Oberon Script: A Lightweight Compiler and Runtime System for the Web

Ralph Sommerer

Microsoft Research, 7 J J Thomson Avenue,
Cambridge, United Kingdom
som@microsoft.com

Abstract. Oberon Script is an experimental scripting language and runtime system for building interactive Web Client applications. It is based on the Oberon programming language and consists of a compiler that translates Oberon Script at load-time into JavaScript code, and a small runtime system that detects and compiles script sections written in Oberon Script.

1 Introduction

Oberon is the name of a modular, extensible operating system for single user workstations [19], and also of an object-oriented programming language specifically developed to implement the former [17]. Although originally designed as the native operating system for custom built workstations, Oberon was subsequently ported to various different computing platforms including personal computers [2][4] and Unix workstations [1][14][15].

With the recent emergence and proliferation of sophisticated Web client applications, the *Web browser* has become a computing platform on its own. It offers the Web application programmer scripting facilities based on the JavaScript language [3] to programmatically interact with a Web server, and to manipulate the Web page in-place and without reloading. It thus allows the construction of rich Web application user interfaces that are not limited to the page-based hypertext model anymore and approach those of desktop applications.

As the Web browser morphs into a runtime system and operating platform for Web client applications, the question arises whether it can provide a suitable target platform for another installment of Oberon, especially in light of all previous porting efforts that have shown Oberon's demands of the host platform to be very limited. While attempting to answer this question we can explore in particular the suitability of JavaScript as a "portable object code" to compile Oberon to, and the feasibility of performing the compilation online, i.e. on the browser itself. Oberon promises to strike the right balance between being simple enough to make this experiment feasible and powerful enough to make it meaningful.

In this paper we present *Oberon Script*, an experimental effort to develop a simple and lightweight application programming framework for building complex Web client applications in Oberon. The system consists of a load-time Oberon-to-JavaScript compiler and a small runtime system to process and run script sections written in Oberon Script.

2 Web Client Programming

The page based hypertext model of the Web is unsuitable for rich Web applications user interfaces because the unit of interaction – the execution of a link and the corresponding loading of a new page even for simple interactions – is too coarse to provide a smooth and pleasant user experience. Simple interactions such as attaching a file to an email message in a Web based email client require as many as 3 page loads. Recently, however, Script-based Web applications have started to emerge, that employ a so-called Ajax-style of application design. Ajax stands for *Asynchronous JavaScript and XML* [10]. In applications built using these techniques the page is modified on-the-fly by programs written in browser-run scripting languages, thus avoiding the reloading of the page even for complex user activities or display updates. This application style was popularized by Google through their e-mail [7] and mapping [8] services, although neither was pioneering in relying on Ajax techniques.

2.1 Ajax

The Ajax-style of Web application programming is usually recognized by the use of the following techniques: HTML DOM [16] manipulation via client-side scripting languages, mainly JavaScript [3], and the use of XML as the data exchange format between server and client. The core foundation of Ajax, however, is a built-in browser component called *XMLHttpRequest* [10] that allows JavaScript code to interact with a Web server "behind the scenes" and without having to reload the page. The use of XML is not essential, and other data formats are commonly employed, including plain text or a linearization of JavaScript objects (JSON) [12].

2.2 JavaScript

JavaScript is an object-based scripting language for the Web. Originally developed under the name of *LiveScript* it was later re-branded as JavaScript because of its superficial syntactical similarities with the programming language Java [9], but also in order to benefit from the publicity around the then new language. JavaScript is now standardized as ECMAScript [3], and all modern Web browsers support the language using different brand names, such as JScript or JavaScript.

JavaScript does not support classes. Instead, it supports a prototype-based inheritance model with shared properties (fields and methods). Objects are created using a *constructor function* that initializes the object's instance variables. Fields and methods that were defined via the constructor function's *prototype* property are subsequently available as instance fields and methods.

JavaScript objects are implemented as hash tables, and instance fields are stored as entries in those tables. The following ways of accessing instance fields are therefore interchangeable: obj.field (field access), obj[field] (hash table access).

The JavaScript runtime system also features a small collection of predefined objects such as strings, arrays, regular expression objects, and so on, some of which also have a correspondence in the language (e.g. string constants in the language are instances of the *String* object).

3 Oberon Script

3.1 Language

Oberon Script is a subset of the Oberon programming language as defined in [17]. "Subset" is to be understood not so much with respect to *language* as to *semantics*. Indeed, the Oberon Script compiler compiles the *full language* as specified in the language report referenced above, i.e. the language Oberon and some of the additions introduced by *Oberon-2* [13]. However, for reasons of simplicity and compactness, and also to be compatible with the underlying runtime system that is based on JavaScript, some of the rules are relaxed. Thus, some of what would be syntactical errors in Oberon is permissible in Oberon Script.

