Functional Scripting

Daan Leijen and Erik Meijer

Utrecht University
Department of Computer Science
{erik,daan}@cs.uu.nl

1 Introduction

Component-based programming and scripting support a style of rapid prototyping program development where reusable off-the-shelf software components are glued together to build new applications. One of the reasons of Visual Basic's popularity is its ability to script 'ready made' GUI components in a fancy development environment. Since all the hard work is done inside the components, not only professional programmers but also end-users are able to create impressive applications in a short period of time.

Visual Basic uses COM as the underlying component framework. COM is a language independent binary standard for implementing and using software components[10, 8]. Vendors can use C++, Java, COBOL or any other language suited for the specific task to create a COM component. The second ingredient that Visual Basic needs is the "glue" to compose different components together. ActiveX is a standard set of COM interfaces that specify how components can be composed and how they interact in an interactive and graphical environment [1, 2]. The interfaces deal with drawing, event notification, persistence and much more. The language independence of COM allowed us to create a COM binding for Haskell [12, 11, 3, 7]. By being able to use COM and ActiveX components from Haskell we can take advantage of the enormous set of software components that is available to the Visual Basic programmer.

Of course Haskell in not the only possible scripting language. However most contemporary scripting languages such as Visual Basic, ECMAScript, Perl, Tcl, etc. lack some of the essential properties of a truly high-level scripting language: *strong typing*, *parametric polymorphism* and *overloading* for flexibility of composition without sacrificing type safety, the unconditional principles of *abstraction*, *parameterization*, *correspondence* to exploit regularity and support reuse in the small, *lazyness* to hide non-termination effects on the way from the producer to the consumer, and *monads* to hide other side-effects on the way from the producer to the consumer.

The type system of Haskell catches many potential bugs at compile time; most contemporary scripting languages issue the bug at run-time at the client-site! The type system is even vital in providing a safe interface to the COM system. Many peoply find monads and higher-order functions esoteric features, but they

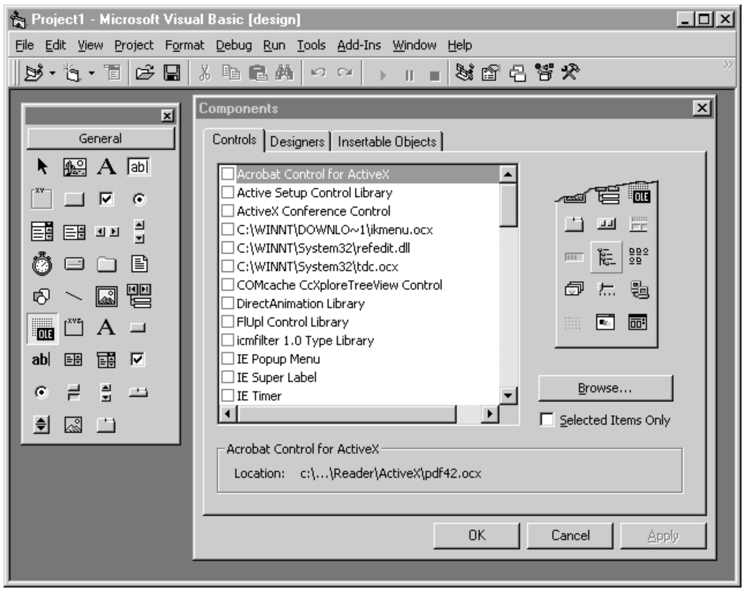

Fig. 1. Visual Basic uses activeX to script components

are essential to make objects and methods first-class values and thus composable and reusable.

2 Agents, or using ActiveX components

We will start with a simple example of using an ActiveX component from Haskell: a deluxe version of the famous "Hello World" program. Microsoft Agent [9] is an ActiveX component that creates cartoon characters (agents) on the screen that can talk, move, perform animations and even listen to commands. Agents can provide user-friendly user interaction in applications such as Word or Excel, or interactively guide users through an internet site. We will use the agent to say "Hello world" to the audience.

