The Revised Oberon-2 programming language

Andreas Pirklbauer

1.4.2019

Revised Oberon-2 is a revision of the programming language Oberon-2[1]. The main difference to the original is that it is based on *Revised Oberon (Oberon-07)* as defined in 2007/2016[2] rather than the original language *Oberon* as defined in 1988/1990[3].

Oberon (1988/1990) ➔ Oberon-2 (1991)

⬇

Revised Oberon (2007/2016) ➔ **Revised Oberon-2 (2019)**

Revised Oberon-2 implements a superset of *Revised Oberon (Oberon-07)* – which is described in a separate report – adding the following features to the language:

- Type-bound procedures
- Dynamic heap allocation procedure for fixed-length and open arrays
- Numeric case statement
- Exporting and importing of string constants
- Forward references and forward declarations of procedures
- No access to intermediate objects within nested scopes
- Module contexts

A compiler for *Revised Oberon-2* has been implemented for Experimental Oberon[4], a revision of the FPGA Oberon operating system[5]. The combined *total* implementation cost of the compiler additions in source lines of code (sloc) is shown below[6]:

Compiler module	Revised Oberon (Oberon-07)	Revised Oberon-2	Difference	Percent
ORS (scanner)	293	293	0	0 %
ORB (base)	394	437	43	+ 10.9 %
ORG (generator)	984	1110	126	+ 12.8 %
ORP (parser)	949	1140	191	+ 20.1 %
Total	**2620**	**2982**	**360**	**+ 13.7 %**

Feature	Source lines of code
Type-bound procedures	220
Dynamic heap allocation procedure for fixed-length and open arrays	30
Numeric case statement	70
All other features combined	40
Total	**360**

[1] *Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991*
[2] *http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf (Revision 3.5.2016)*
[3] *https://inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf (Revision 1.10.1990)*
[4] *http://www.github.com/andreaspirklbauer/Oberon-experimental*
[5] *http://www.projectoberon.com*
[6] *Not counting about 100-150 additional lines of source code in modules Kernel, Modules and System to complement the implementation of Revised Oberon-2.*

Type-bound procedures

Globally declared procedures may be associated with a record type declared in the same module. The procedures are said to be *bound* to the record type. The binding is expressed by the type of the *receiver* in the heading of a procedure declaration. The receiver may be either a variable parameter of record type *T* or a value parameter of type POINTER TO *T* (where *T* is a record type). The procedure is bound to the type *T* and is considered local to it.

```
ProcedureHeading = PROCEDURE [Receiver] IdentDef [FormalParameters].
Receiver = "(" [VAR] ident ":" ident ")".
```

If a procedure P is bound to a type *T0*, it is implicitly also bound to any type *T1* which is an extension of *T0*. However, a procedure *P'* (with the same name as *P*) may be explicitly bound to *T1* in which case it overrides the binding of *P*. *P'* is considered a *redefinition* of *P* for *T1*. The formal parameters of *P* and *P'* must match. If *P* and *T1* are exported, *P'* must be exported too.

If *v* is a designator and *P* is a type-bound procedure, then *v.P* denotes that procedure *P* which is bound to the dynamic type of *v*. This may be a different procedure than the one bound to the static type of *v*. *v* is passed to *P's* receiver according to the standard parameter passing rules.

If *r* is a receiver parameter declared with type *T*, *r.P^* (pronounced *r.P-referenced*) denotes the (redefined) procedure *P* bound to the base type of *T*.

In a forward declaration of a type-bound procedure the receiver parameter must be of the *same* type as in the actual procedure declaration. The formal parameter lists of both declarations must be identical.

Example:

```
 1  MODULE Trees;
 2    IMPORT Out;
 3
 4    TYPE Tree = POINTER TO Node;
 5      Node = RECORD key : INTEGER;
 6        left, right: Tree
 7      END ;
 8
 9      CenterTree = POINTER TO CenterNode;
10      CenterNode = RECORD (Node) width: INTEGER;
11        subnode: Tree
12      END ;
13
14    PROCEDURE (T: Tree) Insert (node: Tree);          (*procedure bound to Tree*)
15      VAR p, father: Tree;
16    BEGIN p := T;
17      REPEAT father := p;
19        IF node.key < p.key THEN p := p.left
20        ELSIF node.key > p.key THEN p := p.right
21        ELSE p := NIL
22        END
23      UNTIL p = NIL;
24      IF node.key < father.key THEN father.left := node ELSE father.right := node END;
25      node.left := NIL; node.right := NIL
26    END Insert;
```

```
27
28    PROCEDURE (T: CenterTree) Insert (node: Tree);    (*redefinition of Insert bound to CenterTree*)
29    BEGIN Out.Int(node(CenterTree).width, 3);
30      T.Insert^(node)                                 (*calls the Insert procedure bound to Tree*)
31    END Insert;
33
34  END Trees.
```

