Streamlining symbol files in the Oberon operating system

Andreas Pirklbauer

2.2.2022

Purpose

This technical note presents a simplification of the handling of import and export for the Oberon programming language and system.

Brief historical context

The topic of *symbol files* (module interface files) has accompanied compiler development ever since the original *module* concept with *separate compilation* and type-checking *across* module boundaries (as opposed to *independent* compilation where no such checks are performed) has been introduced in the 70s and adopted in languages such as Mesa, Ada, Modula-2 and Oberon.

A correct implementation of the *module* concept was by no means obvious initially. However, the concept has evolved and today, simple implementations exist covering all key requirements, e.g.,

1. Include non-exported (*hidden*) fields in exported records (for example, for use by the Oberon garbage collector). Types too may be exported even if they are not explicitly marked for export.
2. Handle *re-export conditions*. In Oberon, imported types may be *re-exported* and their *imports* may be hidden.
3. Allow for *recursive data structures*. Pointer declarations may *forward reference* a record type and such forward references must be resolved correctly in symbol files.
4. Handle *module aliases* correctly. A module can be imported under a different name.

A careful and detailed study of the evolution that led to today's status quo – which contains many useful lessons and is therefore well worth the effort – is far beyond the scope of this technical note. The reader is referred to the literature [1-13]. Here, a very rough sketch must suffice:

- Module concept introduced in 1972, early languages include Mesa, Modula and Ada [1].
- Modula-2 implementation on PDP-11 already used the concept of *separate* compilation [2].
- Modula-2 implementation on Lilith already used the concept of *separate* compilation [3].
- First single-pass compiler for Modula-2 compiler in 1984 used a *post-order* traversal [4, 5, 7].
- Some Oberon compilers in the 1990s used a *pre-order* traversal of the symbol table [8-11].
- The Oberon on ARM compiler (2008) used a *fixup* technique for types in symbol files [12].
- The FPGA Oberon RISC compiler (2013) uses *pre-order* traversal and a *fixup* technique [13].

As with the underlying languages, all these re-implementations and refinements of the handling of import and export (and the symbol files) are characterized by a continuous *reduction* of complexity.

In this technical note, we present yet another potential step in this direction by eliminating the so-called "fixup" technique (see below) for *types* in symbol files.

Symbol files in ARM Oberon (2008) and in FPGA Oberon (2013)

The Oberon system and compiler were re-implemented in 2013 using FPGA. The compiler was derived from an earlier version of the Oberon compiler for the ARM processor. In the FPGA Oberon compiler, the same "fixup" technique to implement forward references *in* symbol files as in the ARM Oberon compiler is used. Quoting from the *Oberon on ARM* report [12]:

If a type is imported again and then discarded, it is mandatory that this occurs before a reference to it is established elsewhere. This implies that types must always be defined before they are referenced. Fortunately, this requirement is fulfilled by the language and in particular by the one-pass strategy of the compiler. However, there is one exception, namely the possibility of forward referencing a record type in a pointer declaration, allowing for recursive data structures:

TYPE P = POINTER TO R;
R = RECORD x, y: P END

Hence, this case must be treated in an exceptional way, i.e. the definition of P must not cause the inclusion of the definition of R, but rather cause a forward reference in the symbol file. Such references must by fixed up when the pertinent record declaration had been read. This is the reason for the term {fix} in the syntax of (record) types. Furthermore, the recursive definition

TYPE P = POINTER TO RECORD x, y: P END

suggests that the test for re-import must occur before the type is established, i.e. that the type's name must precede the type's description in the symbol file, where the arrow marks the fixup.:

TYP [#14 P form = PTR [^1]]
TYP [#15 R form = REC [^9] lev = 0 size = 8 {y [^14] off = 4 x [^14] off = 0}] → 14

Observations

The above excerpt correctly states that "*types must always be defined before they are referenced*". However, if ***pre-order traversal*** is used when generating the symbol file – as is the case in FPGA Oberon on RISC – this is *already* the case. When an identifier is to be exported, the export of the type *(Type)* precedes that of the identifier *(Object)*, which therefore always refers to its type by a *backward* reference. Also, a type's name always *precedes* the type's description in the symbol file (see procedures *OutType* and *Export* in *ORB*):

```
PROCEDURE OutType(VAR R: Files.Rider; t: Type);
  ...
BEGIN
  IF t.ref > 0 THEN (*type was already output*) Write(R, -t.ref)
  ELSE ...
    IF t.form = Pointer THEN OutType(R, t.base)
    ELSIF t.form = Array THEN OutType(R, t.base); ...
    ELSIF t.form = Record THEN
      IF t.base # NIL THEN OutType(R, t.base) ELSE OutType(R, noType) END ;
   ELSIF t.form = Proc THEN OutType(R, t.base); ...
   END ; ...
END OutType;
```

```
PROCEDURE Export*(VAR modid: ORS.Ident; VAR newSF: BOOLEAN; VAR key: LONGINT);
BEGIN ...
  WHILE obj # NIL DO
  IF obj.expo THEN
    Write(R, obj.class); Files.WriteString(R, obj.name);   (*type name*)
    OutType(R, obj.type);
    IF obj.class = Typ THEN ...
    ELSIF obj.class = Const THEN ...
    END ;
    obj := obj.next
  END ;
  ...
END Export;
```

