haskell. Properties can be get or set using the `get` and `set` methods. Calling a method on an Automation interface is done using the `#` operator as in normal COM interfaces. There is hardly any difference at a syntactic level with normal COM interfaces. Even the property get and set is orthogonal to the rest of the design. All the machinery needed to use the `IDispatch` interface is completely hidden within these operators.

All this functionality is explicitly programmed using Haskell and not built into the language like in JavaScript or VB. The Haskell Automation controller can therefore be viewed as another domain specific language within Haskell to program automation objects (see the previous chapter). Since all mechanisms are explicit and first class values, it is easy to extend them and to program our own operators.

6.4 Typing

In untyped languages like VB, it is possible to use an Automation method or property without declaring it:

```
> window.title = "nice title"
```

This little VB program will construct a call to `Invoke` which tries to set the `title` property. To do this, VB intermixes identifiers with property names (strings). In the statically typed world of Haskell this would be impossible. A property like `title` must be explicitly declared in order to be typed correctly. The `Automation` module contains functions to conveniently declare methods and properties in Haskell. The propery `title` and the method `confirm` are defined as[1]:

```
> getTitle  :: Com IDispatch -> IO String
> getTitle      = propertyGet "Title" [] outString
>
> setTitle :: String -> Com IDispatch -> IO ()
> setTitle s    = propertySet "Title" [inString s]
>
> confirm :: String -> Com IDispatch -> IO Bool
> confirm s     = function1 "confirm" [inString s] outBool
```

In order to write a definition in Haskell, the properties and methods of an Automation object need to be known. IDL can be also used to describe an Automation interface. For example:

```
> [uuid(...), oleautomation]
> dispinterface IWindow {
>   properties:
>       BSTR Title;
>   methods:
>       BOOL Confirm( [in] BSTR message );
> };
```

The `oleautomation` attribute is used to flag an Automation interface. The special keyword `dispinterface` is a Microsoft extension to ease the specification of an Automation interface. The `properties` keyword can now be used to define the properties and `methods` to define the methods of the interface. Without these extensions, the interface should be written as:

```
> [uuid(...), oleautomation]
> interface IWindow : IDispatch {
>   [id(0),propget] HRESULT Title( [out,retval] BSTR* );
>   [id(0),propput] HRESULT Title( [in] BSTR );
>
>   [id(1)] HRESULT Confirm( [in] BSTR message, [out,retval] BOOL* );
> };
```

The `id` attribute is now needed to identify the members at run-time. The `propget` and `propput` attributes control the reading and writing of properties. The `retval` attribute flags that an Automation controller should check the `HRESULT` value implicitly and return the `retval` argument as the result to the client.

When the properties and methods are known, the definitions in Haskell are simple enough to write by hand. This section will explain how to do this. The next section will give the exact translation of Automation interfaces as done by Redcard; indeed, by giving the `-auto` flag, Redcard generates these definitions automatically from an IDL description.

Basic types

Since automation calls need to be constructed efficiently at run-time, only a specific set of types is allowed as arguments to an automation interface. These basic types are called Automation types. The Automation types are summarized in figure 6.2. The `Empty` type is used as the $\perp$ type. The `Null` is the SQL-style `NULL` type.

[1] The types are given just for readability; they can be inferred automatically.

τ	$\mathbf{N}[\![\tau]\!]$	$\mathbf{A}[\![\tau]\!]$	$\mathbf{D}[\![\tau]\!]$
[Empty]	Empty	()	()
Null	Null	()	()
char	Char	Char	’\NUL’
byte	Int	Int	0
short	Int	Int	0
int	Int	Int	0
long	Int	Int	0
BOOL	Bool	Bool	False
float	Float	Float	0.0
double	Double	Double	0.0
DATE	Date	Double	0.0
BSTR	String	String	””
VARIANT	Variant	Variant	varStr ””
CURRENCY	Currency	TagInt64	TagInt64 0 0
HRESULT	HRESULT	HRESULT	0
IUnknown*	IUnknown	Com IUnknown	nullCom
IDispatch*	IDispatch	Com IDispatch	nullCom

Figure 6.2: *Haskell Name, type and default value of Automation types*

The `BSTR` type is the Automation string type. The `VARIANT` type is a structure containing any of the Automation type values with a tag identifying the type. All other types are self explanatory.