Dynamic heap allocation procedure for fixed-length and open arrays

If *p* is a variable of type *P = POINTER TO T*, a call of the predefined procedure *NEW* allocates a variable of type *T* in free storage at run time. The type *T* can be a record *or* array type.

If T is a record type or an array type with *fixed* length, the allocation has to be done with

NEW(p)

If T is an *open* array type, the allocation has to be done with

NEW(p, len)

where *T* is allocated with the length given by the expression *len*, which must be an integer type.

In either case, a pointer to the allocated variable is assigned to *p*. This pointer *p* is of type *P*, while the referenced variable *p^* (pronounced *p-referenced*) is of type *T*.

If *T* is a record type, a field *f* of an allocated record *p^* can be accessed as *p^.f* or as *p.f*. If *T* is an array type, the elements of an allocated array *p^* can be accessed as *p^[0]* to *p^[len-1]* or as *p[0]* to *p[len-1]*, i.e. record and array selectors imply dereferencing.

If *T* is an array type, its element type can be a *record*, *pointer*, *procedure* or a *basic* type (BYTE, BOOLEAN, CHAR, INTEGER, REAL, SET), but not an *array* type (no multi-dimensional arrays).

Example:

```
1   MODULE Test;
2     TYPE R = RECORD x, y: INTEGER END ;
3
4       A = ARRAY OF R;                  (*open array*)
5       B = ARRAY 20 OF INTEGER;         (*fixed-length array*)
6
7       P = POINTER TO A;                (*pointer to open array*)
8       Q = POINTER TO B;                (*pointer to fixed-length array*)
9
10    VAR a: P; b: Q;
11
12    PROCEDURE New1*;
13    BEGIN NEW(a, 100);  a[53].x := 1
14    END New1;
15
16    PROCEDURE New2*;
17    BEGIN NEW(b);  b[3] := 2
19    END New2;
20
21  END Test.
```

The following rules and restrictions apply:

- Bounds checks on *fixed-length* arrays are performed at *compile* time.
- Bounds checks on *open* arrays are performed at *run* time.
- If *P* is of type *P = POINTER TO T*, the type *T* must be a *named* record or array type[7].

Allocating dynamic arrays requires a modified version of the inner core module *Kernel*, which introduces a new *kind* of heap block – *array* block in addition to *record* block[8] – and a modified garbage collector to handle them. Array blocks allocated using *NEW(p)* or *NEW(p, len)* are collected by the Oberon garbage collector in the same way as regular *record* blocks[9].

Numeric case statement

The official Oberon-07 language report[10] allows *numeric* case statements, which are however not implemented in the official release[11]. The revised compiler brings the compiler in line with the language report, and now also allows *numeric* CASE statements *(CASE integer OF, CASE char OF)* in addition to *type* case statements *(CASE pointer OF, CASE record OF)*.

Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then the statement sequence is executed whose case label list contains the obtained value. If the case expression is of type INTEGER or CHAR, all labels must be integers or single-character strings, respectively.

```
CaseStatement  =  CASE expression OF case {"|" case} [ELSE StatementSequence] END.
case           =  [CaseLabelList ":" StatementSequence].
CaseLabelList  =  LabelRange {"," LabelRange}.
LabelRange     =  label [".." label].
label          =  integer | string | qualident.
```

Example (*numeric* case statement):

```
1  CASE ch OF
2       "A" .. "Z":  ReadIdentifier
3     | "0" .. "9":  ReadNumber
4     | " ' ", ' " ':  ReadString
5  ELSE SpecialCharacter
6  END
```

The type *T* of the case expression (case variable) may also be a record or pointer type. Then the case labels must be extensions of *T*, and in the statements S_i labelled by T_i, the case variable is considered as of type T_i. Case expressions of *type* case statements must be *simple* identifiers that cannot be followed by selectors, i.e. they cannot be elements of a structure (array elements or record fields).

