

Client-side Web Scripting with HaskellScript

Erik Meijer, Daan Leijen, and James Hook

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

Oregon Graduate Institute
Pacific Software Research Center
hook@cse.ogi.edu

Abstract. Using client-side scripting it is possible to build interactive web pages that don't need round-trips to the server for every user-event. The web browser exposes itself to the script via an object model (DOM), which means that scripts can add and remove page content, or change the position and color of elements via their style attributes. We explain the object model as implemented by Microsoft Internet Explorer by means of examples and report on our experiences of using Haskell as a programming language for client-side web scripting using the *HaskellScript* scripting engine.

1 Introduction

Since the introduction of the web there has been a constant migration of computational power from the server to the client. Server-side scripting using CGI [12] provides critical interactive data entry capabilities and dynamically generated content, but in a very explicit, modal, turn-taking style of interaction. Client-side scripting extends the interaction paradigm by allowing content designers to specify reactive behaviors in their web pages in a much more modular and efficient fashion. For example, with client-side web scripting it is possible to locally make the appearance of an HTML element change when the mouse is positioned over the element, without the need of generating a completely new page on the server.

The reactive model exported by the browser to the scripting language interpreter is essentially a mechanism for registering call-back functions on events as well as access functions for the rich object structure of the browser and its contained HTML document. The use of a lazy, higher-order, typed language like Haskell [3] as a scripting language matches the event/call-back model very well because it supports the necessary low-level imperative features, but it also allows encapsulation and hiding of these gory details into powerful, reusable high-level abstractions.

The official W3C HTML 4.0 specification [15] allows for embedded scripts in a document, but it does not provide a standard object model that describes how scripts can manipulate the document content. Unfortunately, the Netscape and Microsoft browsers have incompatible DOM versions. The recently announced DOM Level 1 specification [4] defines a standard API, but it is not yet fully implemented in the standard browsers. In this paper we will therefore concentrate on the object model as implemented by Microsoft's Internet Explorer 4 [6], which in contrast to Navigator makes *all* objects in a document scriptable, and provides an elegant, language independent, open framework for plugging in non-standard scripting languages (section 5). We are currently working on an implementation of the full DOM Level 1 API on top of the IE5 object model.

The remainder of this paper is organized as follows. The next section (2) gives a short introduction to Haskell, in particular to the IO-monad and the `do{}` notation and gives references for further reading. The core paper starts in section 3 with a motivating example of the classic nervous text applet in Haskell. Internet Explorer exposes its object model as COM Automation objects. Section 4 first explains how Automation objects are invoked from Haskell and then continues by exploring the different objects that make up the DOM, garnished with illustrative examples. The underlying ActiveX-Scripting architecture is the subject of section 5, and we conclude with some reflexive remarks in section 6.

2 Minuscule introduction to Haskell

When interacting with the outside world or accessing object models we have to deal with side-effects. In Haskell [1], effectful computations live in the IO monad [7, 9, 17]. A value of type `IO a` is a *latently* effectfull computation that *when executed* will produce a value of type `a`. The command `getChar :: IO Char` will read a character from the standard input when it is executed. Like any other type, computations are first class citizens that can be passed as arguments, stored in list, and returned as results. For example `putChar :: Char -> IO ()` is a function that takes a character and then returns a computation that when executed will print that character on the standard output.

Commands can be composed using the `do{}`-notation. The compound command `do{ a <- ma; mf a } :: IO b` is a latent computation, that, when executed, first executes the command `ma :: IO a` to obtain a value `a :: a`, passes that to the action-producing function `mf :: a -> IO b` and executes `(mf a)` to obtain a value of type `b`. For example `do{ c <- getChar; putChar c }` is a command that when executed reads a character from the standard input and copies it to the standard output.

In other scripting languages such as JavaScript, commands are not first class values and are represented as code *strings*. The JavaScript `eval` method takes a code string argument, which is parsed and then executed. This is obviously a very primitive and error-prone way of composing effectful behaviour. It wrecks

all hope for a static type system, and even worse it loses static scoping since the code passed to the `eval` method is executed in the same context as the call to the `eval` method.

