Streamlining symbol files in Oberon

Andreas Pirklbauer

1.10.2023

Overview

This technical note presents both a simplification and improvement 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* 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. *Cyclic references:* Record and pointer types may contain cyclic references among themselves.
5. *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 in 1979 already used *separate* compilation [2]
- Modula-2 implementation on Lilith in 1980 already used *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 note, we present yet another simplification by eliminating the so-called "fixup" technique for *types* during export and subsequent import, as well as some additional improvements.

[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 in in Project Oberon (2013)

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* also needs to be adapted to bring it in sync with the modified module *ORB*.

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

In the Oberon programming language, imported types can be re-exported and their original import may be hidden from the re-importing module during the *import process*. This means that a type from one module *(M.T)* can be imported by another module *(M1)* and then re-exported to a third module *(M2)* without *M2* being aware of the original import from *M*. This can lead to hidden dependencies and create a complex hierarchy of module imports.

In Oberon, two common approaches to implement the re-export mechanism include:

1. *Self-contained symbol files*: This approach involves including imported types in symbol files, making the files self-contained and complete. This ensures that all required information is available in each symbol file, eliminating the need for recursive imports.

2. *Recursive imports*: In this approach, all required symbol files are imported recursively in full, to ensure that all necessary types are available for use. This method can result in more imports, but it is more transparent and easier to understand the dependencies between modules.

Project Oberon (2013) has chosen the first approach (self-contained symbol files). But it does not allow an explicit import of a module *M* after individual types of *M* have previously been re-imported via other modules. Our implementation removes this limitation. Consider the following scenario:

```
MODULE M;
  TYPE T0* = RECORD (*…*) END; T1* = RECORD (*…*) END; T2* = RECORD (*…*) END;
END M.

MODULE M0;
  IMPORT M;                  (*import all types of M*)
  VAR t0*: M.T0;             (*re-export type M.T0 to clients of M0*)
END M0.

MODULE M1;
  IMPORT M;                  (*import all types of M*)
  VAR t0*: M.T0;             (*re-export type M.T0 to clients of M1*)
    t1*: M.T1;               (*re-export type M.T1 to clients of M1*)
END M1.

MODULE C;                    (*allowed in Project Oberon 2013 and in Extended Oberon*)
  IMPORT M0,                 (*re-import type M.T0 via module M0*)
    M1;                      (*re-import types M.T0 and M.T1 via module M1*)
END C.

MODULE D;                    (*allowed in Project Oberon 2013 and in Extended Oberon*)
  IMPORT M,                  (*import types M.T0, M.T1 and M.T2 from module M directly*)
    M0,                      (*re-import type M.T0 via module M0*)
    M1;                      (*re-import types M.T0 and M.T1 via module M1*)
END D.

MODULE E;                    (*not allowed in Project Oberon 2013, allowed in Extended Oberon*)
  IMPORT M0,                 (*re-import type M.T0 via module M0*)
    M1,                      (*re-import types M.T0 and M.T1 via module M1*)
    M;                       (*import types M.T0, M.T1 and M.T2 from module M directly *)
END E.
```

A robust implementation must correctly manage *named* types in all possible scenarios, such as when a type is explicitly imported or re-imported via different symbol files (or both), and no matter in which order the various imports occur during the import process. The primary requirement is to ensure that, irrespective of how many times a type is encountered during the import process, it consistently maps to a *single* node in the symbol table of the compiler. For example:

- During compilation of module *C*, the type *M.T0* is re-imported twice: first, when module *M0* is imported and second, when module *M1* is imported. During the second re-import via *M1*, the type *M.T0* is discovered within the object list of module *M,* because it has already been re-imported via module *M0* before. The symbol file of the declaring module *M* is never loaded.

- During compilation of module *D*, the type *M.T0* is first explicitly imported when the symbol file of module *M* is loaded and subsequently re-imported twice: first, when module *M0* is imported and second, when module *M1* is imported. During both re-imports, the type *M.T0* is discovered within the object list of module *M*, because the symbol file of the declaring module *M* has been loaded before, thereby explicitly importing the type *M.T0*.

- During compilation of module *E*, the type *M.T0* is first re-imported twice via modules *M0* and *M1* and subsequently imported from *M* directly. During the second re-import via *M1* and the explicit import from *M*, the type *M.T0* is discovered within the object list of module *M*, because it has already been re-imported via module *M0* before.

Failure to achieve this mapping can lead to incompatibilites during *type checking* (recall that type equality is determined through the equality of pointers referring to the compared type descriptors in the symbol table of the compiler – which in turn is made possible by the fact that the Oberon language definition specifies equivalence of types on the basis of names rather than structure).