Each automation type needs a corresponding Haskell type. The **A** (automation type) function is defined to return the Haskell type of an Automation type. At run-time, functions are needed to marshall and unmarshall Automation values. The Automation module defines such functions for each Automation type. To give the marshalling functions consistent names, figure 6.2 defines the **N** (name) function that returns a Haskell name for each type. For each Automation type τ, there exist four marshalling functions with name $\mathbf{N}[\![\tau]\!]$ prefixed with: `in` for input arguments, `out` for output arguments, `inout` for input/output arguments and `res` for function results. For example, the string type has the following marshalling functions: `inString`, `outString`, `inoutString` and `resString`. The **D** (default) function in figure 6.2 returns a default value for each Automation type and is only used in the automatic translation.

Properties

The following IDL defines an Automation interface `Foo`:

```
> [uuid(...), oleautomation]
> dispinterface Foo {
>   properties:
>       BOOL Visible;
>       [readonly] int  Len;
> };
```

The corresponding properties in Haskell are defined as:

```
> getVisible :: Com IDispatch -> IO Bool
> getVisible    = propertyGet "Visible" [] outBool
>
> setVisible :: Bool -> Com IDispatch -> IO ()
> setVisible b  = propertySet "Visible" [inBool b]
>
> getLen :: Com IDispatch -> IO Int
> getLen         = propertyGet "Len" [] outInt
```

Suppose the variable foo points to an object supporting the Foo interface. A program might use the properties as:

```
> do foo # setVisible True             -- make visible
>    l <- foo # geLen                 -- get the length
>    ...
```

Methods

Methods are slightly more complex. The function function is used when the method returns a result (other than void), if not, the function method is used. The first argument is the method name. The next argument is a (possibly empty) list of input arguments using the in function. The following arguments are out or inout functions specifying the returned arguments. The function function has as the last argument an out function specifying the result.

Suppose we have the following methods:

```
> dispinterface Foo {
>   methods:
>    void        Speak( [in] BSTR s );
>    IDispatch*  LoadName( [in] VARIANT v, [in] BSTR s );
> };
```

The methods are defined in Haskell as:

```
> speak :: String -> Com IDispatch -> IO ()
> speak s      = method0 "Speak"  [inString s]
>
> loadName :: Variant a => a -> String -> Com IDispatch -> IO (Com IDispatch)
> loadName v s = function1 "LoadName" [inVariant v,inString s] outDispatch
```

Note that we can pass any Variant type by using the Variant class. Any in,out or inout arguments will have their marshalling functions following the input argument list. To correctly type this in Haskell, the method and function functions get the arity of the tuple appended to their name. For example:

```
> dispinterface Foo {
>  methods:
>    void gnu( [in] float f, [in,out] int* i, [out] BSTR* s );
>    void Load( [in] VARIANT* path, [out] long* charid,
>                                [out,retval] long* reqid );
>    BOOL Speak( [in] BSTR s, [out] long* reqid );
> };
```

The methods are defined in Haskell as:

```
> gnu :: Float -> Int -> Com IDispatch -> IO (Int,String)
> gnu f i       = method2 "gnu"  [inFloat f] (inoutInt i) outString
>
> load :: Variant a => a -> Com IDispatch -> IO (Int,Int)
> load path     = method2 "Load" [inVariant path] outInt outInt
```