To make it patently obvious that we are dealing with effectful computations, we will wrap expressions of type `IO` inside a `do{}`. In the same vein, we will write values of functional type `a -> b` as lambda-expressions `\a -> b`. To accommodate OO-programmers, we will use postfix function application `object # method = method object` to mimic the `object.method` notation. These conventions result in highly stylized programs from which it is easy to tell the type of an expression by looking at its syntactical shape.

3 Nervous Text

A classic web page animation is the eye catching “nervous text” applet, which randomly jitters the characters of a sentence.

Ha^S kell isCool

To script this in Haskell, we define a HTML page with a `<DIV>` container¹ to hold the animated sentence. The container has an `ID="text"` attribute, so that we can refer to it in the script via the call `window # getItem "text"`. The element `<SCRIPT SRC="Nervous.hs" LANGUAGE="HaskellScript">` instructs the browser to load the HaskellScript engine and execute the main function of module `Nervous.hs`:

```
<STYLE>.a {position:relative}</STYLE>
<DIV ID="text" >Haskell is Cool</DIV>
<SCRIPT LANGUAGE="HaskellScript" SRC="Nervous.hs"></SCRIPT>
```

The main function of module `Nervous.hs` splits the content of `DIV` element into the individual letters and installs an event handler that bounces these around at every `interval` milliseconds (library functions like `getWindow`, `item`, and `setInterval` are explained in section 4).

```
main :: IO ()
main =
  do{ window <- getWindow
      ; division <- window # getItem "text"
      ; letters <- division # splitToLetters
      ; window # setInterval (do{letters # bounce}) interval
    }
```

¹ `DIV` abbreviates “division”. A document division is essentially a labeled block in HTML.

Function `splitLetters` wraps every character of its argument inner text into a `SPAN` element², inserts this as HTML, and finally returns the newly produced list of children:

```
splitToLetters :: IHTML_Element -> IO [IHTML_Element]
splitToLetters = \element ->
  do{ cs <- element # getInnerText
      ; element # setInnerHTML
        (unlines ["<SPAN CLASS=\"a\">"+[c]+"</SPAN>" | c <- cs])
      ; element # getChildren
    }
```

In conjunction with the `STYLE` element, the `CLASS="a"` attribute makes the position of all animated elements relative to their normally rendered position. The Haskell prelude function `unlines :: [String] -> String` turns a list of strings into a single string separated by newlines.

An `IHTML_Element` object can be moved around on a page by changing the `PixelLeft` and `PixelTop` properties of its `style` attribute:

```
getStyle :: IHTML_Element -> IO IStyle
setPixelLeft, setPixelTop :: Int -> IStyle -> IO ()
```

Function `move` combines these three functions into a compound method that is more convenient to use:

```
move :: (Int,Int) -> IHTML_Element -> IO ()
move = \(newLeft,newTop) -> \element ->
  do{ style <- element # getStyle
      ; style # setPixelLeft newLeft
      ; style # setPixelTop newTop
    }
```

Assuming a function `random :: IO Float` that returns a random number between 0.0 and 1.0, we can randomly reposition an element using `moveRandom`:

```
moveRandom :: IHTML_Element -> IO ()
moveRandom = \element ->
  do{ newLeft <- random; newTop <- random
      ; element # move (scale jump newLeft,scale jump newTop)
    }
```

Function `scale jump :: Float -> Int` scales the random number to an integer in the range `-jump` to `jump :: Int`.

Finally, function `bounce` maps `moveRandom` over a list of `IHTML_Elements` to create the groovy effect:

² The `SPAN` tag defines the scope of style attributes. In this case, every character has an independent set of style attributes.

```

bounce :: [IHTMLElement] -> IO ()
bounce = \elements ->
    forEach_ elements $ \element -> do{element # moveRandom}

```

The function `forEach_ :: [a] -> (a -> IO ()) -> IO ()` applies a monadic, side-effecting function to each element in a list. It can easily be defined in terms of the standard monadic map function `mapM_`.

That's all. A complete rocking and rolling web-page in just a few lines of plain Haskell.

4 The HTML Object Model

The HTML object model is an API through which all document content of a web page, including elements and attributes, is programmatically accessible and manipulable. This means that scripts can add, remove and change elements and/or tags and navigate through document structure at will. The visual rendering of HTML objects (such as font properties, colors, backgrounds, and box properties) is controlled by style sheets [10], which are fully scriptable as well.