Untyped languages like PERL or VB can use COM component directly, but with the risk of many possible run-time errors. In Haskell a typed description of that component is needed to avoid those run-time errors. COM components use a standard language to describe their interfaces: the interface description language (IDL). An IDL compiler is used to translate the IDL description of the component to a target language like C++ or Java. In collaboration with the University of Glasgow, we have made a freely available IDL compiler called *H/Direct* [11] that automatically generates appropriate Haskell modules for a COM component from its IDL specification. All the components used in this article (and many others) are included with HaskellScript and H/Direct is therefore not described any further. The Agent component can be used from Haskell by importing the (automatically generated) `AgentScript` module.

To create an instance of a COM component we can call the function [1]:

```
comCreateObject :: String -> IID a -> IO a
```

The first argument of this function identifies the object using a version independent programmatic identifier, for example the Agent component is identified by `"Agent.Server"`. The COM infrastructure takes care of associating the identifier with actual code on the system, loading the component and returning the requested interface pointer.

The second argument of function `comCreateObject` is the *interface identifier*, often abbreviated as IID, of the initial interface on the component requested by the programmer. The IID of the Agent interface is exported as `iidIAgent` from the `AgentScript` module.

```
iidIAgent :: IID (IAgent ())
```

The polymorphic type of `comCreateObject` ensures that the value returned by `comCreateObject "Agent.Server" iidAgent` is of type `IAgent ()`, thus ensuring a connection between the IID of the requested interface and the returned interface.

To create an agent character we call the method `loadCharacter` of the `IAgent` interface. To free the character in the end we must call `unloadCharacter`.

```
loadCharacter   :: FilePath -> IAgent a -> IO (IAgentCharacter ())
unloadCharacter :: IAgentCharacter a -> IAgent b -> IO ()
```

The first argument to `loadCharacter` is a path to the character resource file. The second argument is the 'self' pointer to the agent interface. The self argument comes last to mimic the usual dot-notation for calling methods in an OO language using postfix application `#`:

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

The `IAgentCharacter` interface has methods for animating the character in various ways:

```
slowShow :: IAgentCharacter a -> IO ReqID
speak    :: String -> IAgentCharacter a -> IO ReqID
play     :: String -> IAgentCharacter a -> IO ReqID
moveTo   :: (Int,Int) -> IAgentCharacter a -> IO ReqID
slowHide :: IAgentCharacter a -> IO ReqID
```

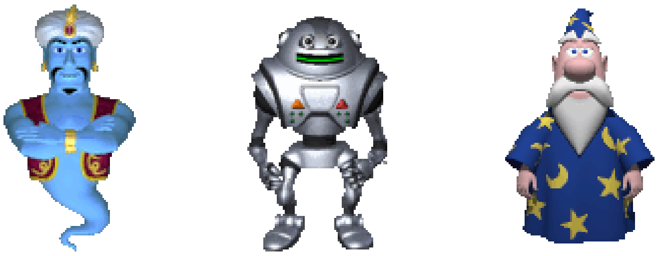

Fig. 2. Three of the Agent characters

Each of these methods returns a request id `ReqID` that uniquely identifies the the command being executed. These request ids are used for synchronizing multiple parallel agents. For our simple example they can be ignored; the genie will show itself, greet the audience, speak a timeless sentence and disappear without interacting with other agents:

```
module AgentDemo where
import AgentScript

main :: IO ()
main = comRun $
  do{ server <- comCreateObject "Agent.Server" iidIAgent
    ; genie  <- server # loadCharacter geniePath
    ; genie # slowShow
    ; genie # play "Greet"
    ; genie # speak "Hello world!"
    ; genie # slowHide
    ; server # unloadCharacter genie
    }
```

Function `comRun` takes care of initializing and releasing the COM library.