The decision to support the full language was based chiefly on the following principles: First, we consider an effort to port a language to a new computing environment to be incomplete as long as the full language is not supported. Changing the language to simplify its porting is tantamount to adjusting a question to fit an answer. Problems encountered during such an endeavor should be regarded as challenges, and not opportunities to shortcut. Dropping or adjusting features later for purposes of optimization or simplicity are acceptable but only once the system has proved working. Second, we believe the Oberon language to be sufficiently concise such that stripping it down any further will likely harm its expressiveness. The language report specifying the syntax and semantics of Oberon is one of the shortest around (28 pages). The JavaScript language specification, in comparison, covers 188 pages [3]. Third, by basing our experiment on the full language Oberon instead of a cut-down toy language we can assess more accurately the limits of a language's complexity that can be reasonably compiled and processed in the browser on-the-fly.

3.2 Compiler

The Oberon Script compiler is a simple one-pass recursive-descent parser [18] that performs very basic syntax analyses and emits JavaScript constructs as a side-effect. Manual translation of Oberon constructs into JavaScript revealed that many features and constructs of the former have a structure that is very similar to those in the latter. For example, designators, expressions, statements, and control structures look basically the same in both languages, apart from trivial differences such as the symbols used to express them. This similarity suggests employing regular expressions to translate Oberon's syntax into that of JavaScript. However, after some initial experiments we decided against it. Apart from very simple expressions, most syntactical elements require the translator to have a certain minimal understanding of their structure in order to translate them into correct JavaScript. For instance, a simple designator, such as a local variable, can be discovered using regular expressions, but a moderately complex one, e.g. one involving arrays, type tests, or even a combination of these, requires at least *some* (recursive) parsing to establish its extent. But if *some* parsing is required in any case for any moderately complex program, it stands to reason that we can as well parse the *whole* program.

While the syntactical differences of Oberon with the resulting JavaScript code are too big to allow using regular expressions to translate one into the other, they are small enough to greatly simplify the compiler. For example, in many places it is only necessary to identify syntactical *patterns* instead of their details. The same parsing routine can therefore be employed in different places in our compiler where the different semantics of such constructs would require different routines in a regular compiler. Consider for example the following syntactical constructs:

```
FieldList = [IdentList ":" type].
VariableDeclaration = IdentList ":" type.
FPSection = [VAR] ident {"," ident} ":" FormalType.

IdentList = identdef {"," identdef}.
identdef = ident ["*"].
```

Although it is obvious that field lists (of record type declarations), variable declarations, or formal parameter sections (FPSection) are different syntactical constructs and require different processing in a regular compiler (such as different allocation methods), for our purposes they are simply lists of identifiers followed by a type. Their different processing requirements can easily be accommodated for by passing an appropriate handler method, but the compiler doesn't need to parse them differently. A single parser method thus suffices for all three.

For reasons of simplicity and compactness of the compiler – and interoperability with regular JavaScript – only very minimal semantics analyses are performed, and only where it is necessary to establish a certain condition in order to proceed with the parsing. Designators, for example, are fully developed, including the type of the current selector, in order to determine certain features of the designated object, e.g. to distinguish procedure calls from type tests, or to handle reference parameters correctly. Expressions, as a counter example, are not developed at all, and are simply output to the JavaScript generator. Therefore, a standard procedure call such as the following (where s is a string variable):

```
INC(s, "hello world")
```

which is illegal in Oberon, is not only permissible in Oberon Script, its translation in JavaScript actually makes perfect sense:

```
s+="hello world" //concatenation
```

3.2.1 Modules

Oberon modules can be described in object-oriented terms as singleton objects [6], with static fields and methods representing the global variables and procedures. This is also the approach used in Oberon Script to implement modules.

An Oberon Script module is translated into a JavaScript object constructor function bearing the name of the module. In the body of that function, all exported items, including (record) types, constants, variables, and procedures are assigned as static members of the function object. They can thus be accessed from the "outside" (other Oberon Script modules or regular JavaScript) using the familiar "dotted" qualified identifier notation consisting of the module name and that of the object in the form *Module.Object*.

