Streamlining symbol files in Oberon

Andreas Pirklbauer

11.11.2022

Overview

This technical note presents a simplification of the handling of import and export for the Oberon programming language, as realized in *Extended Oberon*[1], a revision of the *Project Oberon 2013 (PO 2013)* system, which is itself a reimplementation of the original *Oberon* system on an FPGA development board around 2013, as published at *www.projectoberon.com.*

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. *Hidden record fields:* Offsets of non-exported pointer fields are needed for garbage collection.
2. *Re-export conditions*: Imported types may be *re-exported* and their *imports* may be hidden.
3. *Recursive data structures*: Pointer declarations may *forward reference* a record type.
4. *Module aliases*: A module can be imported under a different (alias) 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 Project Oberon 2013 compiler 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 simplification by eliminating the so-called "fixup" technique for *types* during export and subsequent import.

[1] *http://www.github.com/andreaspirklbauer/Oberon-extended*

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

The Oberon system and compiler were re-implemented around 2013 on an FPGA development board (www.projectoberon.com). The compiler was derived from an earlier version of the Oberon compiler for the ARM processor. In the Project Oberon 2013 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 traversing the data structure called the “symbol table” to generate the symbol file – as is the case in Project Oberon 2013 – 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 *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.

```
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);   (*<---*)
          (*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)                         (*<---*)
      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*. Some additional improvements to the symbol file structure are described below.

Propagate imported export numbers of type descriptor addresses to client modules

We have replaced the following code in *ORB.OutType* from the PO 2013 implementation:

```
ELSIF t.form = Record THEN
  ...
  IF obj # NIL THEN Files.WriteNum(R, obj.exno) ELSE Write(R, 0) END ;
```

with:

```
ELSIF t.form = Record THEN
  ...
  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 ;
```

This makes sure that *imported* export numbers of type descriptor addresses (stored in the field *t.len* of the type node in the symbol table of the compiler) are re-exported to client modules, making it possible to perform type tests on such types when they are later re-imported.

The reader may think that this change is unnecessary, because one cannot perform a type test on re-imported types anyway (since only *explicitly* imported types can be referred to *by name* in client modules). However, our implementation *allows* explicitly importing a module *M after* types of *M* have previously been re-imported via other modules. In such a case, any previously re-imported type of M will simply be converted to an explicitly imported one in the compiler's symbol table.

Thus, we must make sure that if a type is exported by the declaring module *M* and later imported (or re-imported) and then re-exported by another module, the export number of the type descriptor address in the *declaring* module M is propagated through the module hierarchy correctly.

Handle alias type names among imported modules correctly

We have replaced the following code in *ORB.Import* from the *PO 2013* implementation:

```
obj.type.typobj := obj
```

with:

```
IF obj.typ.typobj = NIL THEN obj.type.typobj := obj END
```

which establishes the *typobj* backpointer only if it doesn't exist yet. This makes sure that an imported alias type always points to the original (imported) type, not another (imported) alias type.

This is the same type of precaution as in *ORP.Declarations*, where alias types declared in the module currently being compiled are initialized as follows:

```
IF tp.typobj = NIL THEN tp.typobj := obj END
```

which also makes sure that an alias type declared in the module currently being compiled always points to the original type (no matter whether the original type is imported or declared in the module being compiled).

Allow reusing the original module name if a module has been imported under an alias name

The Oberon language report defines an aliased module import as follows: *If the form "M := M1" is used in the import list, an exported object x declared within M1 is referenced in the importing module as M.x*.

Unfortunately, this definition allows several different interpretations.

In our implementation, we have adopted the following interpretation:

- *It is module M1 that is imported, not M*
- *The module alias name M renames module M1*
- *A module can only have a single module alias name*

This interpretation implies that an imported object *x* can be accessed only through *one* qualified identifier *M.x* and allows reusing the original module name *M1* if a module has been imported under a module alias name *M*.

For example, the following scenarios are all legal:

```
MODULE A1; IMPORT M0 := M1, M1 := M2; END A1.
MODULE A2; IMPORT M0 := M1, M1 := M0; END A2.
MODULE A3; IMPORT M0 := M1, M2 := M0; END A3.
```

whereas the following scenario is illegal:

```
MODULE B1;
 IMPORT M1, A := M1, B := M1, C := M1;  (*only a single alias name allowed*)
 VAR m: M1.T; a: A.T; b: B.T; c: C.T;
END B1.
```

This is implemented by simply *not* checking the two cross-combinations *obj.orgname = name and obj.name = orgname* in *ORB.ThisModule*, where *obj* is an existing module in the module list.

Not checking the cross-combination *obj.orgname = name* allows the explicit import M := M1 in:

```
MODULE A1; IMPORT M0 := M, M := M1; END A1
```

Not checking the cross-combination *obj.name = orgname* allows the explicit import M := M0 in:

```
MODULE A2; IMPORT M0 := M, M := M0; END A2.
MODULE A3; IMPORT M0 := M, X := M0; END A3.
```

An alternative interpretation to the one that we have adopted would have been:

- *It is module M1 that is imported, not M*
- *A module alias name M is just an additional name for M1*
- *A module can have multiple module alias names*

This interpretation does *not* allow reusing the original module name *M1* (i.e. *M1* remains valid) and implies that an imported object *x* can be accessed through *multiple* identifiers *M1.x, M.x, N.x*, etc.

This interpretation is similar to an *alias type* in Oberon (a type can have multiple alias types, and the original type name remains valid). Under this definition, modules *A1-A3* above would be illegal, whereas *B1* would be legal.

We have decided *not* to adopt this alternative interpretation for various reasons:

- Accessing the same imported object through different identifiers may be confusing.
- It disallows the sometimes useful ability to "swap" two modules by writing *M0 := M1, M1 := M0*.
- The symbol table data structure would be even more complex due to the need to now having to manage a *list* of alias objects in addition to the actual module objects – we believe this to be unnecessary complexity. If used at all, a single module alias should suffice.

Allow re-imports to co-exist with module alias names and globally declared identifiers

In our implementation, we allow re-imported modules to co-exist with:

- module alias names of explicitly imported modules
- globally declared identifiers of the importing module

Making symbol files "self-contained" by re-exporting types that stem from other imported modules is a hidden *implementation* mechanism. One could also have chosen a different approach. For example, one could have decided to recursively import all symbol files (in full) from which types are needed. Since the programmer should not know about such implementation details, we have decided to "hide" re-imports from the module name space. This is implemented by simply *skipping* over re-imports in various loops that traverse the list of identifiers in the module list anchored in *ORB.topScope*.

Noting that *obj.rdo = TRUE* means "explicit import" and *obj.rdo = FALSE* means "re-import", this is achieved by adding the extra condition

```
OR (obj.class = Mod) & ~obj.rdo   (*skip over re-imports*)
```

to the various loops in procedures *ORB.NewObj*, *ORB.thisObj* and *ORB.ThisModule*:

Allow an explicit import after previous re-imports of types of the same module

We also allow types of a module *M* to be first re-imported and then *M* to be explicitly imported. For example, in the following scenario:

- A base module M exports the types *M.T0, M.T1* and *M.T2*
- Module *M0* imports module *M* and re-exports the type *M.T0* (to clients of *M0*)
- Module *M1* imports module *M* and re-exports the type *M.T1* (to clients of *M1*)
- Module *C* first imports modules *M0* and *M1*, and then imports module *M*