All HTML elements are exposed to the Haskell programmer as COM Automation objects [8,2] of type `IDispatch d`. To enjoy a good cup of Java, you don't have to be an expert on growing beans. Similarly, to use `HaskellScript`, no knowledge of COM is required. Just a basic understanding of monadic IO is sufficient.

Automation objects are special COM components that implement the `IDispatch` interface [13]. An example Automation interface is `IHTMLLocation` whose IDL interface description looks like:

```

[uuid(3050F25E-98B5-11CF-BB82-00AA00BDCE0B)]
interface IHTMLStyle : IDispatch {
    ...;
    [propput] HRESULT color([in] VARIANT p);
    [propget] HRESULT color([out, retval] VARIANT* p);
    ...;
    HRESULT toString([out, retval] BSTR* String);
    ...;
}

```

The `color` property is of type `VARIANT` to allow different color representations like string color names such as "red" and hexadecimal RGB encodings such as #FF0000.

Arguments and results types of properties and methods of Automation interfaces are limited to a fixed set of `VARIANT` types. The type class `Variant` contains two member functions, the first one `marshal :: Variant a => a -> Pointer VARIANT -> IO ()` marshals a Haskell value into a `VARIANT` and the second

`unmarshal :: Pointer VARIANT -> IO a` unmarshals a `VARIANT` into its corresponding Haskell representation:

```
class Variant a where
  { marshal :: a -> Pointer VARIANT -> IO ()
  ; unmarshal :: Pointer VARIANT -> IO a
  }
```

The real `Variant` class has some additional member functions as it uses a trick due to Phil Wadler [16] to simulate overlapping instances akin to the `Read` and `Show` classes in Haskell.

We have written an IDL compiler called `H/Direct` [5] that generates Haskell bindings for COM components whose interfaces are specified in IDL. The compiler maps the properties and methods of the `IHTMLStyle` interface into corresponding Haskell functions that take an interface pointer to the `IStyle` as their last argument:

```
type IStyle = IDispatch ()

getColor :: (Variant a) => IStyle -> IO a
getColor = \c -> \style -> do{ style # get "color" }

setColor :: (Variant a) => a -> IStyle -> IO ()
setColor = \c -> \style -> do{ style # set "color" c }

toString :: IStyle -> IO String
toString = \style -> do{ selection # method_0_1 "toString" }
```

The (overloaded) functions `get` and `set` inspect the properties of an Automation object. They take the name of the property as a string argument:

```
set :: (Variant a) => String -> a -> IDispatch d -> IO ()
get :: (Variant a) => String -> IDispatch d -> IO a
```

The overloaded family of functions `methodnm` is used to call the methods of Automation objects from Haskell. Function `methodnm` maps an *n* `Variant` arguments to an *m*-tuple of `Variant` results:

$$\text{method}_{n,m} :: \overbrace{(\dots, \text{Variant } a, \dots)}^{n>0}, \overbrace{(\dots, \text{Variant } b, \dots)}^m \\ \Rightarrow \text{String} \rightarrow \dots \rightarrow a \rightarrow \dots \rightarrow \text{IDispatch } d \\ \rightarrow \text{IO } (\dots, b, \dots)$$

Internally, `get`, `set`, and `methodnm` are implemented using `marshal` and `unmarshal` to marshal and unmarshal arguments to and from `VARIANT`s and low level functions to call raw COM components. Note that `marshal` value is a function that when given a pointer, marshal value in the memory to which that pointer points.

```

set member = \value -> \obj ->
do{ dispid <- obj # getMemberID member
  ; p <- obj # primInvokeMethod dispPROPERTYSET False dispid
    [marshall value] []
  ; freeMemory p
}

```

All HTML elements can generate and handle events such as user events (e.g. mouse clicks), update events, and change events. Events bubble upwards from the source element to the root of the object tree and can be caught along the way. Handling events requires that the client of an Automation object (in this case the script written by the user) implements outgoing interfaces (event sinks, or callbacks) that will be called by the event source object when the event occurs. The HaskellScript library function `onEventn_m` installs an event sink for a given component:

$$\begin{aligned}
\text{onEvent}_{n_m} &:: \overbrace{(\dots, \text{Variant } a, \dots)}^{n>0}, \overbrace{(\dots, \text{Variant } b, \dots)}^m \\
&=> \text{String} \rightarrow (\dots \rightarrow a \rightarrow \dots \rightarrow \text{IO } (\dots, b, \dots)) \\
&\rightarrow \text{IDispatch} \rightarrow \text{IO } ()
\end{aligned}$$

In order to accommodate for scripting languages without first class functions, all HTML events have zero arguments and return no results. Instead, they use a global event object that contains information about the most recent event.