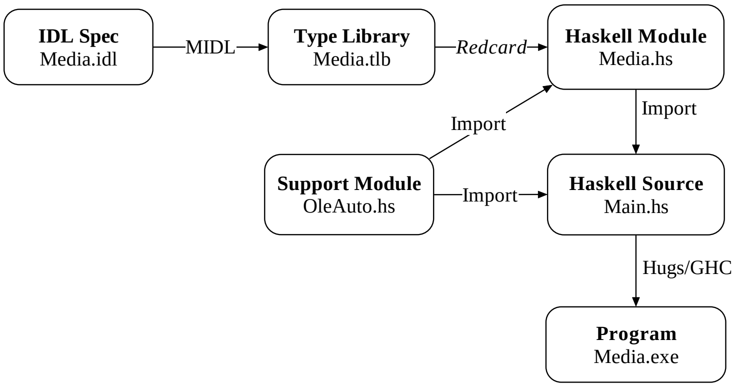


Figure 6.3: *Compilation process for Automation components*

```
>
> speak :: String -> Com IDispatch -> IO (Bool,Int)
> speak s       = function2 "Speak" [inString s] outInt outBool
```

This concludes the description of how to write Automation definitions by hand. The next sections will describe how Redcard can do the translation of Automation interfaces automatically.

6.5 Automatic Automation

By giving the `-auto` switch, Recard automatically translates Automation interfaces. The media control for example supports Automation. But we have used the media control as a COM object ? Indeed, since the media control is a so called `dual` interface, it supports both mechanisms to access its functionality. The IDL is described as[2]:

```
> [uuid(56A868B0-0AD4-11CE-B03A-0020AF0BA770)]
> library QuartzTypeLib
> {
> importlib("stdole32.tlb");
>
> [uuid(56A868B1-0AD4-11CE-B03A-0020AF0BA770),
>  dual,oleautomation
> ]
> interface IMediaControl : IDispatch {
>    [id(0)] HRESULT Run();
>    [id(1)] HRESULT RenderFile([in] BSTR strFilename);
>    ...
> };
>
> [uuid(E436EBB3-524F-11CE-9F53-0020AF0BA770)]
> coclass FilgraphManager {
>    [default] interface IMediaControl;
>    interface IMediaEvent;
>    ...
> };
>
```

[2] Compare this to the IDL given in chapter 4.

```
> ...
> };
```

The type library can be compiled to a Haskell module:

```
> redcard -auto  c:\windows\system\quartz.dll media.hs
```

Redcard will generate the following module:

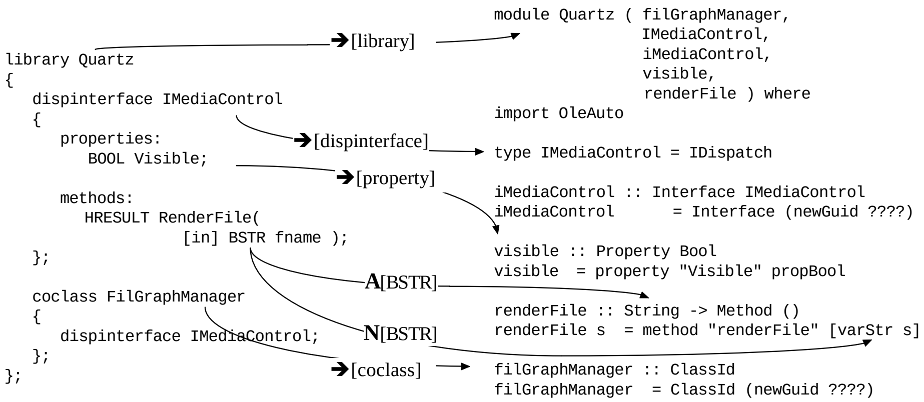
```
> ------------------------------------------------------------
> -- redcard, version 0.1, (c) 1997 Daan Leijen
> -- This file is automatically generated from a type library
> ------------------------------------------------------------
> module Media (  filGraphManager
>               , IMediaControl
>               , iMediaControl
>               , run
>               , renderFile
> ...
> import Automation
>
> ------------------------------------------------------------
> -- dispatch interface IMediaControl
> ------------------------------------------------------------
> type IMediaControl = IDispatch
>
> iidIMediaControl :: Interface IDispatch
> iidIMediaControl   = Interface (newGuid 1453877425 2772 4558
>                              [176 ,58 ,0 ,32 ,175 ,11 ,167 ,112])
>
> run                       = method0 "Run" []
> renderFile strFilename    = method0 "RenderFile" [inString strFilename]
> ...
>
> ------------------------------------------------------------
> -- coclass filgraphManager
> ------------------------------------------------------------
> clsidFilgraphManager   = mkCLSID (newGuid (-466162765) 21071 4558
>                              [159 ,83 ,0 ,32 ,175 ,11 ,167 ,112])
> ...
```