Example of an Oberon Script module and its representation as JavaScript object.

```
(*Oberon Script module*)
MODULE Mod;

CONST
  N*=1024;

TYPE
  Point*=RECORD x,y:INTEGER END;

VAR
  pt*, pt0:Point;

PROCEDURE Move*(dx,dy:INTEGER);
BEGIN INC(pt.x,dx); INC(pt.y,dy)
END Move;

PROCEDURE SetOrg*(x,y:INTEGER);
BEGIN pt0.x := x; pt0.y := y
END SetOrg;

BEGIN pt0.x := 0; pt0.y := 0; pt := pt0
END Mod.
```

```
//JavaScript translation
function Mod
{
  Mod.N=1024;
  Mod.Point=function(){this.x=0;this.y=0}
  Mod.pt=new Mod.Point();
  var pt0=new Mod.Point();
  Mod.Move=function(dx,dy){pt.x+=dx;pt.y+=dy}
  Mod.SetOrg=function(x,y){pt0.x=x;pt0.y=y}
  pt0.x=0;
  pt0.y=0;
  _cpy(pt,pt0); //value copy
}
Mod(); //execute body
```

Non-exported objects (variables, types and procedures) are translated as local functions and/or variables in the body of the constructor function that represents the module. Note that this use of local objects (variables, functions) as "private global" objects is perfectly legal in JavaScript, and possible due to its *execution contexts* in which a local function can reference objects of an outer scope and keep them alive even if their containing scope dies. The global variable *pt0* the example above is referenced in the exported (hence static) procedure *SetOrg* and thus kept alive even if the body of the function *Mod* terminates. If *SetOrg* were not exported both it and the global variable *pt0* would disappear (i.e. be garbage collected) when *Mod* terminates. However, this is perfectly valid, since objects that are not referenced need not be kept alive, irrespective of whether they are dynamic data structures, or functions and global variables.

3.2.2 Record Types

JavaScript distinguishes only a few type *classes* (e.g. numbers, objects, and strings), but doesn't support *types*. Objects in JavaScript are considered compatible if they support the same fields.

An Oberon record type is represented in JavaScript by a constructor function that initializes the record's fields and thus renders it "compatible" with one of equal or extended type. The identity of the type (as opposed to its compatibility) is only required for type tests. It is represented by a (static) array of constructor functions that encodes the record's extension hierarchy. The constructor function also gets as part of its *prototype* properties (remember that those are shared by all instances of the object) a base-type initializer function and a type check function that implements the IS operator. Those features are assigned to the constructor function by a runtime *extension initializer* function called *_ext.*

```
TYPE
  R0=RECORD x,y: INTEGER END;
  R1=RECORD(R0) b:BOOLEAN END;

VAR
  r:RECORD(R0)k:INTEGER END;
  r1:R1;

function R0(){this.x=0;this.y=0}
_ext(R0);

function R1(){this._b();this.b=false}
_ext(R1,R0);

var r=new function(){this._b=R0;this._b();this.k=0}();
var r1=new R1();
```

The example above illustrates a named record type declaration, a named type extension and an anonymous record declaration. The field *_b* holds the base-type initializer. In the example above the value of *_b* in *R1* is *R0*, and will initialize the inherited fields *x* and *y* of *R1*. In multi-level extensions, the corresponding base-type initializer call will cascade through all levels until all fields are initialized.

The anonymous record type (3rd example above) does not get an extension list because it cannot appear on the right-hand side of a type test (left-hand side appearances can be checked by the compiler). Therefore, the extension initializer *_ext* is not called for the record type, and the base-type initializer *_b* needs to be assigned in-place before it can be called.

As a consequence of records being JavaScript objects special care is required to handle record assignments correctly. Assignments to record variables and value parameters require copying the record contents (recursively if necessary). A generic runtime function is provided for that purpose. It copies all fields of the source record for which there is a correspondence in the target record, by enumerating all target field names and then using them to copy the corresponding source values to the respective target fields. This is not the most efficient way of handling record assignments, but record value assignments are relatively rare in Oberon. For reference parameters (see below) passing the pointer of the record object is sufficient.

3.2.3 Reference Parameters

In Oberon, reference parameters (*var parameters* in Oberon lingo) allow addresses of variables to be passed to functions instead of their values. This usually serves one of two purposes: either to return structured and/or multiple values from functions (return values are scalar in Oberon), or to pass complex sizeable structures to functions even if there's no intention to modify (any of) their values, in order to save the computing effort of copying the structures onto the argument stack. In JavaScript arguments are always passed to functions by value.