4.1 The IWindow object

The root of the HTML object model is the IWindow object, which represents the entire browser. Scripts can call `getWindow :: IO IWindow` to obtain a handle to their window object.

The IWindow object is quite rich, it has about 40 properties, 30 methods and can handle 10 different events. Amongst these are methods for standard GUI widgets such as `alert` that pops up an alert box, `confirm` that asks for confirmation, and `prompt` that ask the user to enter a string:

```

alert    :: String -> IWindow -> IO ()
confirm  :: String -> IWindow -> IO Bool
prompt   :: String -> String -> IWindow -> IO String

```

From IWindow, we can descend down the object tree using the following access functions

```

getHistory  :: IWindow -> IO IHistory
getNavigator :: IWindow -> IO INavigator
getLocation  :: IWindow -> IO ILocation

```

```

getScreen    :: IWindow -> IO IScreen
getEvent     :: IWindow -> IO IEvent
getDocument  :: IWindow -> IO IDocument

```

The `IHistory` object contains the list of URLs that the browser has visited; it is the programmatic interface to the “forward” and “backward” buttons in the IE toolbar.

The `INavigator` object contains information about the browser itself. JavaScript programs often use the `INavigator` object to distinguish between Internet Explorer and Netscape Navigator.

The `ILocation` object is the programmatic interface of the “address bar” of the browser. The various properties of the `ILocation` object dissect the currently loaded URL into its constituents (`hash`, `host`, `hostname`, `href`, `pathname`, `port`, `protocol`, and `search`).

The three `ILocation` object methods `assign`, `reload`, and `replace` control the currently displayed document. The `IScreen` object contains various properties of the physical screen, the most important of these are `width` and `height`.

The `IEvent` object is a ‘global variable’ that contains the parameters of all events as they occur. For instance, the `srcElement` property gives the `IHTMLElement` on which the last event occurred and there are various other properties like `screenX` and `screenY` from which the (relative) location of a button-click can be computed.

4.2 Scrolling status bar

The `IWindow` object has a `status` property which contains the text that is currently displayed in the status bar of the window. Many webpages have scrolling status bars to attract attention. The idea behind a scrolling statusbar is to pad the status text on the left with spaces, and then remove them one after the other at every clock tick.

```

msg = take 50 (repeat ' ') ++ "...

main =
  do{ window # setInterval (do{window # scroll}) scrollInterval }

scroll :: IWindow -> IO ()
scroll = \window ->
  do{ msg' <- window # getStatus
      ; window # setStatus (if msg=="" then msg else (tail msg'))
    }

```

4.3 The IDocument object

Much of the dynamism of dynamic HTML comes from the access to the window's document object. Through the IDocument object we can access the individual IHTML element objects that make up the page, but also the document's background color, and much much more. For example, the following script will change the background color to red when the user clicks anywhere on the document:

```
main :: IO ()
main =
  do{ window # getWindow
      ; document <- window # getDocument
      ; document # onClick (do{document # setBgColor "red"})
    }
```

The function getItem that returns a handle to a named element is in fact an abbreviation for a compound method that finds the element in the document's all collection:

```
Window.getItem :: String -> IWindow -> IO IHTML element
Window.getItem = \name -> \window ->
  do{ window # (getDocument ## getAll ## Document.getItem name) }
```

4.4 The IHTML element and IStyle objects

As we have seen in the nervous text example, we can change properties of the style property of IHTML elements to create interesting effects. Amongst the 75 properties of the style object are properties like

- `fontSize`, `fontStyle`, `fontVariant`, and `fontWeight` to change the font using which the parent IHTML element is rendered,
- `pixelLeft`, `pixelTop` and `zIndex` that control the element's position, and
- `visibility` and `display` that control the element's visibility.

