Enhanced FPGA Oberon-07 Compiler

Andreas Pirklbauer

12.12.2018

The official FPGA Oberon-07 compiler[1] has been enhanced with various new features, including

- Type-bound procedures (Oberon-2 style)[2]
- Dynamic heap allocation procedure for fixed-length and open arrays (Oberon-2 style)
- 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

The combined *total* implementation cost in the compiler of these new features in source lines of code (*sloc*)[3], broken down by compiler module, is shown below:

Compiler module	FPGA Oberon	Experimental Oberon	Difference	Percent
ORS (scanner)	293	293	0	0 %
ORB (base)	394	440	46	+ 11.7 %
ORG (generator)	984	1112	128	+ 13.0 %
ORP (parser)	949	1143	194	+ 20.4 %
Total	**2620**	**2988**	**368**	**+ 14.0 %**

The detailed implementation cost of the various new features is as follows[4]:

Feature	Source lines of code
Type-bound procedures	~220
Dynamic heap allocation procedure for fixed-length and open arrays	~30
Numeric case statement	~80
All other features combined	~40
Total	**~370**

Type-bound procedures (Oberon-2 style)

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.

[1] *http://www.projectoberon.com*
[2] *Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991*
[3] *Not counting empty lines*
[4] *Not counting approximately 100-150 additional lines of source code in modules Kernel, Modules and System to complement the implementations of type-bound procedures, the dynamic heap allocation procedure NEW(p, len) and module contexts.*

```
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.

```
TYPE Tree = POINTER TO Node;
  Node = RECORD key : INTEGER;
    left, right: Tree
  END ;

  CenterTree = POINTER TO CenterNode;
  CenterNode = RECORD (Node) width: INTEGER;
    subnode: Tree
  END ;

PROCEDURE (t: Tree) Insert (node: Tree);
  VAR p, father: Tree;
BEGIN p := t;
  REPEAT father := p;
    IF node.key < p.key THEN p := p.left
    ELSIF node.key > p.key THEN p := p.right
    ELSE p := NIL
    END
  UNTIL p = NIL;
  IF node.key < father.key THEN father.left := node ELSE father.right := node END;
  node.left := NIL; node.right := NIL
END Insert;

PROCEDURE (t: CenterTree) Insert (node: Tree);  (*redefinition*)
BEGIN Out.Int(node(CenterTree).width, 3);
  t.Insert^(node)  (*calls the Insert procedure bound to Tree*)
END Insert;
```

Dynamic heap allocation procedure for fixed-length and open arrays (Oberon-2 style)

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 type *or* an 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:

```
MODULE Test;
  TYPE R = RECORD x, y: INTEGER END ;

    A = ARRAY OF R;                    (*open array*)
    B = ARRAY 20 OF INTEGER;           (*fixed-length array*)

    P = POINTER TO A;                  (*pointer to open array*)
    Q = POINTER TO B;                  (*pointer to fixed-length array*)

  VAR a: P;
    b: Q;

  PROCEDURE New1*;
  BEGIN NEW(a, 100);  a[53].x := 1
  END New1;

  PROCEDURE New2*;
  BEGIN NEW(b);  b[3] := 2
  END New2;

END Test.
```

If the variable passed as parameter *p* of *NEW(p)* or *NEW(p, len)* is a *named* global pointer variable pointing to a record or an array type, the object referenced by *p* will be marked during the *mark* phase of the garbage collector and collected during the *scan* phase, if it is no longer referenced, i.e. both record *and* array blocks are garbage-collected.

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[5].

[5] *Restricting pointers to n a m e d arrays is consistent with the official Oberon-07 compiler, which only allows pointers to n a m e d records.*

Allocating dynamic arrays requires a modified version of module *Kernel*, which introduces a new *kind* of heap block (*arrayblk* in addition to *recordblk*)[6]. The implementation of garbage collection on open arrays is similar to implementations in earlier versions of the Original Oberon system[7].

Numeric case statement

The official Oberon-07 language report[8] allows numeric CASE statements, which are however not implemented in the official release[9]. The modified Oberon-07 compiler brings the compiler in line with the language report, i.e. it also allows *numeric* CASE statements *(CASE integer OF, CASE char OF)* in addition to *type* CASE statements *(CASE pointer OF, CASE record OF)*[10].

Implemented syntax:

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

Example:

```
CASE i OF
      2..5:  k := 1                (*lower case label limit = 2*)
    | 8..10:  k := 2
    | 13..15:  k := 3
    | 28..30, 18..22:  k := 4
    | 33..36, 24:  k := 5          (*higher case label limit = 36*)
END
```

The essential property of the *numeric* CASE statement is that it represents a *single*, indexed branch, which selects a statement sequence from a set of cases according to an index value. This is in contrast to a cascaded conditional statement which contains multiple branches. Case statements are recommended only if the set of selectable statements is reasonably large.

The compiler constructs a *jump table* of branch statements (containing the branch distances as operands) to the various component statements, leading to a *constant* number of instructions needed for any selection in a numeric CASE statement. The selection ("switch") is generated by procedure *ORG.CaseHead*, using a jump table generated by procedure *ORG.CaseTail.*

Jump tables are located in the *code* section of a module and are addressed relative to the *static base* (SB) of the module. The offset to the static base is restricted to a maximum of 64KB (one could also have placed a table of jump table offsets in the module's area for constants, which is sometimes used if the processor features an indexed branch instruction, as is typically the case in CISC processors, but not in the RISC processor used by FPGA Oberon).

The following rules and restrictions apply:

[6] *A call to NEW(p, n), where p is a pointer to an array of BYTE, is equivalent to a call to the low-level procedure SYSTEM.NEW(p, n) provided in some implementations of the Original Oberon system to allocate a storage block of n bytes ("sysblk"). In our implementation, no such additional low-level procedure exported by the pseudo-module SYSTEM is necessary.*
[7] *See, for example, "Oberon Technical Notes: Garbage collection on open arrays", J. Templ, ETH technical report, March 1991.*
[8] *http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf (as of 3.5.2016)*
[9] *http://www.projectoberon.com*
[10] *http://github.com/andreaspirklbauer/Oberon-numeric-case-statement*

- Case labels must have values between 0 and 255. This makes them ideally suitable for use with CHAR selectors, without imposing any serious restrictions on integer selectors.
- Case expression values that are between 0 and the highest case label limit, but do not correspond to any case label value in the source text, do *not* cause an error termination, i.e. we treat such events as "empty" actions, unless an ELSE clause is present.
- Case expression values that are *higher* than the highest case label limit *do* cause an error termination (trap 1), regardless of whether an ELSE clause is present. This additional rule has been introduced in order to keep the size of the generated jump table reasonably small, which helps optimize the common case where fewer than 256 entries are actually needed.
- Case statements do *not* allow for the empty case, unlike in the official Oberon-07 language, where a *case* is defined using the following grammatical production *with* brackets [and]

  ```
  case = [CaseLabelList ":" StatementSequence].
  ```

  The *inclusion* of the empty case would allow the insertion of superfluous bars similar to the insertion of superfluous semicolons between statements, which we view as .. superfluous ☺

The ELSE clause is implemented for both the *numeric* and the *type* CASE statement, even though it is not part of the official Oberon-07 language definition. If the clause is present, the statement sequence following the ELSE keyword is executed if the value of the case expression does *not* correspond to any case label value in the source text *and* (in the numeric case) lies between 0 and the highest case label limit.

At first glance, this may seem to be somewhat in contradiction with conventional wisdom in programming language design, which suggests that the ELSE clause in a language construct should normally be reserved for the *exceptional* cases only, i.e. those that are neither numerous among the possible cases in the source text 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, for example if 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.

We have nevertheless opted to re-introduce the ELSE clause for the following reason. In our implementation, case expression values *higher* than the highest case label limit cause an error termination. This additional rule effectively limits the *total* range of possible case label values, making it easier to align it with the actual use case. Consequently, if a CASE statement is well designed, the ELSE clause is much more likely to *actually* be used for the exceptional cases – as it should. In addition, there exist genuine examples where an ELSE clause appears useful.

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 modified Oberon-07 compiler implements this feature[11].

Example:

```
MODULE M;
  CONST s* = "This is a sample string";          (*exported string constant*)
END M.

```

[11] *http://github.com/andreaspirklbauer/Oberon-importing-string-constants*

```
MODULE N;
  IMPORT Texts, Oberon, M;
  VAR W: Texts.Writer;

  PROCEDURE P*;
  BEGIN Texts.WriteString(W, M.s); Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END P;

BEGIN Texts.OpenWriter(W)
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 modified Oberon compiler implements forward references of procedures for 2 use cases[12]:

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

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

then the official Oberon-07 compiler generates the following code:

```
20   P'    BL  10          … forward branch to line 31 (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 modified 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
```

i.e. it does not generate an extra forward jump in line 20 around 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[13].