In contrast to their conceptual simplicity, implementing reference parameters in an environment that does not support them natively often requires a disproportionate effort to handle them correctly under any circumstances [11]. The reason is the rare but non-negligible possibility of *aliasing*, i.e. the possibility that the variable (memory location) referenced using a reference parameter might be changed using a different designator. For instance, a field of a record might be passed as a reference parameter to a function that later overwrites the complete record (and hence also the field). Although such aliasing effects are rare, they need to be provided for because they are almost impossible to detect by the compiler.

JavaScript offers a relatively simple way to simulate passing a variable instead of its value to a function, but care has to be taken that the passed value behaves correctly under possible aliasing effects. The basic idea is to pass an *execution context* as the actual reference parameter to the function rather than the *value*. The execution context is that of an anonymous function defined in-line, that contains a reference to the variable, such that all modifications prompted through the execution context affect the original variable. Assuming the following declarations in Oberon Script and a call to procedure *P*:

```
PROCEDURE P(VAR x:INTEGER);

VAR k: INTEGER;
...
P(k);  //procedure call
```

The translation to JavaScript looks as follows:

```
function P(x) {...}
...
var k=0;
P(function(v){return(v?k=v:k)});
```

Note that the body of the function passed to *P* in above example operates on the *k* of the outer, i.e. calling scope. If the passed function is called without an argument, it returns the value of k, and if it's called *with* an argument it sets the value of k. For all scalar values (including pointers) above solution is resistant to aliasing effects.

The situation is a bit more involved for complex designators denoting instance fields, values accessed via pointers, and arrays. In these cases the "access path" to the variable must be evaluated like in a regular compiler to determine the "final" variable that is passed to the function by reference. To use the technique introduced above the variable must be referenced in the execution context. To avoid passing a copy instead of the variable itself, the last selector must be evaluated in the execution context. In case of arrays, this means that the last array dimension must be evaluated in the execution context using a cached index expression. The following code segment illustrates passing arguments by reference using multi-selector designators. The three situations shown are the following: (1) a pointer dereferencing chain, (2) a field of a multidimensional array of records, and (3) an element of a multidimensional array. They are based on the type declarations below. The examples list alternately the call in Oberon and then the translation in JavaScript.

```
TYPE
  PR=POINTER TO R;
  R=RECORD k:INTEGER; ptr:PR END;
VAR
  ptr:PR;
  a:ARRAY N,N,N OF R;
  b:ARRAY N,N,N OF INTEGER;

P(ptr.ptr.ptr.k);      // Oberon (1)

var _0= ptr.ptr.ptr;  //JavaScript (1)
P(function(v){return(v?:_0.k=v:_0.k});

P(a[i,j,k].k);         // Oberon (2)

var _0= a[i][j][k];     //JavaScript (2)
P(function(v){return(v?:_0.k=v: _0.k});

P(b[i,j,k]);           // Oberon (3)

var _0= b[i][j];_1=k; //JavaScript (3)
P(function(v){return(v?:_0[_1]=v: _0[_1]});
```

From the discussion above it is obvious that the complexity of handling reference parameters can hardly be justified in light of the simplicity of the original concept. Reference parameters are therefore likely candidates for being discarded if an effort to simplify Oberon Script is ever considered. Structured return values could provide an alternative to reference parameters that are far simpler to realize in JavaScript.

3.2.4 Code Quality

The compiler is effectively a syntax translator that transforms code written in Oberon into equivalent JavaScript code. It specifically does *not* emit JavaScript that resembles artificial “assembly code”. Therefore, the resulting code carries no significant runtime overhead compared to equivalent manually written JavaScript (disregarding the different “styles” of programming in the different languages). Furthermore, the most salient transformations required when compiling Oberon to JavaScript are those that deal with *declarations*, especially those that have no counterparts in JavaScript (modules, records). These incur only an insignificant execution overhead. With regard to *statements* there is more or less a one to one correspondence of Oberon’s features to those of JavaScript. Their respective execution times are therefore equivalent. The most significant additional execution costs can be expected for features not present natively in JavaScript that therefore need to be simulated with extra code. These include type tests, reference parameters, and local record variables which must be allocated each time a procedure is entered.

3.3 Runtime System

The runtime system consists of the above Oberon-to-JavaScript compiler and a small set of utility functions that includes JavaScript and DOM bindings, and a facility that detects script sections written in Oberon Script and subjects them to the compilation process.