Redcard only imports the `Automation` module. Classes are translated as before but all Automation interfaces are of the same type as the `IDispatch` interface. The main difference is in the translation of the methods. No complicated marshalling function and green-card code, just one simple Haskell function definition. A module generated this way can immediately be used by any Haskell code without any further compilation by green-card or the C compiler. Figure 6.3 illustrates the compilation process for an automation module.

The next section will formally define the translation done by Redcard in the same way as done in chapter 4.

6.6 Automation translation

Every argument in Automation will eventually be put in a `VARIANT` structure. This structure contains besides the value also a tag that gives you the type of the value.



```
library Quartz
{
   dispinterface IMediaControl
   {
      properties:
         BOOL Visible;

      methods:
        HRESULT RenderFile(
                   [in] BSTR fname );
   };

   coclass FilGraphManager
   {
      dispinterface IMediaControl;
   };
};
```

```
module Quartz ( filGraphManager,
                IMediaControl,
                iMediaControl,
                visible,
                renderFile ) where
import OleAuto

type IMediaControl = IDispatch

iMediaControl :: Interface IMediaControl
iMediaControl      = Interface (newGuid ????)

visible :: Property Bool
visible  = property "Visible" propBool

renderFile :: String -> Method ()
renderFile s  = method "renderFile" [varStr s]

filGraphManager :: ClassId
filGraphManager  = ClassId (newGuid ????)
```

Figure 6.4: *Translation functions used for the media control*

All arguments are passed as `VARIANT`'s to the `Invoke` function of the `IDispatch` interface. The Haskell type `Variant` represents this structure in Haskell.

To marshall and unmarshall Automation types, they just need to be mapped to a `Variant` stucture which is later marshalled automatically during the `Invoke` call (with automatically generated Redcard code). For each Automation type, the following primitive injection and projection functions are defined (in `Automation`):

$$\begin{array}{ll} \text{in}\mathbf{N}[\![\tau]\!] & :: \mathbf{A}[\![\tau]\!] \rightarrow \text{Variant} \\ \text{res}\mathbf{N}[\![\tau]\!] & :: \text{Variant} \ \rightarrow \text{IO } \mathbf{A}[\![\tau]\!] \end{array}$$

Since projection could fail due to a type error, it has type `IO`. These functions seem a perfect application for overloading. However, Haskell 1.4 doesn't allow overlapping instances which prohibit us to define an overloaded instance for the `String` type. However, just recently, we found a technique where the same behaviour as overlapping instances is possible within Haskell, when the overlapping instances are already known. This technique is used in the definition of the `Show` class in haskell and can also be used here to overload the `var` and `res` functions.

The next sections formally define the translation for each IDL declaration to Haskell. However, the translation is much simpler than in chapter 4. The only translation functions needed are **N**, **A** and **D** ! Figure 6.4 shows how these translation functions are used when translating the media control.

6.6.1 Libraries

$$\boxed{\textbf{library}\ x\ \textbf{\{}\ \ \{\ \textit{definition}\ d_i\ ;\ \}^n\ \ \textbf{\}}}$$

$$\mapsto$$

$$\boxed{\textbf{module}\ x\ \textbf{(}\ d_1\ ,\ldots,\ d_n\ \textbf{)}\ \textbf{where}}$$

import Automation

The only difference in the translation of libraries is that the Automation module is imported.