And similarly for procedures *InType* and *Import* in *ORB*:

```
PROCEDURE InType(VAR R: Files.Rider; thismod: Object; VAR T: Type);
BEGIN Read(R, ref);
  IF ref < 0 THEN T := typtab[-ref]  (*already read*)
  ELSE NEW(t); T := t; typtab[ref] := t; t.mno := thismod.lev;
      ..
    IF form = Pointer THEN InType(R, thismod, t.base); ...
    ELSIF form = Array THEN InType(R, thismod, t.base); ...
    ELSIF form = Record THEN InType(R, thismod, t.base); ...
    ELSIF form = Proc THEN InType(R, thismod, t.base); ...
    END
  END
END InType;

PROCEDURE Import*(VAR modid, modid1: ORS.Ident);
BEGIN ...
  Read(R, class);
  WHILE class # 0 DO
    NEW(obj); obj.class := class; Files.ReadString(R, obj.name);
    InType(R, thismod, obj.type); ...
    IF class = Typ THEN ...
    ELSE
      IF class = Const THEN ...
      ELSIF class = Var THEN ...
      END
    END ;
    ...
  END
END Import;
```

One can easily verify that types are *always* already “fixed” with the right value, by slightly modifying the current implementation of *ORP.Import* as follows

```
  WHILE k # 0 DO
    IF typtab[k].base # t THEN ORS.Mark("type not yet fixed up") END ;
    typtab[k].base := t; Read(R, k)
  END
```

The message "type not yet fixed up” will *never* be printed while importing a module.

This shows that the fixup of cases of previously declared pointer types is *not necessary* as they are already "fixed" with the right value. A more formal proof can of course easily be constructed. It rests on the observation that the *type* is written to the symbol file before the corresponding *object*.

Code that can be omitted

The following code (shown in red) in procedures *Import* and *Export* in ORB can be omitted. See the appendix for a complete program listing of module ORB showing all changes made.

```
PROCEDURE Import*(VAR modid, modid1: ORS.Ident);
   ...
BEGIN
  IF modid1 = "SYSTEM" THEN
    ...
    IF F # NIL THEN
      ...
      Read(R, class);
      WHILE class # 0 DO
        NEW(obj); obj.class := class; Files.ReadString(R, obj.name);
        InType(R, thismod, obj.type); obj.lev := -thismod.lev;
        IF class = Typ THEN t := obj.type; t.typobj := obj; Read(R, k);  (*always 0*)
          (*fixup bases of previously declared pointer types*)
          WHILE k # 0 DO typtab[k].base := t; Read(R, k) END
        ELSE
          IF class = Const THEN ...
          ELSIF class = Var THEN ...
          END
        END
        obj.next := thismod.dsc; thismod.dsc := obj; Read(R, class)
      END ;
    ELSE ORS.Mark("import not available")
    END
  END
END Import;

PROCEDURE Export*(VAR modid: ORS.Ident; VAR newSF: BOOLEAN; VAR key: LONGINT);
BEGIN ...
  obj := topScope.next;
  WHILE obj # NIL DO
    IF obj.expo THEN
      Write(R, obj.class); Files.WriteString(R, obj.name);
      OutType(R, obj.type);
      IF obj.class = Typ THEN
        IF obj.type.form = Record THEN obj0 := topScope.next;
          (*check whether this is base of previously declared pointer types*)
          WHILE obj0 # obj DO
            IF (obj0.type.form = Pointer) & (obj0.type.base = obj.type)
              & (obj0.type.ref > 0) THEN Write(R, obj0.type.ref) END ;
            obj0 := obj0.next
          END
        END ;
        Write(R, 0)    (*for backward compatibility of symbol files, one may decide to keep this one statement*)
      ELSIF obj.class = Const THEN ...
      ELSIF obj.class = Var THEN ...
      END
    END ;
    obj := obj.next
  END ;
  ...
END Export;
```

Module *ORTool* will also need to be adapted to bring it in sync with the modified module *ORB*. See the appendix for a complete program listings of modules ORB and ORTool.

References

1. Parnas D.L. *On the Criteria To Be Used in Decomposing Systems into Modules*. Comm ACM 15, 12 (December 1972)
2. Wirth N. *MODULA-2*. Computersysteme ETH Zürich, Technical Report No. 36 (March 1980) (ch. 15 describes the use of an implementation of Modula-2 on a DEC PDP-11 computer)
3. Geissmann L. *Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith*, ETH Zürich Dissertation No. 7286 (1983)
4. Wirth N. *A Fast and Compact Compiler for Modula–2*. Computersysteme ETH Zürich, Technical Report No. 64 (July 1985)
5. Gutknecht J. *Compilation of Data Structures: A New Approach to Efficient Modula–2 Symbol Files*. Computersysteme ETH Zürich, Technical Report No. 64 (July 1985)
6. Rechenberg, Mössenböck. *An Algorithm for the Linear Storage of Dynamic Data Structures*. Internal Paper, University of Linz (1986)
7. Gutknecht J. *Variations on the Role of Module Interfaces*. Structured Programming 10, 1, 40-46 (1989)
8. J. Templ. *Sparc–Oberon. User's Guide and Implementation*. Computersysteme ETH Zürich, Technical Report No. 133 (June 1990).
9. Griesemer R. *On the Linearization of Graphs and Writing Symbol Files.* Computersysteme ETH Zürich, Technical Report No. 156a (1991)
10. Pfister, Heeb, Templ. *Oberon Technical Notes.* Computersysteme ETH Zürich, Technical Report No. 156b (1991)
11. Franz M. *The Case for Universal Symbol Files.* Structured Programming 14: 136-147 (1993)
12. Wirth N. *An Oberon Compiler for the ARM Processor*. Technical note (December 2007, April 2008), www.inf.ethz.ch/personal/wirth
13. Wirth N., Gutknecht J. *Project Oberon 2013 Edition*, www.inf.ethz.ch/personal/wirth

Appendix: Changes to modules ORB and ORTool:

Changes to module ORB:

```
MODULE ORB;   (*NW 25.6.2014  / 1.12.2018  in Oberon-07 / AP 12.12.18*)
  IMPORT Files, ORS;
  (*Definition of data types Object and Type, which together form the data structure
    called "symbol table". Contains procedures for creation of Objects, and for search:
    NewObj, this, thisimport, thisfield (and OpenScope, CloseScope).
    Handling of import and export, i.e. reading and writing of "symbol files" is done
    by procedures
    Import and Export. This module contains the list of standard identifiers, with which
    the symbol table (universe), and that of the pseudo-module SYSTEM are initialized. *)

  CONST versionkey* = 1; maxTypTab = 64;
    (* class values*) Head* = 0;
      Const* = 1; Var* = 2; Par* = 3; Fld* = 4; Typ* = 5;
      SProc* = 6; SFunc* = 7; Mod* = 8;

    (* form values*)
      Byte* = 1; Bool* = 2; Char* = 3; Int* = 4; Real* = 5; Set* = 6;
      Pointer* = 7; NilTyp* = 8; NoTyp* = 9; Proc* = 10;
      String* = 11; Array* = 12; Record* = 13;

  TYPE Object* = POINTER TO ObjDesc;
    Module* = POINTER TO ModDesc;
    Type* = POINTER TO TypeDesc;

    ObjDesc*= RECORD
      class*, exno*: BYTE;
      expo*, rdo*: BOOLEAN;   (*exported / read-only*)
      lev*: INTEGER;
      next*, dsc*: Object;
      type*: Type;
      name*: ORS.Ident;
      val*: LONGINT
    END ;

    ModDesc* = RECORD (ObjDesc) orgname*: ORS.Ident END ;

    TypeDesc* = RECORD
      form*, ref*, mno*: INTEGER;  (*ref is only used for import/export*)
      nofpar*: INTEGER;  (*for procedures, extension level for records*)
      len*: LONGINT;  (*for arrays, len < 0 => open array; for records: adr of descriptor*)
      dsc*, typobj*: Object;
      base*: Type;  (*for arrays, records, pointers*)
      size*: LONGINT;  (*in bytes; always multiple of 4, except for Byte, Bool and Char*)
    END ;

  (* Object classes and the meaning of "val":
    class    val
    ----------
    Var      address
    Par      address
    Const    value
    Fld      offset
    Typ      type descriptor (TD) address
    SProc    inline code number
    SFunc    inline code number
    Mod      key

  Type forms and the meaning of "dsc" and "base":
    form     dsc      base
```

```
  ------------------------
  Pointer  -        type of dereferenced object
  Proc     params   result type
  Array    -        type of elements
  Record   fields   extension *)

VAR topScope*, universe, system*: Object;
  byteType*, boolType*, charType*: Type;
  intType*, realType*, setType*, nilType*, noType*, strType*: Type;
  nofmod, Ref: INTEGER;
  typtab: ARRAY maxTypTab OF Type;

PROCEDURE NewObj*(VAR obj: Object; id: ORS.Ident; class: INTEGER); (*insert new Object*)
  VAR new, x: Object;
BEGIN x := topScope;
  WHILE (x.next # NIL) & (x.next.name # id) DO x := x.next END ;
  IF x.next = NIL THEN
    NEW(new); new.name := id; new.class := class; new.next := NIL;
    new.rdo := FALSE; new.dsc := NIL;
    x.next := new; obj := new
  ELSE obj := x.next; ORS.Mark("mult def")
  END
END NewObj;

PROCEDURE thisObj*(): Object;
  VAR s, x: Object;
BEGIN s := topScope;
  REPEAT x := s.next;
    WHILE (x # NIL) & (x.name # ORS.id) DO x := x.next END ;
    s := s.dsc
  UNTIL (x # NIL) OR (s = NIL);
  RETURN x
END thisObj;

PROCEDURE thisimport*(mod: Object): Object;
  VAR obj: Object;
BEGIN
  IF mod.rdo THEN
    IF mod.name[0] # 0X THEN
      obj := mod.dsc;
      WHILE (obj # NIL) & (obj.name # ORS.id) DO obj := obj.next END
    ELSE obj := NIL
    END
  ELSE obj := NIL
  END ;
  RETURN obj
END thisimport;

PROCEDURE thisfield*(rec: Type): Object;
  VAR fld: Object;
BEGIN fld := rec.dsc;
  WHILE (fld # NIL) & (fld.name # ORS.id) DO fld := fld.next END ;
  RETURN fld
END thisfield;

PROCEDURE OpenScope*;
  VAR s: Object;
BEGIN NEW(s); s.class := Head; s.dsc := topScope; s.next := NIL; topScope := s
END OpenScope;

PROCEDURE CloseScope*;
BEGIN topScope := topScope.dsc
END CloseScope;

(*------------------------------ Import ---------------------------------*)

PROCEDURE MakeFileName*(VAR FName: ORS.Ident; name, ext: ARRAY OF CHAR);
  VAR i, j: INTEGER;