```
MODULE C;
IMPORT M0, M1, M;   (*re-imports M.T0 via M0 and M.T1 via M1, then directly imports M.T2 from M*)
```

the types *M.T0, M.T1* and *M.T2* are imported by module *C* in the following order:

- By importing module *M0*, module *C* re-imports the type *M.T0* (via *M0*)

- By importing module *M1*, module *C* re-imports the type *M.T1* (via *M1*)
- By importing module *M*, module *C* then imports the type *M.T2* from *M* directly

A correct implementation must make sure that for *named* types, which may be re-imported via *several* symbol files, the first loaded instance (called the "primary instance") is always used and inserted into the symbol table of the compiler and the translation table *typtab*, and a module object for its *declaring* module is inserted into the symbol table as well.

If the primary instance already exists when reading a type again later from a different symbol file – either directly from the symbol file of the declaring module or indirectly via a re-import – the remaining type data is read and then discarded.

This is necessary because we must make sure that a type which is imported or re-imported via *multiple*symbol files is always mapped to the *same* type node in the symbol table. Otherwise, incorrect incompatibilites during *type checking* may occur.

Our implementation approach is as follows:

1. Propagate the *reference number* of a type *t* (e.g. the type *M.T0*, *M.T1* or *M.T2* in the above example) in its *declaring* module *M* across the entire module hierarchy by means of a newly introduced field *t.orgref* (*ORB.InType*, *ORB.OutType*).

2. When a module M exports a type *t* (e.g., the type *M.T0* in the above example) and a module C *re-imports* the type *t* via another module M0, a module object for its *declaring* module M and a type object for the imported type *M.T0* in the object list of *M* – together with its original reference number *in M* – is inserted into the symbol table (*ORB.InType*).

3. When module *C* later also *explicitly* imports module *M*, we start by initializing the translation table *typtab* for module *M* with all previously re-imported types, using their reference numbers in their *declaring* module *M* as the index to the translation table. In the above example, these are the types *M.T0, M.T1* and *M.T2* (*ORB.Import*).

4. When a type *t* declared in *M* is read from the symbol file of *M* for the *first* time, we *use* the information in the translation table to detect whether *t* has previously been re-imported via *other* modules or not (*ORB.InType*).

5. If the type t *has* been re-imported via other modules before, we read the type information from the symbol file of *M* but then discard it. If the type t has *not* been re-imported before, we create a new entry for it in the object list of module M (*ORB.InType*).

6. Before explicitly importing an object of class *Typ*, we check whether the object has already been re-imported. If so, we reuse the existing object and discard the one just read from the symbol file (*ORB.Import*).

In the above example, the type *M.T0* will be identified as having been re-imported via module *M0*, and the type *M.T1* will be identified as having been re-imported via module *M1*. After these two re-imports, the type *M.T2* will be explicitly imported by *C*, since it has not been re-imported via other modules before.

Write the module anchor before the type description to the symbol file (pre-order)

An additional detail is perhaps worth mentioning: Our implementation writes the module anchor (module name and key) immediately *after* the type's reference number to the symbol file, but *before* the remaining type description (insead of *after* it, as in Project Oberon 2013).

This ensures that the code *also* works in the presence of *cyclic* references among types (recall that a named type may refer to itself, either directly or indirectly via type extensions).

If, for example, a record type *R1* contains a field of type *P2 = POINTER TO R2*, where the type *R2* is an extension of *R1*, then procedure *ORB.InType* will be recursively called for the types R1 -> P2 -> R2 -> R1 (in this order!), where the last call to *ORB.InType(R, thismod, t.base)* is made when reading the type R2, i.e. with *t.base* = *R1*.

If the name and key of the module, from which the type *R1* is re-imported, were stored in the symbol file *after* the type description of *R1*, the recursion R1 -> P2 -> R2 -> R1 would also start *after* the initial assignment to *t.base* (via the statement *T := t* at the beginning of *ORB.InType*), but *before* the code to read the name and key of the module from which R1 is re-imported is executed.Thus, the last call to *InType(R, thismode, t.base)* when reading *R2* (with *t.base* = *R1*) would also be made *before* the code to handle the re-import of the base type R1 is executed.

But this code may *change* the field *t.base* to an entry from the translation table *typtab* (since the parameter *T* is *variable* parameter in *ORB.InType*), thereby invalidating the initial assignment to *t.base* that was made at the beginning of *InType(R, thismod, t.base)*.

Wonders of multiple nested recursions among record and pointer base types! A little subtle indeed.

All in all, it seems best to be on the safe side by simply making sure that module anchors for re-imported types are created and inserted into the symbol table *before* any calls to *InType(R, thismod, t.base)* are made.

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