6.6.2 Type definitions

$$\boxed{\textbf{typedef}\ t\ x\ ;}$$

$$\mapsto$$

$\boxed{\textbf{type } x = \mathbf{A}[\![\mathrm{t}]\!]}$

Type definitions are translated to a type declaration in Haskell. Other than readability, type definitions have no effect on the translation process:

$$\begin{aligned}
\mathbf{N}[\![x]\!] &= \mathbf{N}[\![t]\!]\\
\mathbf{A}[\![x]\!] &= \mathbf{A}[\![t]\!]\\
\mathbf{D}[\![x]\!] &= \mathbf{D}[\![t]\!]
\end{aligned}$$

6.6.3 Enumerations

$\boxed{\textbf{enum } x \ \pmb{\{} \ \ \{ \, e_i \, [= k_i] \ \ , \ \ \}^n \ \ \pmb{\}}}$

Enumerations are translated just as in section 4.5.3 but they are not directly usable as Automation types. The (generated) `toEnum` and `fromEnum` functions need to be used to pass them as integers to an Automation object.

6.6.4 Structures, Unions and Interfaces

These declarations are not translated when the `-auto` switch is used. The declarations are not usable within Automation and require a separate greencard and C-compiler run.

6.6.5 Com classes

$[\textbf{uuid}(\mathit{guid}_x)]$
$\boxed{\textbf{coclass}\ \mathrm{x}\ \pmb{\{} \ \ \ldots \ \ \pmb{\}}}$

$$\mapsto$$

$\boxed{\mathrm{clsid}x :: \mathrm{CLSID}}$

$\mathrm{clsid}x = \mathrm{mkCLSID}\ (\mathrm{newGuid}\ \mathit{guid}_x)$

Each COM class is translated to a function of type `CLSID`. Since a class is not a first-class value, there is no need to define any translation function.

6.6.6 Automation interfaces

$[\textbf{uuid}(\mathit{guid}_x)]$
$\boxed{\textbf{dispinterface}\ \mathrm{x}\ \pmb{\{} \ \ \ldots \ \ \pmb{\}}}$

$$\mapsto$$

$\boxed{\textbf{type } x = \textbf{IDispatch}}$

$\boxed{\mathrm{iid}x :: \mathrm{Interface}\ x}$
$\mathrm{iid}x = \textbf{Interface}\ (\mathrm{newGuid}\ \mathit{guid}_x)$

Automation interfaces are just equivalent to the `IDispatch` interface. Redcard generates a type synonym to make the type definitions more readable. Secondly, the `Interface` still uses the real `IID` instead of the `IID` of `IDispatch`. A `queryInterface` for a specific interface won't degrade to a `query` for the general `IDispatch`.

Since an automation interface is basically a typedef for `IDispatch`, the translation functions simply use the `IDispatch` type:

$$\begin{array}{ll} \mathbf{N}\llbracket x\rrbracket & = \mathbf{N}\llbracket \text{IDispatch*}\rrbracket \\ \mathbf{A}\llbracket x\rrbracket & = \mathbf{A}\llbracket \text{IDispatch*}\rrbracket \\ \mathbf{D}\llbracket x\rrbracket & = \mathbf{D}\llbracket \text{IDispatch*}\rrbracket \end{array}$$

6.6.7 Properties

$$\boxed{\textbf{dispinterface}\ x\ \{\quad \textbf{properties}\text{: } \{\ type_i\ p_i\ ;\ \}^I\quad \}}$$

$$\mapsto$$

$$\boxed{\begin{array}{l} \text{get}p_i :: \text{Com IDispatch } \to \text{IO } \mathbf{A}\llbracket type_i\rrbracket \\ \text{set}p_i :: \mathbf{A}\llbracket type_i\rrbracket \to \text{Com IDispatch } \to \text{IO ()} \end{array}}$$

$$\begin{array}{l} \text{get}p_i = \text{propertyGet “}p_i\text{” [] out}\mathbf{N}\llbracket type_i\rrbracket \\ \text{set}p_i\ x\ = \text{propertySet “}p_i\text{” [in}\mathbf{N}\llbracket type_i\rrbracket\ x] \end{array}$$

In Haskell, a distinction is made between the identifier and the name of of the property. This is especially important in Haskell, since the identifier is subject to the name translation defined in section 4.2. If the **readonly** attribute is given, the `set` will not be generated. `propertyGet` and `propertySet` are primitive methods of the `Automation` module.