Example (*type* case case statement):

---

[7] *Restricting pointers to n a m e d arrays is consistent with the official Oberon-07 compiler, which already restrict pointers to point to to n a m e d (but not anonymous) records only.*
[8] *In some implementations of the Oberon system, an additional kind of heap block describing a storage block of n bytes ("sysblk") exists in addition to array and record blocks, typically allocated by a special low-level procedure SYSTEM.NEW(p, n). In our implementation, no such procedure is needed, as it is covered by a call to NEW(p, n), where p is a pointer to an array of BYTE.*
[9] *The implementation of garbage collection on fixed-length and open arrays is similar to implementations in earlier versions of the Original Oberon system. See, for example, "Oberon Technical Notes: Garbage collection on open arrays", J. Templ, ETH technical report, March 1991.*
[10] *http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf (as of 3.5.2016)*
[11] *http://www.projectoberon.com*

```
1   TYPE
2      R = RECORD a: INTEGER END ;
3      R0 = RECORD (R) b: INTEGER END ;
4      R1 = RECORD (R) b: REAL END ;
5      R2 = RECORD (R) b: SET END ;
6      P = POINTER TO R;
7      P0 = POINTER TO R0;
8      P1 = POINTER TO R1;
9      P2 = POINTER TO R2;
10
11  VAR p: P;
12
13  CASE p OF
14    | P0: p.b := 10
15    | P1: p.b := 2.5
16    | P2: p.b := {0, 2}
17  END
```

The following rules and restrictions apply:

- Case labels of numeric statements must have values between 0 and 255.
- If the value of the case expression does not correspond to any case label value in the source text, the statement sequence following the symbol ELSE is selected, if there is one. Otherwise the program is aborted (in the case of a *numeric* case statement)[12] or no action is taken (in the case of a *type* case statement).

Note that the ELSE clause has been re-introduced in the revised compiler for both the *numeric* and the *type* case statements, even though it is not part of the official language definition. This decision, which was made mostly for backward compatibility reasons, may be viewed as being somewhat in contradiction with conventional wisdom in programming language design, which suggests that the ELSE clause in a language construct should be reserved for the *exceptional* cases only, i.e. those that are neither numerous among the possible cases in the source text of a program nor do occur frequently at run time.

The presence of an ELSE clause in the source text may also obfuscate the thinking of the programmer if, for example, program execution falls through to the ELSE clause unintentionally. In general, language constructs that allow program execution to continue or *fall through* to the next case or a "default" case are not recommended. Fortunately though, in most cases an ELSE clause can easily be re-expressed using explicit case label ranges (recommended), e.g.,

```
CASE i OF
  | 1:  S1
  | 3:  S3
  | 7:  S7
  | 9:  S9
ELSE S0
END
```

is the same as

```
CASE i OF
  | 1:  S1
  | 3:  S3
  | 7:  S7
  | 9:  S9
  | 0, 2, 4-6, 8, 10-255:  S0        (*preferred*)
END
```

The use of an ELSE clause should be considered only in well-justified cases, for example if the index range (far) exceeds the label range. But even in that case, one should first try to find a representation using explicit case label ranges, as shown in the example above.

[12] *If one wants to treat such events as "empty" actions, an empty ELSE clause can be used.*

Exporting and importing of string constants

The official Oberon-07 language report allows exporting and importing of string constants, but the compiler does not support it. The revised compiler implements this feature[13].

Example:

```
 1  MODULE M;
 2    CONST s* = "This is a sample string";          (*exported string constant*)
 3  END M.
 4
 5  MODULE N;
 6    IMPORT M, Out;
 7
 8    PROCEDURE P*;
 9    BEGIN Out.Str(M.s)                             (*print the imported string constant*)
10    END P;
11
12  END N.
```

Exported *string constants* are treated like (pre-initialized) exported *variables*. The symbol file contains the string's *export number* and *length*, but not the string itself. The object file contains the actual string, together with its *location* in the exporting module's area for *data*.