Clicking on the button displayed by the following HTML, will cause the text within the SPAN element to disappear and reappear:

```
<BUTTON ID="toggle">Toggle</BUTTON>
<SPAN ID="text">Click the button to make me (dis)appear</SPAN>
```

The underlying script installs an event handler for the `onclick` event that will alternately set the `display` property of the style property of item "text":

```
main :: IO ()
main =
  do{ window <- getWindow
```

```

    ; text <- window # getItem "text"
    ; button <- window # getItem "toggle"
    ; button # onClick (do{text # toggle})
  }

toggle :: IHTMLElement -> IO ()
toggle = \element ->
  do{ style <- element # getStyle
      ; display <- style # getDisplay
      ; style # setDisplay (if display=="" then "none" else "")
    }

```

The textual representation of an `IHTMLElement` may be queried and modified through the four properties `innerHTML` and `outerHTML`, and `innerText` and `outerText`. The `innerHTML` property represents element without the enclosing tag of the element itself. Similarly the `innerText` property represents the contained HTML without any markup at all. The `outerHTML` and `outerText` properties are similar to `innerHTML` and `innerText` respectively except that they include the element's begin- and end-tags too. This makes it possible to remove or replace complete elements from a document completely.

As an example of using the `innerHTML` property, we will write a script to rotate between a list of HTML elements within a `SPAN` element [6]:

```

Create
<SPAN ID="word" words="[\<EM>amazing</EM>\",\<B>funky</B>\"]">
</SPAN>
webpages with HaskellScript!

```

The list of words is stored inside a custom attribute `words :: [String]`. For the programmer, there is no distinction between custom and built-in attributes:

```

setWords :: [String] -> IHTMLElement -> IO ()
setWords = \words -> \element ->
  do{ element # setAttribute "words" (show words) }

getWords :: IHTMLElement -> IO [String]
getWords = \element ->
  do{ words <- element # getAttribute "words"; readIO words }
  'catch' \_ -> do{ return [""] }

```

The call `readIO words` raises an exception if `words` cannot be parsed as a list of strings. If an exception occurs, the exception is caught and `getWords` returns a singleton list containing the empty string.

The main function for this example just installs an interval timer to rotate through the word list every `interval` milliseconds:

```

main :: IO ()
main =
  do{ window <- getWindow
      ; word <- window # getItem "word"
      ; window # setInterval (do{word # rotate}) rotateInterval
    }

rotate :: IHTMLElement -> IO ()
rotate = \element ->
  do{ (word:words) <- element # getWords
      ; element # setInnerHTML word
      ; element # setWords (words++[word])
    }

```

5 ActiveX Scripting, the enabling technology

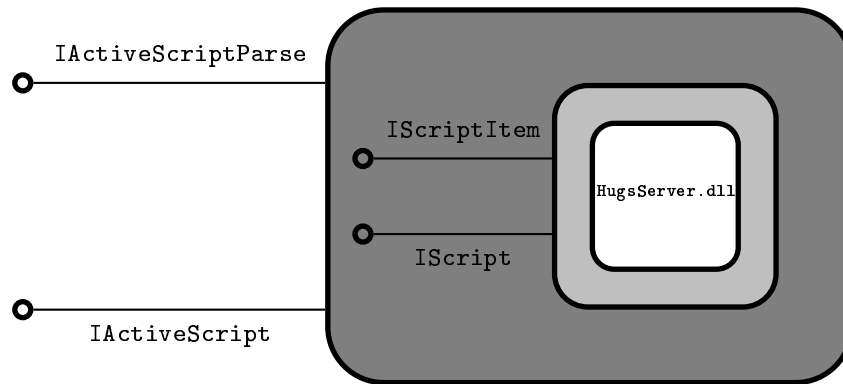
The browser needs to communicate with the scripting language's execution engine in order to execute the scripts it encounters in the web page. The ActiveX Scripting architecture [2] is a language-independent protocol that enables embedded scripts in a variety of scripting host applications, including the Internet Explorer browser, the IIS web server and Windows itself. If a scripting host wants to execute a script, it will look if there is scripting engine available for the particular language in which the script is written. If so, it will create an instance of the engine, send it the script code, make its root objects available, and run the script. For the browser the root object is the `IWindow` object.