Oberon Script is activated on a Web page by specifying in the header section of the page a link to the Oberon runtime scripts using a `<script>` element, and a call to *Oberon.Init()* in the *onload* event handler of the Web page body. As part of the initialization process, the runtime system identifies all code sections that contain Oberon Script. These need to be specified using the type attribute on the `<script>` element. Oberon Script is specified by the experimental MIME [5] type of "text/x-oberonscript". The runtime system then extracts the code from these sections and compiles them one after the other using the compiler, resulting in a collection of JavaScript sections. The compiler then replaces the original `<script>` elements containing Oberon Script code with new ones containing the compiled JavaScript code. Control is then passed to the compiled code. The following code illustrates the core of the Oberon Script detector and compiler.

```
function findLang(scp,typ)
{
  var code=[];
  for(var i=0;i<scp.length;++i){
    if(scp[i].type.toLowerCase()==typ){
      code.push(scp[i].text)
    }
  }
  return code
}

function addScript(par,code)
{
  var scp=document.createElement("script");
  scp.text=code;
  par.appendChild(scp)//this will also execute the code
}

function compileAll(typ,compile)
{
  var scp=document.getElementsByTagName("script");
  if(scp.length>0){
    var par=scp[0].parentNode;
    var code=findLang(scp,typ);
    for(var i=0;i<code.length){
      var res=compile(code[i]);
      if(res)addScript(par,res)//else error
    }
  }
}

compileAll("text/x-oberonscript",Oberon.Compile);
```

Although the compiler is usually not needed after it has finished compiling all Oberon Script sections, it stays around, in case further Oberon Script is created programmatically, and then compiled and executed on-the-fly.

4 Summary and Conclusions

In this paper we have presented an experimental runtime system called *Oberon Script* for using Oberon as a scripting language in the Web environment. It consists of an Oberon Script detector and a simple compiler that translates Oberon into JavaScript as a portable runtime code. We have shown that it is possible to process and compile the *full language* albeit with some effort to handle the few features in Oberon that are difficult to port without native support such as its reference parameters.

For a scripting language it is acceptable to sacrifice some of the parent language's features to simplify its implementation. Supporting the full language, however, makes it possible in theory to port the whole Oberon *system* to the browser, thus turning the latter into a virtual machine. How difficult it is to accomplish this task – and whether it is sensible to attempt it in the first place – needs to remain the subject of further study.

The current version of the Oberon Script compiler which is not optimized for efficiency or code size consists of 1081 lines of JavaScript code (24452 bytes). On a personal computer equipped with a 1.2 GHz CPU and 512 Mbytes of RAM it compiles an Oberon module of 268 lines (7933 bytes) in 783 ms (average of 10 runs).

References

1. Brandis, M., Crelier, R., Franz, M., Templ, J.: The Oberon System Family. Tech. Report ETH 174, (1992)
2. Disteli, A. R.: Oberon for PC on an MS DOS Base. Tech. Report ETH 203, (1993)
3. ECMA International, ECMAScript Language Specification, Standard ECMA-262, 3rd ed. (1999)
4. Franz, M.: Emulating an Operating System on Top of Another. Software - Practice and Experience, Vol. 23:6, 677-692, (1993)
5. Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, RFC 2046 (1996)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1994)
7. Google Mail, http://gmail.google.com
8. Google Maps, http://maps.google.com
9. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edition. Addison-Wesley (2005)
10. Garrett, J. J.: Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/ublications/essays/archives/000385.php
11. Gough, K. J., Courney, D.: Evaluating the Java Virtual Machine as a Target for Languages Other Than Java. Proc. Joint Modular Languages Conf. (JMLC 2000), Zurich, Switzerland. Lecture Notes in Computer Science Vol. 1897, 278-290, Springer (2000)
12. JavaScript Object Notation (JSON), http://www.json.org

13. Mössenböck, H., Wirth N.: The Programming Language Oberon-2. Structured Programming, Vol. 12:4, 179-196. (1991)
14. Supcik, J.: HP-Oberon (TM). The Oberon Implementation for HP 9000 Series 700. Tech. Report ETH 212, (1994)
15. Templ, J.: Design and Implementation of SPARC-Oberon. Structured Programming, Vol. 12, 197-205. (1991)
16. W3C: Document Object Model (DOM). http://www.w3.org/DOM/
17. Wirth, N.: The Programming Language Oberon. Software - Practice and Experience, Vol. 18, 671-690. Springer-Verlag, Berlin Heidelberg New York (1989)
18. Wirth, N.: Compiler Construction. Addison-Wesley (1996)
19. Wirth, N., Gutknecht, J.: The Oberon System. Software - Practice and Experience, Vol. 19, 857-893. Springer-Verlag, Berlin Heidelberg New York (1989)