BEGIN i := 0; j := 0;  (*assume name suffix less than 4 characters*)
```

```
  WHILE (i < ORS.IdLen-5) & (name[i] > 0X) DO FName[i] := name[i]; INC(i) END ;
  REPEAT FName[i]:= ext[j]; INC(i); INC(j) UNTIL ext[j] = 0X;
  FName[i] := 0X
END MakeFileName;

PROCEDURE ThisModule(name, orgname: ORS.Ident; non: BOOLEAN; key: LONGINT): Object;
  VAR mod: Module; obj, obj1: Object;
BEGIN obj1 := topScope; obj := obj1.next;  (*search for module*)
  WHILE (obj # NIL) & (obj.name # name) DO obj1 := obj; obj := obj1.next END ;
  IF obj = NIL THEN  (*insert new module*)
    NEW(mod); mod.class := Mod; mod.rdo := FALSE;
    mod.name := name; mod.orgname := orgname; mod.val := key;
    mod.lev := nofmod; INC(nofmod); mod.type := noType; mod.dsc := NIL; mod.next := NIL;
    obj1.next := mod; obj := mod
  ELSE (*module already present*)
    IF non THEN ORS.Mark("invalid import order") END
  END ;
  RETURN obj
END ThisModule;

PROCEDURE Read(VAR R: Files.Rider; VAR x: INTEGER);
  VAR b: BYTE;
BEGIN Files.ReadByte(R, b);
  IF b < 80H THEN x := b ELSE x := b - 100H END
END Read;

PROCEDURE InType(VAR R: Files.Rider; thismod: Object; VAR T: Type);
  VAR key: LONGINT;
    ref, class, form, np, readonly: INTEGER;
    fld, par, obj, mod: Object;
    t: Type;
    name, modname: ORS.Ident;
BEGIN Read(R, ref);
  IF ref < 0 THEN T := typtab[-ref]  (*already read*)
  ELSE NEW(t); T := t; typtab[ref] := t; t.mno := thismod.lev;
    Read(R, form); t.form := form;
    IF form = Pointer THEN InType(R, thismod, t.base); t.size := 4
    ELSIF form = Array THEN
      InType(R, thismod, t.base); Files.ReadNum(R, t.len); Files.ReadNum(R, t.size)
    ELSIF form = Record THEN
      InType(R, thismod, t.base);
      IF t.base.form = NoTyp THEN t.base := NIL; obj := NIL ELSE obj := t.base.dsc END ;
      Files.ReadNum(R, t.len); (*TD adr/exno*)
      Files.ReadNum(R, t.nofpar);  (*ext level*)
      Files.ReadNum(R, t.size);
      Read(R, class);
      WHILE class # 0 DO  (*fields*)
        NEW(fld); fld.class := class; Files.ReadString(R, fld.name);
        IF fld.name[0] # 0X THEN fld.expo := TRUE; InType(R, thismod, fld.type)
        ELSE fld.expo := FALSE; fld.type := nilType
        END ;
        Files.ReadNum(R, fld.val); fld.next := obj; obj := fld; Read(R, class)
      END ;
      t.dsc := obj
    ELSIF form = Proc THEN
      InType(R, thismod, t.base);
      obj := NIL; np := 0; Read(R, class);
      WHILE class # 0 DO  (*parameters*)
        NEW(par); par.class := class; Read(R, readonly); par.rdo := readonly = 1;
        InType(R, thismod, par.type); par.next := obj; obj := par; INC(np); Read(R, class)
      END ;
      t.dsc := obj; t.nofpar := np; t.size := 4
    END ;
    Files.ReadString(R, modname);
    IF modname[0] #  0X THEN  (*re-import*)
      Files.ReadInt(R, key); Files.ReadString(R, name);
      mod := ThisModule(modname, modname, FALSE, key);
      obj := mod.dsc;  (*search type*)
      WHILE (obj # NIL) & (obj.name # name) DO obj := obj.next END ;
```

```
      IF obj # NIL THEN T := obj.type   (*type object found in object list of mod*)
      ELSE (*insert new type object in object list of mod*)
        NEW(obj); obj.name := name; obj.class := Typ; obj.next := mod.dsc;
        mod.dsc := obj; obj.type := t;
        t.mno := mod.lev; t.typobj := obj; T := t
      END ;
      typtab[ref] := T
    END
  END
END InType;

PROCEDURE Import*(VAR modid, modid1: ORS.Ident);
  VAR key: LONGINT; class: INTEGER;                ← local variables t and k not needed
    obj, thismod: Object;                          ← obj and thismod on a single line
    modname, fname: ORS.Ident;
    F: Files.File; R: Files.Rider;
BEGIN
  IF modid1 = "SYSTEM" THEN
    thismod := ThisModule(modid, modid1, TRUE, key); DEC(nofmod);
    thismod.lev := 0; thismod.dsc := system; thismod.rdo := TRUE
  ELSE MakeFileName(fname, modid1, ".smb"); F := Files.Old(fname);
    IF F # NIL THEN
      Files.Set(R, F, 0); Files.ReadInt(R, key); Files.ReadInt(R, key);
      Files.ReadString(R, modname);
      thismod := ThisModule(modid, modid1, TRUE, key); thismod.rdo := TRUE;