The ActiveX scripting framework is language- and application-independent. Any application that implements the `IActiveScriptHost` interface can be scripted in any language for which the `IActiveScript` interface is available. Traditionally most scriptable applications such as CAD-tools, editors, and workflow-tools have hardwired, application specific, internal scripting languages. This means that we cannot leverage our knowledge of say vi-macros when we use Emacs or \TeX . ActiveX scripting makes it possible to use a single scripting language for a wide range of applications, or many scripting languages within a single application.

To implement an ActiveX scripting engine, we have to create a COM object that supports the `IActiveScript` and `IActiveScriptParse` interfaces. The `IActiveScript` interface provides the methods necessary to initialize the scripting engine. One of its most important methods is `AddNamedItem`, which is called by the host to inject objects into the engine's namespace, and `Close`, which causes the scripting engine to shut down. The `IActiveScriptParse` interface provides the ability to load scripts and evaluate expressions, primary via the `ParseScriptText` method.

Instead directly implementing complete versions of the `IActiveScript` and `IActiveScriptParse` interfaces on top of the Yale/Nottingham Hugs inter-

preter, we have made a generic script engine that uses a much simpler set of interfaces than required by the full ActiveX Scripting framework. Our `IScript` and `IScriptItem` interfaces represent the minimal functionality needed for a language in order to be used as a scripting engine.



The benefit of this layering is that all the C++ adapter code that implements the Active Scripting interfaces on top of the primitive script server interfaces can be reused to implement a scripting engine for any other language (for example Standard ML), or Haskell interpreters other than Hugs. This is considerably simpler than implementing the ActiveX scripting engine interfaces from scratch.

An additional advantage is that the script server interfaces are easy to use from ordinary, non scripting host, applications. This makes it possible have a very close interoperation between languages like Visual Basic or Delphi with good graphical development environments, or applications like Microsoft Office or Visio that have hardwired scripting languages but that can use Automation objects. The Hawk group at OGI has taken this route to implement a Visio-based GUI for their functional hardware description language [11].

The IDL specification for our script server interfaces is:

```
[oleautomation,dual]
interface IScript : IDispatch {
    HRESULT AddItem([in]BSTR itemName, [in]IDispatch* item);
    HRESULT Load([in]VARIANT* scriptSource);
    HRESULT GetItem([in]BSTR name, [out,retval]IScriptItem** item);
};
```

```
[oleautomation,dual]
interface IScriptItem : IDispatch {
    HRESULT Eval( [in] int cArgs,
                 , [in,size_is(cArgs)] VARIANT* args,
                 , [out,retval] VARIANT* result
                 );
};
```

A host loads a script by calling `Load`. The script's source can be specified with a file name but also with an `IStorage` object, which is used by the ActiveX scripting engine to load a script that resides in memory. When a module is loaded, its `main` function is evaluated if present. The `AddItem` method injects host objects into the script's context. The browser uses it (indirectly) to make the `window` object available, and a VB program could use it to make a button or form visible to the script. In the script, objects added by `AddItem` can be retrieved using function `getHostItem :: String -> IO (IDispatch a)`.

In the other direction, hosts can use `GetItem` to get access to variables, objects and functions that exported from the script. To the host these Haskell values are exposed via the `IScriptItem` interface, whose single method is `Eval`. The signature of `Eval` is carefully chosen to match Visual Basic's [14] `DISPPARAMS` structure [13]. Therefore, we can use normal function call syntax for `Eval` in VB, and the resulting outgoing call will have exactly the three arguments as demanded in the `IScriptItem` interface. When using C or C++ we have to pass the three arguments explicitly. On the Haskell side, dynamic values can be exported by the `addScriptItem` family of functions:

$$\begin{aligned} \text{addScriptItem}_{n,m} &:: \overbrace{(\dots, \text{Variant } a, \dots)}^{n>0}, \overbrace{(\dots, \text{Variant } b, \dots)}^m \\ &=> \text{String} \\ &-> (\dots -> a -> \dots -> \text{IO } (\dots, b, \dots)) \\ &-> \text{IO } () \end{aligned}$$

Note the similarity between `addScriptItem` and `onEvent`, both of which make a Haskell value callable by the external world.