Forward references and forward declarations of procedures

The revised compiler implements forward references of procedures for 2 use cases[14]:

Use case A: To make references among nested procedures more efficient:

If a procedure Q which is local to another procedure P refers to the enclosing procedure P, as in

```
1  PROCEDURE P;
2    PROCEDURE Q;
3    BEGIN (*body of Q*) P          (*forward reference from Q to P, as the body of P is not compiled yet *)
4    END Q;
5  BEGIN (*body of P*) …
6  END P;
```

then the official Oberon-07 compiler generates the following code (on FPGA Oberon):

```
20   P'    BL  10          … forward branch to line 31 (across the body of Q to the body of P)
21   Q     body of Q
       ...                 … any calls from Q to P are BACKWARD jumps to line 20 and from there forward to line 31
31   P     body of P
```

whereas the revised compiler generates the following, more efficient, code:

```
20   Q     body of Q
       ...                 … any calls from Q to P are FORWARD jumps to line 30, fixed up when P is compiled
30   P     body of P
```

[13] *http://github.com/andreaspirklbauer/Oberon-importing-string-constants*
[14] *http://github.com/andreaspirklbauer/Oberon-forward-references-of-procedures*

i.e. it does not generate an extra forward jump in line 20 across the body of Q to the body of P and backward jumps from Q to line 20. With the official compiler, the extra BL instruction in line 20 is generated, so that Q can call P (as the body of Q is compiled before the body of P).

Use case B: To implement forward declarations of procedures:

Forward declarations of procedures have been eliminated in Oberon-07, as they can always be eliminated from any program by an appropriate nesting or by introducing procedure variables[15].

Whether forward declarations of procedures *should* be re-introduced into the Oberon language, can of course be debated. Here, we have re-introduced them for three main reasons:

- Direct procedure calls are more efficient than using procedure variables.
- Legacy programs that contain forward references of procedures are now accepted again.
- Introducing forward declarations of procedures added only about 10 lines of source code.

Forward declarations of procedures are implemented in exactly the same way as in the original implementation before the Oberon-07 language revision, i.e.,

- They are explicitly specified by ^ following the symbol PROCEDURE in the source text.
- The compiler processes the heading in the normal way, assuming its body to be missing. The newly generated object in the symbol table is marked as a forward declaration.
- When later in the source text the full declaration is encountered, the symbol table is first searched. If the given identifier is found and denotes a procedure, the full declaration is associated with the already existing entry in the symbol table and the parameter lists are compared. Otherwise a multiple definition of the same identifier is present.

Note that our implementation *both* global and local procedures can be declared forward.

Example:

```
 1  MODULE M;
 2    PROCEDURE^ P(x, y: INTEGER; z: REAL);          (*forward declaration of P*)
 3
 4    PROCEDURE Q*;
 5    BEGIN P(1, 2, 3.0)                            (*Q calls P which is declared forward*)
 6    END Q;
 7
 8    PROCEDURE P(x, y: INTEGER; z: REAL);          (*procedure body of P*)
 9    BEGIN …
10    END P;
11
12  END M.
```

No access to intermediate objects within nested scopes

The official Oberon-07 language report disallows access to *all* intermediate objects from within nested scopes. The revised compiler brings the compiler in line with the language report, i.e. it also disallows access to intermediate *constants* and *types* within nested scopes, not just access to intermediate *variables*[16].

[15] *See section 2 of www.inf.ethz.ch/personal/wirth/Oberon/PortingOberon.pdf*
[16] *http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects*

Like the official Oberon-07 compiler, the revised compiler adopts the convention to implement *shadowing through sco*pe when accessing named objects. This means when two objects share the same name, the one declared at the narrower scope hides, or shadows, the one declared at the wider scope. In such a situation, the *shadowed* element is not available in the narrower scope. If the *shadowing* element is itself declared at an intermediate scope, it is only available at *that* scope level, but *not* in narrower scopes (as access to intermediate objects is disallowed).

The official Oberon-07 compiler already issues an error message, if intermediate *variables* are accessed within nested scopes (line 25 of the program below), *regardless* of whether a global variable with the same name exists (line 7) or not. With the revised compiler, the same error message is now *also* issued for intermediate *constants* (line 21) and *types* (lines 16 and 18).