      Read(R, class); (*version key*)
      IF class # versionkey THEN ORS.Mark("wrong version") END ;
      Read(R, class);
      WHILE class # 0 DO
        NEW(obj); obj.class := class; Files.ReadString(R, obj.name);
        InType(R, thismod, obj.type); obj.lev := -thismod.lev;
        IF class = Typ THEN obj.type.typobj := obj; Read(R, k)   ← code removed
        ELSIF class = Const THEN
          IF obj.type.form = Real THEN Files.ReadInt(R, obj.val
          ELSE Files.ReadNum(R, obj.val)
          END
        ELSIF class = Var THEN Files.ReadNum(R, obj.val); obj.rdo := TRUE
        END ;
        obj.next := thismod.dsc; thismod.dsc := obj; Read(R, class)
      END ;
    ELSE ORS.Mark("import not available")
    END
  END
END Import;

(*------------------------------ Export ---------------------------------*)

PROCEDURE Write(VAR R: Files.Rider; x: INTEGER);
BEGIN Files.WriteByte(R, x)  (* -128 <= x < 128 *)
END Write;

PROCEDURE OutType(VAR R: Files.Rider; t: Type);
  VAR obj, mod, fld, bot: Object;

  PROCEDURE OutPar(VAR R: Files.Rider; par: Object; n: INTEGER);
    VAR cl: INTEGER;
  BEGIN
    IF n > 0 THEN
      OutPar(R, par.next, n-1); cl := par.class;
      Write(R, cl);
      IF par.rdo THEN Write(R, 1) ELSE Write(R, 0) END ;
      OutType(R, par.type)
    END
  END OutPar;

  PROCEDURE FindHiddenPointers(VAR R: Files.Rider; typ: Type; offset: LONGINT);
    VAR fld: Object; i, n: LONGINT;
  BEGIN
    IF (typ.form = Pointer) OR (typ.form = NilTyp) THEN Write(R, Fld); Write(R, 0);
```

```
      Files.WriteNum(R, offset)
    ELSIF typ.form = Record THEN fld := typ.dsc;
      WHILE fld # NIL DO FindHiddenPointers(R, fld.type, fld.val + offset);
        fld := fld.next
      END
    ELSIF typ.form = Array THEN i := 0; n := typ.len;
      WHILE i < n DO FindHiddenPointers(R, typ.base, typ.base.size * i + offset); INC(i) END
    END
  END FindHiddenPointers;

BEGIN
  IF t.ref > 0 THEN (*type was already output*) Write(R, -t.ref)
  ELSE obj := t.typobj;
    IF obj # NIL THEN Write(R, Ref); t.ref := Ref; INC(Ref) ELSE Write(R, 0) END ;
    Write(R, t.form);
    IF t.form = Pointer THEN OutType(R, t.base)
    ELSIF t.form = Array THEN OutType(R, t.base); Files.WriteNum(R, t.len);
      Files.WriteNum(R, t.size)
    ELSIF t.form = Record THEN
      IF t.base # NIL THEN OutType(R, t.base); bot := t.base.dsc
      ELSE OutType(R, noType); bot := NIL
      END ;
      IF obj # NIL THEN
        IF t.mno > 0 THEN Files.WriteNum(R, t.len) ELSE Files.WriteNum(R, obj.exno) END
      ELSE Write(R, 0)
      END ;
      Files.WriteNum(R, t.nofpar); Files.WriteNum(R, t.size);
      fld := t.dsc;
      WHILE fld # bot DO  (*fields*)
        IF fld.expo THEN
          Write(R, Fld); Files.WriteString(R, fld.name); OutType(R, fld.type);
          Files.WriteNum(R, fld.val)  (*offset*)
        ELSE FindHiddenPointers(R, fld.type, fld.val)
        END ;
        fld := fld.next
      END ;
      Write(R, 0)
    ELSIF t.form = Proc THEN OutType(R, t.base); OutPar(R, t.dsc, t.nofpar); Write(R, 0)
    END ;
    IF (t.mno > 0) & (obj # NIL) THEN  (*re-export, output name*)
      mod := topScope.next;
      WHILE (mod # NIL) & (mod.lev # t.mno) DO mod := mod.next END ;
      IF mod # NIL THEN Files.WriteString(R, mod(Module).orgname);
        Files.WriteInt(R, mod.val); Files.WriteString(R, obj.name)
      ELSE ORS.Mark("re-export not found"); Write(R, 0)
      END
    ELSE Write(R, 0)
    END
  END
END OutType;

PROCEDURE Export*(VAR modid: ORS.Ident; VAR newSF: BOOLEAN; VAR key: LONGINT);
  VAR x, sum, oldkey: LONGINT;
    obj: Object;                                    ← local variable obj0 no longer needed
    filename: ORS.Ident;
    F, F1: Files.File; R, R1: Files.Rider;
BEGIN Ref := Record + 1; MakeFileName(filename, modid, ".smb");
  F := Files.New(filename); Files.Set(R, F, 0);
  Files.WriteInt(R, 0); (*placeholder*)
  Files.WriteInt(R, 0); (*placeholder for key to be inserted at the end*)
  Files.WriteString(R, modid); Write(R, versionkey);
  obj := topScope.next;
  WHILE obj # NIL DO
    IF obj.expo THEN
      Write(R, obj.class); Files.WriteString(R, obj.name);