2.1 Making COM more convenient

Since methods and properties of COM components are exposed as normal Haskell functions, it is possible to create custom control functions to manipulate COM objects. For example, Visual Basic has the `with` keyword to execute a list of methods on one single object:

```
With Genie
```

[1] there are several other, similar, functions for creating objects as well.

```
  .Show
  .Play "Greet"
  .Speak "Hello"
  .Hide
End With
```

It is easy to define a function in Haskell which has the same effect.

```
with :: obj -> [(obj -> IO a)] -> IO ()
with obj = \meths -> sequence [ obj # meth | meth <- meths ]
```

The function `with` first applies every method to the object, building a list of IO computations, which are than executed from left to right using the standard function `sequence`. Using the new function `with` we can rewrite the core of our example as follows:

```
with genie [ showUp, play "Greet", speak "Hello", hide ]
```

2.2 Extending a given interface

Besides having animations like `Greet`, the characters also have a repetoire of closing animations like `GreetReturn`. Such animation smoothly transitions the character back to its normal state. The demo looks better if written as:

```
 with genie
   [ showUp,
   , play "Greet", play "GreetReturn"
   , speak "Hello"
   , play "Wave", play "WaveReturn"
   , hide
   ]
```

Since methods and properties of COM components are exposed as normal Haskell functions, a given interface can be seamlessly extended with new methods and properties. The syntax for calling new, compound methods is exactly the same as for primitive methods. For example we can construct "smooth" animations by making use of the fact that if an animation `anim` has a corresponding closing animation, it is called `anim++"Return"`:

```
smoothPlay :: String -> IO b -> IAgentCharacter a -> IO ReqID
smoothPlay anim = \action -> \agent ->
  do{ agent # play anim
    ; action
    ; agent # play (anim ++ "Return")
    }

skip :: IO ()
skip = do{ return () }
```

This newly defined method can be used in just the same way as 'primitive' methods:

```
with genie
  [ showUp
  , smoothPlay "Greet" skip
  , speak "Hello"
  , smoothPlay "Wave" skip
  , hide
  ]
```

Allthough the given example is trivial, the same approach can be extended to complex component interactions where the ability to treat methods and computations as first class values is vital. It allowed us to define higher-order combinators to compose parallel agent animations[12]. In conventional languages, the complexity of synchronizing the parallel components would be completely exposed to the programmer.

2.3 Typing components

Allthough Haskell is not an object based language, it is possible to model COM components using algebraic data types and the Hindley-Milner subset of the typing system; the same approach can thus be used in ML like languages too.

Every COM interface is required to inherit from the `IUnknown` interface. The IDL specification this interface is:

```
[ uuid(00000000-0000-0000-C000-000000000046) ]
interface IUnknown {
    HRESULT QueryInterface ([in]IID* riid, [out]void** ppvObj);
    unsigned long AddRef();
    unsigned long Release();
};
```

Since a COM component can implement many interfaces, the `queryInterface` method is used to switch between the different interfaces of component. In IDL, the connection between the passed IID and the returned interface can not be expressed. In conventional languages, an explicit type cast is necessary:

```
IAgent* pIAgent;
HRESULT hr = pUnknown->QueryInterface (&IID_IAgent, (void**)&pIAgent);
```

In a Hindley-Milner type system, like ML or Haskell, parametric polymorphism can be used to create a typed connections between the passed IID and the returned interface. In some sence, `queryInterface` resembles a function like `head :: [a] -> a` where polymorphism ensures that the returned element is of the same type as the list elements. In the same way, we ensure that the returned interface is of the same type as the IID of the interface:

```
queryInterface :: IID a -> IUnknown b -> IO a
```

We use the same technique in many other COM functions, for example with `comCreateObject`.

The `AddRef` and `Release` methods are used to reference count objects. Whenever the reference count drops to zero (due to a call to `Release`), the object should destroy itself. In Haskell, we use *foreign objects* to hold a reference to a COM component. Foreign objects interact with the garbage collector and will automatically take care of releasing COM components.

```
data Unknown a    = MkUnknown ForeignObj
type IUnknown a   = Unknown a
```

`IUnknown` is a simple type synonym for the `Unknown` data type. The next section will explain how the type variable `a` is used to encode inheritance.