6.6.8 Methods

$$\boxed{\begin{array}{l} \textbf{dispinterface}\ x\ \{ \\ \qquad\qquad \textbf{methods} : \{\ type_m\ method_m\ (\{[\textbf{in}]\ itype_i\ in_i\}^I, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \{[\textbf{in,out}]\ iotype_j\ io_j\}^J, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \{[\textbf{out}]\ otype_k\ out_k\}^K\ )\ \}^M\quad \} \end{array}}$$

$$\mapsto$$

$$\boxed{\begin{array}{l} method_m :: \mathbf{A}\llbracket itype_1\rrbracket \to ... \to \mathbf{A}\llbracket itype_I\rrbracket \\ \qquad\qquad \to \mathbf{OA}\llbracket iotype_1\rrbracket \to ... \to \mathbf{OA}\llbracket iotype_J\rrbracket \to \text{Com IDispatch} \\ \qquad\qquad \to \text{IO } ([\mathbf{A}\llbracket type_m\rrbracket]\ \{,\mathbf{OA}\llbracket iotype_j\rrbracket\}^J\ \{,\mathbf{OA}\llbracket otype_k\rrbracket\}^K) \end{array}}$$

$$\begin{array}{l} method_m\ in_1...in_I\ io_1...io_J \\ \qquad = \mathbf{MF}\llbracket J+K\rrbracket\ \text{“}method_m\text{”} \\ \qquad\qquad [\text{var}\mathbf{N}\llbracket type_1\rrbracket\ in_1\ ,...,\ \text{var}\mathbf{N}\llbracket type_I\rrbracket\ in_I] \\ \qquad\qquad \{\text{inout}\mathbf{N}\llbracket iotype_j\rrbracket\ io_j\}^J\ \{\text{out}\mathbf{N}\llbracket otype_k\rrbracket\}^K \end{array}$$

There are some new functions introduced. Since [**in,out**] and [**out**] parameters are always defined as a pointer type, the **OA** function is introduced to strip off the pointer:

$$\mathbf{OA}\llbracket t*\rrbracket \quad = \mathbf{A}\llbracket t\rrbracket$$

The function **MF** is defined as:

$$\mathbf{MF}\llbracket i\rrbracket = \begin{cases} \text{method}i & \text{if } type_m = \textbf{void} \\ \text{function}(i+1) & \text{otherwise} \end{cases}$$

`method`*i* and `function`*i* are primitive functions in `Automation`. For example:

```
> method1   :: String -> [Variant] -> (Variant,Variant -> IO a)
>                -> Com IDispatch -> IO a
>
> function2 :: String -> [Variant] -> (Variant,Variant -> IO a)
>                -> (Variant -> IO b) -> Com IDispatch -> IO (b,a)
```

[**in**] parameters need to marshall the argument into a variant; the in$\mathbf{N}[\![\tau]\!]$ function can be used to accomplish that.

[**in,out**] parameters need to marshall and unmarshall the argument. The **MF** function expects a tuple containing the marshalled argument and the unmarshall function for the result:

$$\begin{array}{ll} \text{inout}\mathbf{N}[\![\tau]\!] & :: \mathbf{A}[\![\tau]\!] \to (\text{Variant }, \text{Variant } \to \text{IO } \mathbf{A}[\![\tau]\!]) \\ \text{inout}\mathbf{N}[\![\tau]\!] & = \backslash x \to (\text{in}\mathbf{N}[\![\tau]\!]\ x, \text{res}\mathbf{N}[\![\tau]\!]) \end{array}$$

It seems that [**out**] parameters can use the res$\mathbf{N}[\![\tau]\!]$ function and don't need an initial value like [**in,out**] parameters, but most Automation controllers require an initialized [**out**] parameter. The function **D** is therefore needed to define:

$$\begin{array}{ll} \text{out}\mathbf{N}[\![\tau]\!] & :: (\text{Variant }, \text{Variant } \to \text{IO } \mathbf{A}[\![\tau]\!]) \\ \text{out}\mathbf{N}[\![\tau]\!] & = \text{inout}\mathbf{N}[\![\tau]\!]\ \mathbf{D}[\![\tau]\!] \end{array}$$

6.7 Automation controller

Using the above translation, there is hardly any difference between COM and Automation objects for the programmer. It is quite amazing that all the Automation complexity can be hidden this way. Since current focus of research is in scripting components from Haskell, we have used the Automation interface extensively and with success [ML98]. The only real good Automation controller to date is Visual Basic and it seems that Haskell can rival this language in many applications.