      OutType(R, obj.type);
      IF obj.class = Typ THEN Write(R, 0)           ← code removed
      ELSIF obj.class = Const THEN
        IF obj.type.form = Proc THEN Files.WriteNum(R, obj.exno)
```

```
          ELSIF obj.type.form = Real THEN Files.WriteInt(R, obj.val)
          ELSE Files.WriteNum(R, obj.val)
          END
        ELSIF obj.class = Var THEN Files.WriteNum(R, obj.exno)
        END
      END ;
      obj := obj.next
    END ;
    REPEAT Write(R, 0) UNTIL Files.Length(F) MOD 4 = 0;
    FOR Ref := Record+1 TO maxTypTab-1 DO typtab[Ref] := NIL END ;
    Files.Set(R, F, 0); sum := 0; Files.ReadInt(R, x);  (* compute key (checksum) *)
    WHILE ~R.eof DO sum := sum + x; Files.ReadInt(R, x) END ;
    F1 := Files.Old(filename); (*sum is new key*)
    IF F1 # NIL THEN Files.Set(R1, F1, 4); Files.ReadInt(R1, oldkey) ELSE oldkey:= sum+1 END ;
    IF sum # oldkey THEN
      IF newSF OR (F1 = NIL) THEN
        key := sum; newSF := TRUE; Files.Set(R, F, 4); Files.WriteInt(R, sum);
        Files.Register(F)  (*insert checksum*)
      ELSE ORS.Mark("new symbol file inhibited")
      END
    ELSE newSF := FALSE; key := sum
    END
  END Export;

  PROCEDURE Init*;
  BEGIN topScope := universe; nofmod := 1
  END Init;

  PROCEDURE type(ref, form: INTEGER; size: LONGINT): Type;
    VAR tp: Type;
  BEGIN NEW(tp); tp.form := form; tp.size := size; tp.ref := ref; tp.base := NIL;
    typtab[ref] := tp; RETURN tp
  END type;

  PROCEDURE enter(name: ARRAY OF CHAR; cl: INTEGER; type: Type; n: LONGINT);
    VAR obj: Object;
  BEGIN NEW(obj); obj.name := name; obj.class := cl; obj.type := type; obj.val := n;
    obj.dsc := NIL;
    IF cl = Typ THEN type.typobj := obj END ;
    obj.next := system; system := obj
  END enter;

BEGIN
  byteType := type(Byte, Int, 1);
  boolType := type(Bool, Bool, 1);
  charType := type(Char, Char,1);
  intType := type(Int, Int, 4);
  realType := type(Real, Real, 4);
  setType := type(Set, Set,4);
  nilType := type(NilTyp, NilTyp, 4);
  noType := type(NoTyp, NoTyp, 4);
  strType := type(String, String, 8);

  (*initialize universe with data types and in-line procedures;
    LONGINT is synonym to INTEGER, LONGREAL to REAL.
    LED, ADC, SBC; LDPSR, LDREG, REG, COND are not in language definition*)
  system := NIL;  (*n = procno*10 + nofpar*)
  enter("UML", SFunc, intType, 132);  (*functions*)
  enter("SBC", SFunc, intType, 122);
  enter("ADC", SFunc, intType, 112);
  enter("ROR", SFunc, intType, 92);
  enter("ASR", SFunc, intType, 82);
  enter("LSL", SFunc, intType, 72);
  enter("LEN", SFunc, intType, 61);
  enter("CHR", SFunc, charType, 51);
  enter("ORD", SFunc, intType, 41);
  enter("FLT", SFunc, realType, 31);
  enter("FLOOR", SFunc, intType, 21);
  enter("ODD", SFunc, boolType, 11);
```

```
  enter("ABS", SFunc, intType, 1);
  enter("LED", SProc, noType, 81);  (*procedures*)
  enter("UNPK", SProc, noType, 72);
  enter("PACK", SProc, noType, 62);
  enter("NEW", SProc, noType, 51);
  enter("ASSERT", SProc, noType, 41);
  enter("EXCL", SProc, noType, 32);
  enter("INCL", SProc, noType, 22);
  enter("DEC", SProc, noType, 11);
  enter("INC", SProc, noType, 1);
  enter("SET", Typ, setType, 0);   (*types*)
  enter("BOOLEAN", Typ, boolType, 0);
  enter("BYTE", Typ, byteType, 0);
  enter("CHAR", Typ, charType, 0);
  enter("LONGREAL", Typ, realType, 0);
  enter("REAL", Typ, realType, 0);
  enter("LONGINT", Typ, intType, 0);
  enter("INTEGER", Typ, intType, 0);
  topScope := NIL; OpenScope; topScope.next := system; universe := topScope;

  system := NIL;  (* initialize "unsafe" pseudo-module SYSTEM*)
  enter("H", SFunc, intType, 201);     (*functions*)
  enter("COND", SFunc, boolType, 191);
  enter("SIZE", SFunc, intType, 181);
  enter("ADR", SFunc, intType, 171);
  enter("VAL", SFunc, intType, 162);
  enter("REG", SFunc, intType, 151);
  enter("BIT", SFunc, boolType, 142);
  enter("LDREG", SProc, noType, 142);  (*procedures*)
  enter("LDPSR", SProc, noType, 131);
  enter("COPY", SProc, noType, 123);
  enter("PUT", SProc, noType, 112);
  enter("GET", SProc, noType, 102)
END ORB.
```