The following Visual Basic example loads the Haskell module `Fac.hs` and then computes a factorial in Haskell:

```
Dim Script As IScript
Set Script = CreateObject("HugsScript")
Script.Load("Fac.hs")
Set Fac = Script.GetItem("Fac")
MsgBox (Fac.Eval(1000))
```

The Haskell module `Fac.hs` exports the standard tail-recursive factorial under the name `"Fac"`, assuming that the type `Integer` of arbitrary precision integers is an instance of the `Variant` class.

```
main :: IO ()
main = do{ addScriptItem_1_1 "Fac" fac }

fac :: Integer -> Integer
fac =
  let{ fac m = \n ->
        case n of { 0 -> m; n -> fac (m*n) (n-1) }
      } in (\n -> fac 1 n)
```

6 Conclusions

Most computer scientists dismiss declarative languages as naive, elegant curiosities that are largely irrelevant to the work-a-day world of programming. One of the primary reasons is that functional and logic languages have often lived in a closed world, or, at best, had awkward interfaces to the outside world that required significant investment to develop.

In our research program on the use of Haskell for scripting, we are addressing these challenges directly. First, we are exploiting the current popularity of COM and its Interface Description Language to get language independent descriptions of foreign functions with a minimal investment [5].

Second, we are exploiting the structure of the foreign functionality captured in the HTML object model. While there are some significant warts, for instance zero-argument callback functions and the fact that all interfaces final, it is a useful structure and it supports a reasonable type-safe, natural coding style at the lowest level. In our prior experience it was always necessary to encapsulate foreign functionality in a carefully crafted layer of safe abstractions. In this context it is possible to use the raw, imported functionality directly. As we begin to explore other tools with automation interfaces (such as Visio and Microsoft Office applications) we are encouraged to find that “good object models” from an OO point of view are giving rise to usable automatically generated interfaces.

Third, we are reaping the benefits of explicitly monadic functional programming by passing “latently effectfull computations” as values to event handlers and other functions. Ultimately scripting languages are IO behaviors. By having IO behaviors as first-class values, and not just the side effect of evaluation, we are able to support a compositional style of abstraction that is not possible in traditional scripting languages. This benefit of using Haskell as a scripting language is featured in earlier work, in which we develop parallel and sequential combinators for animation behaviors of Microsoft Agents [8].

All these factors converge to make Haskell an excellent language for scripting in general, and scripting Dynamic HTML in particular. However, many challenges remain. We are currently working to improve the robustness of the implementation and to develop a greater body of experience in scripting HTML and other tools with this technology.

HaskellScript is available at the Haskell homepage: <http://www.haskell.org>.

Acknowledgments

Erik Meijer much appreciated the hospitality of the PacSoft group at OGI and the Computer Sciences Research Center at Bell Labs in the initial stages of writing this paper. We all thank Conal Elliott for his extensive comments, and the PADL referees for their apt remarks and constructive suggestions for improvement. Unfortunately, lack of space prevented us from fitting their bill completely.

References

1. Richard Bird. *Introduction to Functional Programming using Haskell (2nd edition)*. Prentice Hall, 1998.
2. David Chappel. *Understanding ActiveX and OLE*. Microsoft Press, 1996.
3. J. Peterson (editor) et. al. Report on the programming language Haskell version 1.4. <http://www.haskell.org/>, April 6 1997.
4. Lauren Wood et. al. Document Object Model (DOM) Level 1 Specification. <http://www.w3.org/TR/REC-DOM-Level-1>, October 1998.
5. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: A Binary Foreign Language Interface to Haskell. In *Proceedings of ICFP'98*, 1998.
6. Scott Isaacs. *Inside Dynamic HTML*. Microsoft Press, 1997.
7. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL 20*, pages 71–84, 1993.
8. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of ICSR5*, 1998.
9. SL Peyton Jones and J Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.
10. Håkon Wium Lie and Bert Bos, editors. *Cascading Style Sheets*. Addison-Wesley, 1997.
11. John Matthews, John Launchbury, and Byron Cook. Microprocessor Specification in Hawk. In *International Conference on Computer Languages*, 1998.
12. Erik Meijer. Server-side Scripting in Haskell. *Journal of Functional Programming*, Accepted for publication.
13. Microsoft Press. *Automation Programmers Reference*, 1997.
14. Microsoft Press. *Visual Basic 5 Reference Manual*, 1997.
15. D. Ragget, Arnoud Le Hors, and Ian Jacobs. HTML 4.0 specification. <http://www.w3.org/TR/REC-html40>, December 1997.
16. Philip Wadler. Personal communication.
17. Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.