Chapter 7

Conclusion & Future work

In this thesis a framework for calling COM components is described. We have developed a typed interface for calling COM from Haskell and a compiler for translating COM interfaces to Haskell. Secondly we have done a case study to see how well Haskell can be used to program COM components. The results are very encouraging. We can already use Haskell to program off-the-shelf components and the features of Haskell can be used to good effect when glueing components together. However there are still many things that would be interesting to investigate further.

The next obvious goal is to implement COM components in Haskell. I have already implemented a prototype system for creating Haskell Automation objects, called HakellObject. This allows Haskell components to be used from environments like Visual Basic, Haskell ! or to be embedded in web pages. Figure 7.1 shows how a client uses two Haskell Automation components. Each Haskell object runs within the HaskellObject DLL. This DLL exposes generic Automation objects that dispatch each method call to Haskell code. This can be accomplished at run-time through the type information available with Automation components. It is not possible yet to create pure COM objects in Haskell; a tool like Redcard is needed to compile special code for entry points in Haskell and argument marshalling. However much of the machinery needed is already developed with Redcard.

Both HaskellObject and HaskellScript need to be able to parse and execute Haskell code. This led to the development of an interface specification, called HaskellServer, to access Haskell interpreters/compilers. Interpreters or compilers that implement this interface become Haskell Servers and can be used by HaskellObject and HaskellScript to execute the Haskell code. At the moment I have just implemented this interface for the Hugs interpreter, but it could be written for

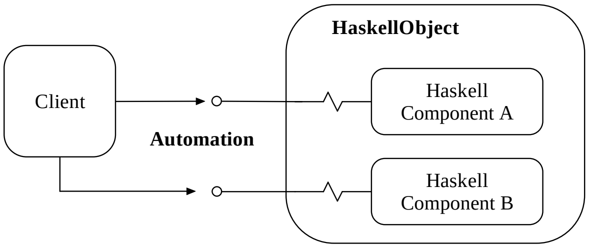


Figure 7.1: *Automation components in Haskell with HaskellObject*

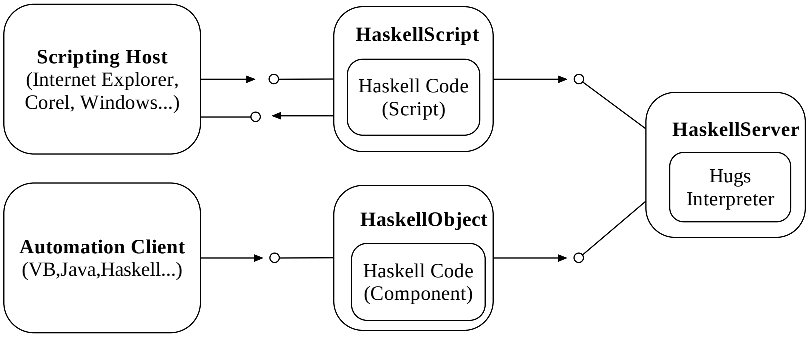


Figure 7.2: *HaskellServer is used by HaskellScript and HaskellObject*

other compilers or interpretes also. Figure 7.2 shows how both HaskellObject and HaskellScript use the HaskellServer to execute the Haskell code. I envision extending the HaskellServer interface to browse the symbol tables, type information etc. It can be used to create hybrid applications that use Haskell and to create common development tools like editors, browers and profilers for Haskell interpreters and compilers. More information about these projects can be found on the ActiveHaskell home page: `http://www.haskell.org/active/activehaskell.html`.