Example:

```
 1  MODULE Test;
 2    CONST C = 10;            (*global constant C, shadowed in Q and therefore not available in R*)
 3
 4    TYPE G = REAL;           (*global type G, not shadowed in Q and therefore available in R*)
 5         T = REAL;           (*global type T, shadowed in Q and therefore not available in R*)
 6    VAR A,                   (*global variable A, not shadowed in Q and therefore available in R*)
 7        B: INTEGER;          (*global variable B, shadowed in Q and therefore not available in R*)
 8
 9    PROCEDURE P;             (*global procedure P*)
10
11      PROCEDURE Q;           (*intermediate procedure Q, contains shadowing elements*)
12        CONST C = 20;        (*intermediate constant C which shadows the global constant C*)
13        TYPE T = INTEGER;    (*intermediate type T which shadows the global type T*)
14        VAR B: INTEGER;      (*intermediate variable B which shadows the global variable B*)
15
16        PROCEDURE R(x: T): T;  (*access to intermediate type T allowed in original, not allowed in modified compiler*)
17          VAR i: INTEGER;
18            q: T;            (*access to intermediate type T allowed in original, not allowed in modified compiler*)
19            g: G;            (*access to global type G (not shadowed) allowed in both compilers*)
20        BEGIN (*R*)
21          i := C;            (*access to interm. constant C allowed in original, not allowed in modified compiler*)
22          P;                 (*access to global (unshadowed) procedure P allowed in both compilers*)
23          Q;                 (*access to intermediate procedure Q allowed in both compilers*)
24          i := A;            (*access to global (unshadowed) variable A allowed in both compilers*)
25          i := B;            (*access to intermediate variable B not allowed in both compilers*)
26          RETURN i
27        END R;
28      END Q;
29    END P;
30  END Test.
```

Disallowing access to intermediate objects within nested scopes while implementing *shadowing through scope* raises the question whether one should *relax* the shadowing rules and *allow* access to the *global* level, *if* an object sharing the same name with a global object is declared again at an *intermediate* level, but *not* at the strictly local level (“piercing through the shadow").

Such an approach would allow access to the global variable *B* (line 7) in *R* (line 25), effectively ignoring intermediate variable *B* (line 14). It would make nested procedures self-contained in the sense that they can be moved around freely (as intermediate scopes no longer cast a shadow). We have opted not to adopt this approach, in order not to break with the language tradition[17].

[17] *In the appendix of http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects, a possible implementation of such relaxed shadowing rules is provided.*

Module contexts

The revised compiler introduces *module contexts*, which were originally introduced for the A2 operating system[18]. A module context acts as a single-level name space for modules. It allows modules with the same name to co-exist within different contexts. The syntax is defined as:

```
Module   =  MODULE ident [IN ident] ";"
Import   =  IMPORT ident [":=" ident ] [IN ident] ";"
```

In the first line, the optional identifier specifies the name of the context the module belongs to. In the second line, it tells the compiler in which context to look for when importing modules.

Module contexts are implemented as follows:

- Module contexts are specified within the *source* text of a module, as an optional feature. If a context is specified, the name of the source file typically (but not necessarily) contains a prefix indicating its module context, for example *Oberon.Texts.Mod* or *EO.Texts.Mod*.
- If a module context is specified within the source text of a module, the compiler generates the output files *contextname.modulename.smb* and *contextname.modulename.rsc*, i.e. the module context is encoded in the symbol and object file names.
- If no module context is specified within the source text of a module, the compiler generates the output files *modulename.smb* and *modulename.rsc*, i.e. no context is assumed.
- On an Experimental Oberon system, the module context "EO" is implicitly specified at run time, i.e. the module loader first looks for *EO.modulename.rsc*, then for *modulename.rsc*.
- A module cannot be loaded under more than one context on the same system.
- A module belonging to a context can only import modules belonging to the same context or no context (implementation restriction).

Example:

```
 1  MODULE Test1 IN EO;    (*Experimental Oberon*)
 2    IMPORT Out;
 3
 4    PROCEDURE Go1*;
 5    BEGIN Out.Str("Hello from module Test1 in context EO (Experimental Oberon)"); Out.Ln
 6    END Go1;
 7  END Test1.
 8
 9  MODULE Test2 IN EO;    (*Experimental Oberon*)
10    IMPORT Out, Test1 IN EO;
11
12    PROCEDURE Go1*;
13    BEGIN Out.Str("Calling Test1.Go1.. "); Test1.Go1
14    END Go1;
15
16    PROCEDURE Go2*;
17    BEGIN Out.Str("Hello from module Test2 in context EO (Experimental Oberon)"); Out.Ln
18    END Go2;
19  END Test2.
```

* * *

[18] *http://www.ocp.inf.ethz.ch/wiki/Documentation/Language?action=download&upname=contexts.pdf*