2.4 Typing inheritance

Surprisingly, polymorphism can also be used to express the inheritance relationship of COM interfaces. For example, the MS Agent interfaces all inherit from the `IDispatch` interface which in turn inherits from the `IUnknown` interface:

```
interface IDispatch : IUnknown {
  HRESULT GetTypeInfoCount( [out] int* count );
  HRESULT GetTypeInfo(...);
  HRESULT GetIDsOfNames(...);
  HRESULT Invoke(...);
};
```

A method like `queryInterface` should apply to both `IUnknown` and `IDispatch` interfaces (and in general any other interface inheriting from `IUnknown`) but the method `GetTypeInfoCount` should only apply to interfaces inheriting from `IDispatch`. Haskell does not have subtyping, but it is possible to use polymorphism to model the inheritance relation.

In the previous definition of the `Unknown` data type we used a type variable `a`. This type variable will function as a slot in which the inheritance tree will be encoded. The IDispatch interface will have type `Unknown (Dispatch a)` and an IAgent interface `Unknown (Dispatch (Agent a))`. The H/Direct compiler automatically generates type synonyms like `IDispatch` and `IAgent`:

```
data Dispatch a   = MkDispatch              -- dummy data type
type IDispatch    = Unknown (Dispatch a)

iidIDispatch :: IID (IDispatch ())

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

The type system will now ensure that `getTypeInfoCount` can only be called with an interface inheriting from `IDispatch` but not with an `IUnknown` interface. Note how any interface passed as argument has type `IFoo a` while an interface result has type `IFoo ()`. This implements the covariance rule; an argument can be any interface *inheriting* from IFoo while the returned interface is exactly an IFoo interface and nothing more.

3 Using multiple components

In this section we will develop a Haskell program that lets an agent character explains the meaning of a word. The word is selected inside MS Word document and the agent will look that word up in an online dictionary on the internet.

As a first step the agent character will say the currently selected word. The script starts by creating an agent character:

```
module ReadDoc where

import AgentScript
import WordScript

main =
  do{ server <- comCreateObject "Agent.Server" iidIAgent
    ; merlin <- server # loadCharacter merlinPath
    ; with merlin
        [ showUp
        , smoothPlay "Greet" skip
        , readSelection
        , smoothPlay "Wave" skip
        ]
```

We now need to define `readSelection` method. First it connects to the active Word application using the COM library function `getActiveObject`:

```
readSelection merlin =
  do{ word <- getActiveObject "Word.Application"
    ; text  <- word # getSelection ## getText
    ; with merlin [ smoothPlay "Read" (do{ speak text }) ]
    }
```

Since methods are first class values, we can compose methods using the `##` operator. The composition of two methods `m1 ## m2` is a new method itself. The operator `##` is defined as reverse monadic composition:

```
f ## g = \a -> do{ b <- f a; g b }
```

3.1 Webster

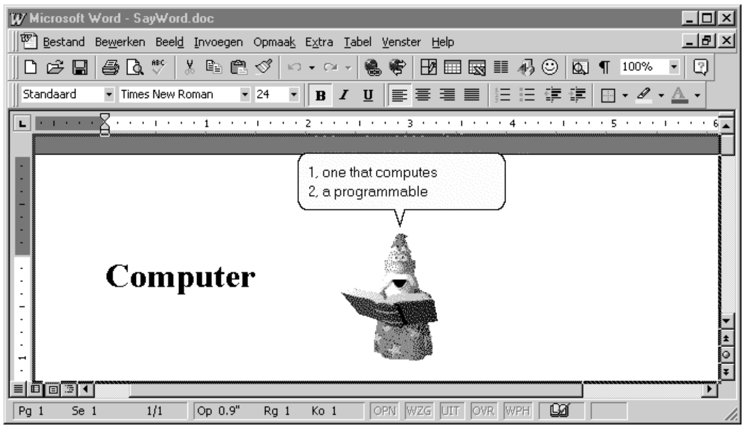

Fig. 3. Merling tells the meaning of "computer"

The previous program will now be extended such that Merlin tells the *meaning* of a selected word instead of just repeating it. Merlin will retrieve the meaning of the word by invoking a CGI script on the Webster web-site. The Internet Explorer component is used to connect to the Webster site.