Changes to module ORTool:

```
MODULE ORTool;  (*NW 18.2.2013 / 15.9.2018*)
  IMPORT Files, Texts, Oberon, ORB;                          ← no longer import module SYSTEM
  VAR W: Texts.Writer;
    mnemo0, mnemo1: ARRAY 16, 4 OF CHAR;  (*mnemonics*)      ← global variable "Form" removed

  PROCEDURE Read(VAR R: Files.Rider; VAR x: INTEGER);
    VAR b: BYTE;
  BEGIN Files.ReadByte(R, b);
    IF b < 80H THEN x := b ELSE x := b - 100H END
  END Read;

  PROCEDURE ReadType(VAR R: Files.Rider);
    VAR key, len, size, off: INTEGER;                        ← local variable "lev" removed
      ref, class, form, readonly: INTEGER;                   ← local variable "mno" removed
      name, modname: ARRAY 32 OF CHAR;
  BEGIN Read(R, ref); Texts.Write(W, " "); Texts.Write(W, "[");
    IF ref < 0 THEN Texts.Write(W, "^"); Texts.WriteInt(W, -ref, 1)
    ELSE Texts.WriteInt(W, ref, 1);
      Read(R, form); Texts.WriteString(W, "  form = "); Texts.WriteInt(W, form, 1);
      IF form = ORB.Pointer THEN ReadType(R)
      ELSIF form = ORB.Array THEN
        ReadType(R); Files.ReadNum(R, len); Files.ReadNum(R, size);
        Texts.WriteString(W, "  len = "); Texts.WriteInt(W, len, 1);
        Texts.WriteString(W, "  size = "); Texts.WriteInt(W, size, 1)
      ELSIF form = ORB.Record THEN
        ReadType(R);  (*base type*)
        Files.ReadNum(R, off);
          Texts.WriteString(W, "  exno = "); Texts.WriteInt(W, off, 1);
        Files.ReadNum(R, off); Texts.WriteString(W, "  extlev = "); Texts.WriteInt(W, off, 1);
```

```
        Files.ReadNum(R, size); Texts.WriteString(W, "  size = "); Texts.WriteInt(W, size, 1);
        Texts.Write(W, " "); Texts.Write(W, "{"); Read(R, class);
        WHILE class # 0 DO (*fields*)
          Files.ReadString(R, name);
          IF name[0] # 0X THEN Texts.Write(W, " "); Texts.WriteString(W, name); ReadType(R)
          ELSE Texts.WriteString(W, " --")
          END ;
          Files.ReadNum(R, off); Texts.WriteInt(W, off, 4); Read(R, class)
        END ;
        Texts.Write(W, "}")
      ELSIF form = ORB.Proc THEN
        ReadType(R); Texts.Write(W, "("); Read(R, class);
        WHILE class # 0 DO (*parameters*)
          Texts.WriteString(W, " class = "); Texts.WriteInt(W, class, 1); Read(R, readonly);
          IF readonly = 1 THEN Texts.Write(W, "#") END ;
          ReadType(R); Read(R, class)
        END ;
        Texts.Write(W, ")")
      END ;
      Files.ReadString(R, modname);
      IF modname[0] # 0X THEN
        Files.ReadInt(R, key); Files.ReadString(R, name);
        Texts.Write(W, " "); Texts.WriteString(W, modname);
        Texts.Write(W, "."); Texts.WriteString(W, name);
        Texts.WriteHex(W, key)
      END
    END ;
    Texts.Write(W, "]")
  END ReadType;

  PROCEDURE DecSym*;  (*decode symbol file*)
    VAR class, k: INTEGER;
      name: ARRAY 32 OF CHAR;
      F: Files.File; R: Files.Rider;
      S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
    IF S.class = Texts.Name THEN
      Texts.WriteString(W, "OR-decode "); Texts.WriteString(W, S.s);
      Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf);
      F := Files.Old(S.s);
      IF F # NIL THEN
        Files.Set(R, F, 0); Files.ReadInt(R, k); Files.ReadInt(R, k);
        Files.ReadString(R, name); Texts.WriteString(W, name); Texts.WriteHex(W, k);
        Read(R, class); Texts.WriteInt(W, class, 3); (*sym file version*)
        IF class = ORB.versionkey THEN
          Texts.WriteLn(W); Read(R, class);
          WHILE class # 0 DO
            Texts.WriteInt(W, class, 4); Files.ReadString(R, name);
            Texts.Write(W, " "); Texts.WriteString(W, name);
            ReadType(R);
            IF class = ORB.Typ THEN Read(R, class)                    ← code removed
            ELSIF (class = ORB.Const) OR (class = ORB.Var) THEN
              Files.ReadNum(R, k); Texts.WriteInt(W, k, 5);  (*Reals, Strings!*)
            END ;
            Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf);
            Read(R, class)
          END
        ELSE Texts.WriteString(W, " bad symfile version")
        END
      ELSE Texts.WriteString(W, " not found")
      END ;
      Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
    END
  END DecSym;

(* ------------------------------------------------*)

  PROCEDURE WriteReg(r: LONGINT);
  BEGIN Texts.Write(W, " ");
```

```
    IF r < 12 THEN Texts.WriteString(W, " R"); Texts.WriteInt(W, r MOD 10H, 1)
    ELSIF r = 12 THEN Texts.WriteString(W, "MT")
    ELSIF r = 13 THEN Texts.WriteString(W, "SB")
    ELSIF r = 14 THEN Texts.WriteString(W, "SP")
    ELSE Texts.WriteString(W, "LNK")
    END
  END WriteReg;

  PROCEDURE opcode(w: LONGINT);
    VAR k, op, u, a, b: LONGINT;
  BEGIN
      k := w DIV 40000000H MOD 4;
      a := w DIV 1000000H MOD 10H;
      b := w DIV 100000H MOD 10H;
      op := w DIV 10000H MOD 10H;
      u := w DIV 20000000H MOD 2;
      IF k = 0 THEN
        Texts.WriteString(W, mnemo0[op]);
        IF u = 1 THEN Texts.Write(W, "'") END ;
        WriteReg(a); WriteReg(b); WriteReg(w MOD 10H)
      ELSIF k = 1 THEN
        Texts.WriteString(W, mnemo0[op]);
        IF u = 1 THEN Texts.Write(W, "'") END ;
        WriteReg(a); WriteReg(b); w := w MOD 10000H;