We have shown how COM components can be used from a strongly-typed, pure functional language as Haskell. It is shown how complex component interactions can be specified using conventional functional programming techniques. The ability to connect to the real-world enables Haskell to be compared with other programming languages and the results are encouraging. I hope the results are encouraging enough to convince people of using Haskell in real-world software projects and thus taking software development to a new level.

Bibliography

[Bro95] Kraig Brockschmidt. *Inside Ole (second edition)*. Microsoft Press, 1995.

[Cha96] David Chappel. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[Den97] Adam Denning. *ActiveX controls inside out (second edition)*. Microsoft Press, 1997.

[EH97] Conall Elliot and Paul Hudak. Functional reactive animations. *International Conference on Functional Programming, Amsterdam*, June 1997.

[Eri95] Erich Gamma, Richard Helm, Ralph Johnson and John Vissides. *Design Patterns*. Addison-Wesley, 1995.

[Eri97] Erik Meijer, Koen Claessen, Joost van Dijk and Arjan van Yzendoorn. The design and implementation of Mondrian. *The Haskell Workshop, Amsterdam*, 1997.

[GH95] Andrew D. Gordon and Kevin Hammond. Monadic I/O in Haskell 1.3. June 1995.

[HF92] Paul Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hug89] John Hughes. Why functional programming matters. *Computer journal*, 32(2):98–107, 1989.

[JJ97] Patrik Jansson and Johan Jeuring. Polyp – a polytypic programming language extension. *Principles of Programming Languages*, ACM press:470–482, 1997.

[JL95] Simon Peyton Jones and John Launchbury. State in Haskell. *Lisp and symbolic computation*, 8(4):293–341, 1995.

[Jon93] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 52–61, June 1993.

[JW93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. *POPL*, 20:71–84, 1993.

[Kli93] Paul Klint. A guide to ToolBus programming. Technical report, University of Amsterdam, May 1993.

[Mic92] Microsoft. *The COM reference*. Microsoft Press, 1992.

[Mic97a] Microsoft. *Automation programmers reference*. Microsoft Press, 1997.

[Mic97b] Microsoft. *Programming Microsoft Agent.* Microsoft Press, 1997.

[ML98] Erik Meijer and Daan Leijen. Client side web scripting with HaskellScript. *submitted to ICFP'98*, August 1998.

[OMG93] OMG. *The Common Object Request Broker: Architecture and Specification (revision 1.2).* Object Management Group, 1993. OMG document number 93.12.43.

[Pau96] Paul Hudak, Tom Makucevich, Syam Gadde and Bo Whong. Haskore music notation – an algebra of music. *to appear in the Journal of functional programming*, 6(3):465–483, May 1996.

[Pet97] John Peterson (editor). Report on the programming language Haskell, version 1.4. Technical report, Yale university, http://www.haskell.org/, April 1997.

[Pla96] David S. Platt. *The essence of OLE with ActiveX.* Prentice Hall, 1996.

[Rog97] Dale Rogerson. *Inside COM.* Microsoft Press, 1997.

[Sie96] Jon Siegel. *CORBA, Fundamentals and Programming.* John Wiley & Sons, 1996.

[Sig98] Sigbjorn Finne, Daan Leijen, Erik Meijer and Simon Peyton-Jones. H/Direct a binary foreign language interface for Haskell. *International Conference on Functional Programming, Baltimore MD, USA*, September 1998.

[Sim97] Simon Peyton-Jones, Thomas Nordin and Alastair Reid. Greencard: a foreign language interface for haskell. *Proc. Haskell Workshop, Amsterdam*, June 1997.

[Sim98] Simon Peyton-Jones, Erik Meijer and Daan Leijen. Scripting com components in haskell. *Fifth International Conference on Software Reuse, Victoria, BC, Canada*, June 1998.

[Wad92] Philip Wadler. The essence of functional programming. *19'th Annual symposium on Principles of Programming Languages*, January 1992.

[Wad98] Philip Wadler. A prettier printer. *Draft paper, http://cm.bell-labs.com/cm/cs/who/wadler/papers/prettier/prettier.ps*, March 1998.

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.

[Wil90] Tony Williams. On inheritance, what it means and how to use it. *Microsoft, internal report*, March 1990.