[12] *http://github.com/andreaspirklbauer/Oberon-forward-references-of-procedures*
[13] *See section 2 of www.inf.ethz.ch/personal/wirth/Oberon/PortingOberon.pdf*

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:

```
MODULE M;
  PROCEDURE^ P(x, y: INTEGER; z: REAL);          (*forward declaration of P*)

  PROCEDURE Q*;
  BEGIN P(1, 2, 3.0)                             (*Q calls P which is declared forward*)
  END Q;

  PROCEDURE P(x, y: INTEGER; z: REAL);           (*procedure body of P*)
  BEGIN …
  END P;

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 modified 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*[14].

Like the official compiler, the modified compiler adopts the convention to implement *shadowing through scope*. 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.

The official Oberon-07 compiler already issues an error message, if intermediate *variables* are accessed within nested scopes, *regardless* of whether a global variable with the same name exists or not (e.g. in line 23 of the program below). With the modified compiler, the same error message is now *also* issued for intermediate *constants* (line 19) and *types* (lines 14, 16).

[14] *http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects*

Example:

```
MODULE Test;
  CONST C = 10;                 (*global constant C - shadowed in Q*)
  TYPE G = REAL;                (*global type G - NOT shadowed in Q*)
      T = REAL;                 (*global type T - shadowed in Q*)
  VAR A,                        (*global variable A - NOT shadowed in Q*)
      B: INTEGER;               (*global variable B - shadowed in Q*)

  PROCEDURE P;                  (*global procedure P*)
    PROCEDURE Q;                (*intermediate procedure Q*)
      CONST C = 20;             (*intermediate constant C - shadows global constant C*)
      TYPE T = INTEGER;         (*intermediate type T - shadows global type T*)
      VAR B: INTEGER;           (*intermediate variable B - shadows global variable B*)

      PROCEDURE R(x: T): T;     (*access to int. type T allowed in original, NOT allowed in modified compiler*)
        VAR i: INTEGER;
          q: T;                 (*access to int. type T allowed in original, NOT allowed in modified compiler*)
          g: G;                 (*access to global type G (not shadowed) allowed in BOTH compilers*)
      BEGIN (*R*)
        i := C;                 (*access to int. constant C allowed in original, NOT allowed in modified compiler*)
        P;                      (*access to global procedure P allowed in BOTH compilers*)
        Q;                      (*access to intermediate procedure Q allowed in BOTH compilers*)
        i := A;                 (*access to global variable A (not shadowed) allowed in BOTH compilers*)
        i := B;                 (*access to intermediate variable B NOT allowed in either compiler*)
        RETURN i
      END R;
    END Q;
  END P;

END Test.
```

Module contexts

The modified Oberon-07 compiler introduces *module contexts*, originally introduced for the A2 operating system[15]. 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 itself 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 in the source text, the compiler will automatically generate the output files *contextname.modulename.smb* and *contextname.modulename.rsc*, i.e. the module contexts of symbol and object files is encoded in their file names.

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

- If no module context is specified in the source text, the output files *modulename.rsc* and *modulename.smb* are generated.
- In Experimental Oberon, the module context "EO" is implicitly specified at run time. Thus, the module loader will first look for the file *EO.modulename.rsc*, then for *modulename.rsc*.
- A module belonging to a context can only import modules belonging to the same context, and vice versa (implementation restriction).

Example:

```
MODULE Test1 IN EO;  (*Experimental Oberon*)
  IMPORT Texts, Oberon;
  VAR W: Texts.Writer;

  PROCEDURE Go1*;
  BEGIN Texts.WriteString(W, "Hello from module Test1 in context EO (Experimental Oberon)");
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Go1;

BEGIN Texts.OpenWriter(W)
END Test1.

MODULE Test2 IN EO;  (*Experimental Oberon*)
  IMPORT Texts, Oberon, Test1 IN EO;
  VAR W: Texts.Writer;

  PROCEDURE Go2*;
  BEGIN Texts.WriteString(W, "Hello from module Test2 in context EO (Experimental Oberon)");
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Go2;

BEGIN Texts.OpenWriter(W)
END Test2.
```

* * *