```
with merlin
  [ play "Read"
  , tellMeaning (words text)
  , play "ReadReturn"
  ]
```

If there is no word selected, genie will speak an error message. Otherwise we create an instance of the browser component, direct it to look up the meaning of the word on the Webster web-site, and let genie read the this text:

```
tellMeaning ("") merlin
  = merlin # speak "Sorry, you have to specify a valid word"

tellMeaning (word) merlin
```

```
    = do{ ie <- comCreateBrowser
        ; ie # webster word
        ; text <- ie # (getDocument ## getBody ## getInnerText)
        ; merlin # smoothPlay "Read" (do{ speak (meanings text) })
        }

  webster :: String -> IWindow -> IO ()
  webster = \word -> \ie ->
    do{ ie # navigate "http://www.m-w.com/cgi-bin/dictionary?va="
                      ++word }
```

The `meanings` function parses the HTML and extracts the possible meanings of the word. The powerful list processing functions of Haskell are very convenient here.

4 Scripting hosts

The previous example shows how a script uses MS Word and the Internet Explorer as COM objects. However, it would be much nicer to embed the scripts inside a Word document or HTML page. ActiveX scripting provides exactly this capability.

Any application that implements the `IActiveScriptHost` interface can use any language that implements the `IActiveScript` interface. Examples of applications that are scripting hosts are the Internet Explorer and even the Windows operating system itself. We have made an implementation of the `IActiveScript` interface on top of the Hugs interpreter. Haskell scripts can now be embedded inside HTML or ASP pages, or directly exectuted from Windows by giving the `.hash` extension. Other examples of languages with a script engine are JavaScript, VBScript, Perl, Python and REXX.

4.1 Microsoft Internet Explorer, or event handling

The Dynamic HTML object model (DOM) makes everything, headers, images, hyperlinks, individual characters, really everything, in a web page scriptable [4]. The DHTML object model is very rich and it takes more than a full paper to explain all the ins and outs [3].

While the DOM gives scripts access to a web-page via its rich API of methods and properties, style sheets control the visual appearance of elements on the screen. Of course, style sheets are fully scriptable themselves.

Pages are animated by defining scripts that react on user events. For example we can attach an event handler to a paragraph that changes the color of the paragraph when the user moves the mouse over it. An event raised by a DHTML component bubbles upwards in the structural hierarchy of the document, calling event handlers as it goes along until it is cancelled or reaches the root. This means that different objects can share a single event handler, and that a single event can activate different handlers. To accomodate scripting languages that lack first class functions, all DHTML event handlers take no arguments and return no results. Instead they communicate via a global event object that contains the information about the most recent event.

As a simple example of scripting HTML, we will create two buttons that move a text up and down the screen. In the script, the function

```
item :: String -> IHTMLWindow -> IO IHTMLElement
```

returns an interface pointer to an HTML element whose `ID` attributes corresponds to the first argument of `item`.

```
<BUTTON ID="up">Up</BUTTON> <BUTTON ID="down">Down</BUTTON>
<DIV ID="it" STYLE="position:relative">
Push the buttons to move me up and down
</DIV>
<SCRIPT LANGUAGE="HaskellScript" SRC="....."></SCRIPT>
```

When the Internet Explorer renders this page and encounters the `SCRIPT` tag with `LANGUAGE` attribute `HaskellScript`, it will connect to the Haskell scripting engine, and execute the `main` function of the module referred to in the `SRC` attribute. Scripts can also be written directly between the `SCRIPT` tags.

The script first gets the interface pointers to the `"up"` and `"down"` buttons and the element that has to be moved. Then it installs two event handlers that will be called when user clicks on the buttons.