When *re-importing* types via other modules, the Project Oberon 2013 implementation already has a built-in mechanism to identify instances where a type to be re-imported is already present in the symbol table of the compiler. This may be the case because the symbol file of the module defining the type has already been loaded, or because the type has already been read when loading other symbol files. The implementation is straightforward and involves a simple search for the type's name within the object list of the declaring module *M* (*ORB.InType*):

```
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
```

This mechanism covers the case where a type is re-imported via other modules (once or several times) without its declaring module ever being explicitly imported, and the case where a type is first explicitly imported and then re-imported via other modules (once or several times).

But it does not cover the case where a type is first re-imported via other modules (once or several times) and *then* explicitly imported from its declaring module (as is the case in module *E* above). Instead, the Project Oberon 2013 implementation enforces a restriction: It prevents explicit imports of modules, from which individual types have previously been re-imported via other modules.

If we want to also allow *explicit* imports *after* prior re-imports of the same type, we could of course employ the same technique that is already used for handling *re-imports*.

```
IF modname[0] # 0X THEN (*re-import*) ...
ELSE (*explicit import*)
  Search the type in the object list of the currently imported module.
  If type is found, map it to the type node of this (previously re-imported) type.
END
```

But this approach would involve searching for *each* explicitly imported type within the object list of the currently imported module *M*. Furthermore, it would require additional modifications in module *ORB* to ensure that the type's name can be accessed in *ORB.InType*.

An alternative approach is to propagate the *reference number* of a type *t* in its *declaring* module *M* across the entire module hierarchy using a new field *t.orgref* as follows *(ORB.InType)*:

- If a type *t* (= *M.T*) declared in module *M* is directly imported by a module *M0*, the field *t.orgref* is set to the reference number of *t* in *M* (read from the symbol file of *M* itself). This effectively marks the beginning of the chain of re-exports and subsequent re-imports of this type[2].

- If *t* is re-exported by *M0*, a reference number for the type *"M.T in M0"* is assigned and written to the symbol file of *M0*, together with its reference number in its declaring module *M* (*t.orgref*).

- If *t* is subsequently *re-imported* by a client module *C* via *M0*, the field *t.orgref* is again set to the reference number of *t* in its *declaring* module *M* (but this time read from the symbol file of *M0*).

- The field *t.orgref* is only written to symbol files if *t* is *re-exported* (otherwise it is implicit). Since re-exports are rare, this has a rather negligible effect on the overall length of symbol files.

With this preparation, our implementation approach can be summarized as follows:

When a module *M* exports a type *t* to a module *M0* and a client module *C* subsequently re-imports this type via *M0*, a module object for its declaring module *M* and a type object for the re-imported type in the object list of *M* is inserted into the compiler's symbol table during compilation of *C*, together with its original reference number (*t.orgref*) in its declaring module *M*.

If the same client *C* later explicitly imports *M*, we start by initializing the compiler's type translation table[3] for module *M* with all types of *M* that have previously been re-imported (by *C*) via other modules, using their own reference numbers in the declaring module *M* as the index (this is why they are propagated). In the above example, these are the types *M.T0* and *M.T1* (*ORB.Import*).

For convenience, we also *mark* each previously re-imported type by temporarily inverting the sign of its module number *t.mno* (this is strictly speaking not necessary, as one could also extract this information from the type translation table in *ORB.InType*, but it leads to a simpler solution). After

[2] *Note that for the re-export mechanism to function, the symbol file of M must be read entirely at least once before initiating the chain of re-exports and subsequent re-imports of individual types of M.*
[3] *The compiler's type translation table (typtab) for a module M is a table containing references to all types that already exist in the object list of M.*

this preparatory step, we can now easily detect, by checking *t.mno*, whether a type *t,* read from the symbol file of *M*, has previously been re-imported via *other* modules. There are two cases:

- If the type *t has* been re-imported via other modules before its direct import from its declaring module *M (t.mno < 0)*, we reuse the type referenced in the type translation table *(typtab)*, while continuing to read the type information from the symbol file of *M*. In the above example, this is the case when module *C* reads the types *M.T0* and *M.T1* from the symbol file of *M*.

- If the type *t* has *not* previously been re-imported via other modules *(t.mno ≥ 0)*, we insert a new type object into the object list of module *M*. This is the regular (and frequent) scenario, as exemplified when module *C* reads the type *M.T2* from the symbol file of *M*.