        IF w >= 8000H THEN w := w - 10000H END ;
        Texts.WriteInt(W, w, 7)
      ELSIF k = 2 THEN  (*LDR/STR*)
        IF u = 1 THEN Texts.WriteString(W, "STR ") ELSE Texts.WriteString(W, "LDR") END ;
        WriteReg(a); WriteReg(b); w := w MOD 100000H;
        IF w >= 80000H THEN w := w - 100000H END ;
        Texts.WriteInt(W, w, 8)
      ELSIF k = 3 THEN  (*Branch instr*)
        Texts.Write(W, "B");
        IF ODD(w DIV 10000000H) THEN Texts.Write(W, "L") END ;
        Texts.WriteString(W, mnemo1[a]);
        IF u = 0 THEN WriteReg(w MOD 10H) ELSE
          w := w MOD 100000H;
          IF w >= 80000H THEN w := w - 100000H END ;
          Texts.WriteInt(W, w, 8)
        END
      END
  END opcode;

  PROCEDURE Sync(VAR R: Files.Rider);
    VAR ch: CHAR;
  BEGIN Files.Read(R, ch); Texts.WriteString(W, "Sync "); Texts.Write(W, ch); Texts.WriteLn(W)
  END Sync;

  PROCEDURE Write(VAR R: Files.Rider; x: INTEGER);
  BEGIN Files.WriteByte(R, x)  (* -128 <= x < 128 *)
  END Write;

  PROCEDURE DecObj*;   (*decode object file*)
    VAR class, i, n, key, size, adr, data: INTEGER;      ← local vars “fix” and “len” removed
      ch: CHAR;
      name: ARRAY 32 OF CHAR;
      F: Files.File; R: Files.Rider;
      S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
    IF S.class = Texts.Name THEN
      Texts.WriteString(W, "decode "); Texts.WriteString(W, S.s); F := Files.Old(S.s);
      IF F # NIL THEN
        Files.Set(R, F, 0); Files.ReadString(R, name); Texts.WriteLn(W); Texts.WriteString(W,
name);
        Files.ReadInt(R, key); Texts.WriteHex(W, key); Read(R, class); Texts.WriteInt(W, class,
4); (*version*)
        Files.ReadInt(R, size); Texts.WriteInt(W, size, 6); Texts.WriteLn(W);
        Texts.WriteString(W, "imports:"); Texts.WriteLn(W); Files.ReadString(R, name);
        WHILE name[0] # 0X DO
```

```
          Texts.Write(W, 9X); Texts.WriteString(W, name);
          Files.ReadInt(R, key); Texts.WriteHex(W, key); Texts.WriteLn(W);
          Files.ReadString(R, name)
        END ;
      (* Sync(R); *)
        Texts.WriteString(W, "type descriptors"); Texts.WriteLn(W);
        Files.ReadInt(R, n); n := n DIV 4; i := 0;
        WHILE i < n DO Files.ReadInt(R, data); Texts.WriteHex(W, data); INC(i) END ;
        Texts.WriteLn(W);
        Texts.WriteString(W, "data"); Files.ReadInt(R, data); Texts.WriteInt(W, data, 6);
Texts.WriteLn(W);
        Texts.WriteString(W, "strings"); Texts.WriteLn(W);
        Files.ReadInt(R, n); i := 0;
        WHILE i < n DO Files.Read(R, ch); Texts.Write(W, ch); INC(i) END ;
        Texts.WriteLn(W);
        Texts.WriteString(W, "code"); Texts.WriteLn(W);
        Files.ReadInt(R, n); i := 0;
        WHILE i < n DO
          Files.ReadInt(R, data); Texts.WriteInt(W, i, 4); Texts.Write(W, 9X); Texts.WriteHex(W,
data);
          Texts.Write(W, 9X); opcode(data); Texts.WriteLn(W); INC(i)
        END ;
      (* Sync(R); *)
        Texts.WriteString(W, "commands:"); Texts.WriteLn(W);
        Files.ReadString(R, name);
        WHILE name[0] # 0X DO
          Texts.Write(W, 9X); Texts.WriteString(W, name);
          Files.ReadInt(R, adr); Texts.WriteInt(W, adr, 5); Texts.WriteLn(W);
          Files.ReadString(R, name)
        END ;
      (* Sync(R); *)
        Texts.WriteString(W, "entries"); Texts.WriteLn(W);
        Files.ReadInt(R, n); i := 0;
        WHILE i < n DO
          Files.ReadInt(R, adr); Texts.WriteInt(W, adr, 6); INC(i)
        END ;
        Texts.WriteLn(W);
      (* Sync(R); *)
        Texts.WriteString(W, "pointer refs"); Texts.WriteLn(W); Files.ReadInt(R, adr);
        WHILE adr # -1 DO Texts.WriteInt(W, adr, 6); Files.ReadInt(R, adr) END ;
        Texts.WriteLn(W);
      (* Sync(R); *)
        Files.ReadInt(R, data); Texts.WriteString(W, "fixP = "); Texts.WriteInt(W, data, 8);
Texts.WriteLn(W);
        Files.ReadInt(R, data); Texts.WriteString(W, "fixD = "); Texts.WriteInt(W, data, 8);
Texts.WriteLn(W);
        Files.ReadInt(R, data); Texts.WriteString(W, "fixT = "); Texts.WriteInt(W, data, 8);
Texts.WriteLn(W);
        Files.ReadInt(R, data); Texts.WriteString(W, "entry = "); Texts.WriteInt(W, data, 8);
Texts.WriteLn(W);
        Files.Read(R, ch);
        IF ch # "O" THEN Texts.WriteString(W, "format eror"); Texts.WriteLn(W) END
      (* Sync(R); *)
      ELSE Texts.WriteString(W, " not found"); Texts.WriteLn(W)
      END ;
      Texts.Append(Oberon.Log, W.buf)
    END
  END DecObj;

BEGIN Texts.OpenWriter(W); Texts.WriteString(W, "ORTool 1.12.2018")
  Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf);
  mnemo0[0] := "MOV";
  mnemo0[1] := "LSL";
  mnemo0[2] := "ASR";
  mnemo0[3] := "ROR";
  mnemo0[4] := "AND";
  mnemo0[5] := "ANN";
  mnemo0[6] := "IOR";
  mnemo0[7] := "XOR";
```

```
  mnemo0[8] := "ADD";
  mnemo0[9] := "SUB";
  mnemo0[10] := "MUL";
  mnemo0[11] := "DIV";
  mnemo0[12] := "FAD";
  mnemo0[13] := "FSB";
  mnemo0[14] := "FML";
  mnemo0[15] := "FDV";
  mnemo1[0] := "MI ";
  mnemo1[8] := "PL";
  mnemo1[1] := "EQ ";
  mnemo1[9] := "NE ";
  mnemo1[2] := "LS ";
  mnemo1[10] := "HI ";
  mnemo1[5] := "LT ";
  mnemo1[13] := "GE ";
  mnemo1[6] := "LE ";
  mnemo1[14] := "GT ";
  mnemo1[15] := "NO”
END ORTool.
```