```
main =
  do{ window <- externRoot
    ; up <- window # item "up"; down <- window # item "down
    ; it <- window # item "it"
    ; up   # onClick_ (do{ it # changeTop (\y -> y+10)})
    ; down # onClick_ (do{ it # changeTop (\y -> y-10)})
    }
```

Function `changeTop` moves an HTML element by modyfing the `PixelTop` property of its `style` attribute:

```
changeTop :: (Int -> Int) -> IHTMLElement -> IO ()
changeTop = \f -> \element ->
  do{ style <- element # getStyle
    ; y     <- style # getPixelTop
    ; style # putPixelTop (f y)
    }
```

There exist many graphical libraries for Haskell [?,?,?]. The use of polymorphism and higher-order functions enables GUI programming at a high level of abstraction[?]. Since dynamic HTML is a full featured graphical language, we hope to develop a small library for programming DHTML too.

4.2 The Windows Scripting Host

The Windows Scripting Host (WSH) is a language-independent scripting host for Windows platforms. The WSH can invoke scripts from either the command line, or from the windows GUI. The object model of the WSH provides interfaces for doing user interaction, creating COM and ActiveX objects, as well as more system oriented interfaces for mapping network drives, connecting to printers, accessing environment variables, and manipulating the registry.

Many applications create a shortcut in the start menu when they are installed. The following script creates a shortcut to Hugs in the start menu. The scripts starts by creating an instance of the `Shell` component. Then it gets the path of the start menu from the `SpecialFolders` collection and creates a shortcut to hugs:

```
main =
  do{ wScript <- getWScript
    ; shell   <- wScript # createObject "WScript.Shell"
    ; start   <- shell # (specialFolders ## getItem "StartMenu")
    ; hugs    <- shell # createShortcut (start ++ "\\Hugs.lnk")
    ; with hugs
        [ setTargetPath hugsPath
        , setWindowStyle 4
        , save
        ]
```

In section 5.4 we will give a more elaborate example that uses both WSH and Internet explorer.

5 The Script Server

Instead of creating a Haskell specific implementation of the IActiveScript interface, we have made a language independent ActiveX script engine. This engine uses a much simpler set of interfaces, `IScript` and `IScriptItem`. Those interfaces represent the minimal functionality needed for a language in order to be programmed by an external application. Any language implementing those interfaces is called a *script server*. The advantage is that all the C++ code written to implement the fairly complex ActiveX scripting engine on top of the script server interfaces, can be reused. Only an implementation of the script server interfaces is needed to add ActiveX scripting to a language.

The script server interfaces are specified in IDL as:

```
[dual,oleautomation] interface IScript {
   HRESULT AddHostItem( [in] BSTR name, [in] IDispatch* item );
   HRESULT Load( [in] VARIANT* source );
   HRESULT Expr( [in] BSTR expr, [out] IScriptItem* item );
   HRESULT GetScriptItem( [in] BSTR name, [out] IScriptItem* item );
};

[dual,oleautomation] interface IScriptItem {
   HRESULT Eval( [in] int numArgs,
                 [in,sizeis(numArgs)] VARIANT* args,
                 [out] VARIANT* result );
};
```

A script is loaded with the `Load` method. The script can be specified with a file path but also for example with an `IStorage` object. This is used in the ActiveX script engine to load a script that resides in memory. The `AddHostItem` method makes external objects visible in the namespace of the script. The ActiveX script engine makes the root item visible with this method but it can also be used to add elements like buttons or forms from a Visual Basic program.

`GetScriptItem` returns elements defined in the script itself. The only method of an `IScriptItem` is `Eval` which takes an array of arguments and returns a single result.

The following Visual Basic program starts the Haskell script server and runs the `main` function of a script:

```
Dim script as IScript
Set script = CreateObject( "HugsScript" )
script.Load( "Test.hs" )
Set main = script.GetScriptItem( "main" )
main.Eval()
```

Since the script server is a COM component, a language like Java or C++ could also be used. Secondly, the script server interface can be implemented for any language; we are currently working on a script server implementation for the new Haskell run-time system and even for general dynamic link libraries (DLL's).

5.1 Exposing Haskell

Since Haskell is strongly typed, the Hugs interpreter doesn't need to maintain type information at run-time. The script server is therefore unable to check if a call to `Eval` is type correct. The script server simply passes the unprocessed arguments directly to Haskell where the function *itself* will check and marshal the arguments.

Any function that is called with `Eval` should have the abstract type `Export`. Values of that type can be created with the *export_n_m* function, where n is the number of arguments and m the number of results. These functions automatically marshal any parameters *and* check their types at run-time. For example:

```
export_1_0 :: (Variant a) => (a -> IO ()) -> Export
export_1_1 :: (Variant a,Variant b) => (a -> IO b) -> Export
```

Constructor classes [6] are used to automatically invoke the proper marshaling function. The `Variant` class is defined as:

```
class Variant a where
  inVar :: a -> Pointer VARIANT -> IO ()
  resVar :: Pointer VARIANT -> IO a

  inVarList :: [a] -> Pointer VARIANT -> IO ()
  resVarList :: Pointer VARIANT -> IO [a]
```

The `inVar` and `resVar` functions are used to marshal the arguments or results to COM world. The `List` functions are used to define custom marshaling for `String`'s[3].

Any type that is in the `Variant` class can be used as parameter. The types in the `Variant` class are exactly those types that are available to a JavaScript or Visual Basic programmer. A function like `getChar` is exported as:

```
exportGetChar :: Export
exportGetChar = export_0_1 getChar
```

A function like `length` can only be exported by constraining its (polymorphic) type to strings, which belong to the `Variant` class:

```
exportStrLen :: Export
exportStrLen  = export_1_1 strLen

strLen :: String -> Int
strLen        = length
```

Allthough we can write these definitions by hand, the H/Direct compiler is able to automatically generate export definitions from an IDL specification.

5.2 Hybrid applications: Visual Basic & Haskell

As an example of the script server, we'll create a hybrid program in Visual Basic and Haskell that computes the factorial of a number. Haskell is well suited to

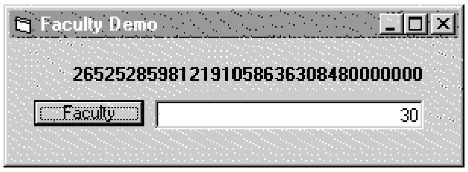

Fig. 4. The faculty of 30, computed using Visual Basic and Haskell

compute the factorial since it can use infinite precision integers to represent the possibly huge numbers. Visual Basic is well suited for programming the interface to the user since it has an excellent graphical environment.

The front end is drawn in Visual Basic with a text input field, a button and a text field in which the result is displayed. The program will create an instance of the Haskell script server and will invoke the Haskell function `exportFac` on a button click and update the result field accordingly.

```
Dim Script As IScript

Private Sub Form_Load()
Set Script = CreateObject("HugsScript")
Script.Load ("FacDemo.hs")
End Sub

Private Sub Button_Click()
Set Fac = Script.GetItem("exportFac")
OutputBox.Caption = Fac.Eval(InputBox.Text)
End Sub
```

The haskell program uses infinite precision `Integer` types to compute the factorial:

```
module FacDemo where
import HScript

exportFac :: Export
exportFac = export_1_1 (return . factorial)

factorial :: String -> String
factorial = show . fac . read

fac :: Int -> Integer
fac 0     = 1
fac n     = fromInt n * fac (n-1)
```

5.3 Event handling in Haskell

Just as we attached event handlers inside HTML pages, it is also possible to handle all events of the VB program in Haskell. All the application logic is moved to Haskell and VB is only used to draw the user interface. The VB program will create an instance of HugsScript, add the button, the input field and the result field to the script and run the `main` function.

```
Dim Script As IScript

Private Sub Form_Load()
  Set Script = CreateObject("HugsScript")
  Script.Load ("FacDemo2.hs")
  Script.AddItem "button", Button
  Script.AddItem "output", OutputBox
  Script.AddItem "input", InputBox
  Set main = Script.GetItem("main")
  Call main.Eval
End Sub
```

The Haskell script needs to import `MSForms` in order to script the Visual Basic components.

The `extern` function is used to access the added Visual Basic elements. Note that the `externRoot` funtion that was used in the HTML example is a special case of the `extern` function and could also be replaced with `extern "window"`.

Event handlers can be installed with the `onEvent` functions:

```
onEvent :: String -> IO () -> IO ()
onEventEx :: String -> Export -> IO ()
```

The `onClick_` method of the HTML example, is simply defined as `onEvent "onClick"`. H/Direct can automaticaly generate these "event" definitions from an IDL specification. The Haskell program installs an event handler for the button and returns. The `factorial` function is automatically called when the user presses the button.

```
module FacDemo where

import HScript
import MSForms

main :: IO ()
main = do { button <- extern "button"
          ; button # onEvent "Click" factorial
          ; return ()
```

```
          }

factorial :: IO ()
factorial = do { input  <- extern "input"
               ; output <- extern "output"
               ; text   <- input # getText
               ; output # setCaption (show (fac (read text)))
               }

fac :: Int -> Integer
fac 0   = 1
fac n   = fromInt n * fac (n-1)
```

The script server interface is used extensively in a front end to the Hawk system. Hawk is language to describe hardware circuits[5]. By using the Haskell script server, it is possible to *draw* hardware circuits in the Visio CAD/CAM application. The script server attaches events to the drawn Visio objects and automatically generates appropiate Hawk code.

5.4 Hybrid applications: DHTML as interface

In the previous example we took advantage of Visual Basic's powerful graphical environment to draw the front-end of the program. However, we still needed to explicitly program the script server interfaces; if the script would be embedded inside HTML, no such programming would be necessary.

HTML is a powerful language to describe a graphical user interfaces for which there are many powerful development environments. The logic of the application can be provided by scripts embedded inside the HTML page. Standard script languages, like JavaScript or VBScript, are not well suited for this purpose since their safety mechanisms makes their use rather limited. Haskell scripts can do anything on a host machine that normal applications can do. This makes Haskell less useful for general client-side scripting but much better suited for intranet scripts and multi-tier applications.

6 Conclusion

We have shown how Haskell can be used as a scripting language for COM components. Ten years ago, interaction with the outside world was difficult to achieve in a functional language. Now we can not only *use* a functional language to interact with standard mass-market components, but it is even *advantageous* to do so. Allthough their are many scripting languages around, the use of a strongly typed, higher-order language makes scripting considerably safer and easier.

Our next obvious step is to create COM/ActiveX components in Haskell. In Glasgow, the H/Direct compiler is extended to generate low-level COM objects which can be compiled. We are currently experimenting with *scriptlets*. A scriptlet is a combination of an embedded script and an XML description of its interface. A scriptlet will automatically be exposed as a COM object using the ActiveX scripting interfaces. DHTML can used as a drawing engine to add graphics and events to the scriptlet which is than exposed as an ActiveX component.

Acknowledgements

We would like to thank the PacSoft group at the Oregon Graduate Institute and in particular James Hook for their useful comments and examples. Daan leijen would also like to thank Sigbjorn Finne for his motivation to adapt the H/Direct compiler to our needs.

References

1. Kraig Brockschmidt. *Inside Ole (second edition)*. Microsoft Press, 1995.
2. David Chappel. *Understanding ActiveX and OLE*. Microsoft Press, 1996.
3. Erik Meijer, Daan Leijen and James Hook. Client side web scripting with HaskellScript. *submitted to PADL'99*, August 1998.
4. Mr. Foo. *Dynamic HTML*. Microsoft Press, 1998.
5. John Matthews, Byron Cook and John Launchbury. Microprocessor specification in Hawk. *ICCL, Chicago, Illinois*, May 1998.
6. 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.
7. Daan Leijen. Functional Components, Using COM components in Haskell. Master's thesis, University of Amsterdam, August 1998. `http://www.haskell.org/active/activehaskell.html`.
8. Microsoft. *The COM reference*. Microsoft Press, 1992.
9. Microsoft. *Programming Microsoft Agent*. Microsoft Press, 1997.
10. Dale Rogerson. *Inside COM*. Microsoft Press, 1997.
11. Sigbjorn Finne, Daan Leijen, Simon Peyton Jones and Erik Meijer. H/direct, a binary language interface for haskell. *International Conference on Functional Programming, Baltimore*, September 1998.
12. Simon Peyton-Jones, Erik Meijer and Daan Leijen. Scripting com components in haskell. *Fifth International Conference on Software Reuse, Victoria, BC, Canada*, June 1998.