The following code excerpt shows a possible implementation of this scheme:

ORB.InType:

```
Files.ReadString(R, modname);
IF modname[0] # 0X THEN  (*re-import*) ...
  Files.ReadInt(R, key); Files.ReadString(R, name); Read(R, orgref);
  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.orgref := orgref
  END
ELSE  (*explicit import*)
  IF typtab[ref] # NIL THEN T := typtab[ref] END  (*reuse already re-imported type*)
END
```

ORB.Import:

```
thismod := ThisModule(modid, modid1, TRUE, key); obj := thismod.dsc;
WHILE obj # NIL DO  (*initialize typtab with already re-imported types*)
  obj.type.mno := -obj.type.mno;  (*mark type as re-imported*)
  typtab[obj.type.orgref] := obj.type;
  obj := obj.next
END ;
...
Read(R, class);
WHILE class # 0 DO
  Files.ReadString(R, name); InType(R, thismod, t);
  IF t.mno < 0 THEN t.mno := -t.mno  (*type already re-imported via other modules*)
  ELSE NEW(obj);  (*insert a new object into the object list of thismod*)
    obj.class := class; obj.name := name; obj.type := t; obj.lev := -thismod.lev;
    IF class = Const THEN ...
    ELSIF class = Var THEN ...
    ELSIF t.typobj = NIL THEN t.typobj := obj
    END ;
    obj.next := thismod.dsc; thismod.dsc := obj
  END ;
  Read(R, class)
END
```

Handle alias type names among imported and re-imported modules correctly

Special care must be taken to handle cases in which *imported* or *re-imported* types have alias names associated with them. Consider the following scenario:

```
MODULE M;
  TYPE A* = RECORD END ;   (*exported original type*)
    B = RECORD END ;       (*non-exported original type, B is considered local in M*)
    C* = A;                (*alias type of an exported original type, will be exported as M.A*)
    D* = B;                (*alias type of a non-exported original type, will be exported as M.D*)
END M.

MODULE M1;
  IMPORT M;
  TYPE E* = M.C;       (*local alias type for an imported alias type M.C, which is itself an alias type for M.A*)
    F* = M.D;          (*local alias type for an imported type M.D, which was not imported as an alias type*)
  VAR c*: M.C;         (*the original type M.A is re-exported, not the imported alias type M.C*)
    e*: E;             (*the original type M.A is re-exported, not the local alias type E*)
END M1.

MODULE C;
  IMPORT M1, M;        (*only the original type name M.A is re-imported, when M1 is imported*)
  VAR c*: M.C;         (*the alias type M.C is directly imported from M*)
END C.
```

In our implementation, the following measures are taken to handle *re-imported* alias types:

- *An imported alias type name is re-exported under its original type name, if it is available in the re-exporting module*. In the above example, when *M1* re-exports the imported alias type *M.C*, it writes the *original* imported type name *M.A* to the symbol file of *M1*, because it is also available in *M1*. But when *M1* re-exports the imported type *M.D*, it writes the type name *M.D* itself to the symbol file of *M1*, because the original type name *M.B* is not available in *M1*. Alias type names can always be *directly* imported from the module in which they have been defined. In the above example, the client module *C* directly imports the alias type name *M.C* from *M*.

- *Exported original types are always written to symbol files before any exported alias types*. This ensures that when a type *t* is first re-imported and then also explicitly imported, no type node is created for the very *first* occurrence of the type *t* in the symbol file of its declaring module *M*, i.e., the original type. Any subsequent occurrences of *t* in the symbol file of *M* must be alias types and will (correctly) lead to the creation of new type nodes within the symbol table.

- *Alias types must always refer to their original type objects in the symbol table of the compiler*. This is manifest by the presence of a back-pointer *t.typobj* in each named type which *always* points to the original type object, *regardless* of whether the alias type is defined in the module currently being compiled or in an imported (or even re-imported) module.

In the Project Oberon 2013 implementation, the last criteria is only met for alias types declared in the module currently being compiled, but not for *imported* alias types. For example, compilation of module *M2* below leads to a compile time error:

```
MODULE M;
  TYPE A* = RECORD END ;
    B* = A;            (*alias type*)
```

```
END M.

MODULE M1;
  IMPORT M;              (*import type M.A and alias type M.B*)
    VAR a*: M.A;         (*re-export type M.A*)
      b*: M.B;           (*type name M.B is not re-exported, since it is just an alias type of M.A*)
END M1.

MODULE M2;
  IMPORT M1, M;          (*first re-import type M.A via M1 and then directly import type M.B from M*)
  VAR c: M.A;            (*compile time error in Project Oberon 2013 if the explicit import of M were allowed*)
    d: M.B;
END M2.
```

The first reason is that the explicit import of module *M* (from which a type *M.A* has previously been re-imported via *M1*) in *M2* is not allowed in Project Oberon 2013 (“invalid import order”). But even if it were allowed, it would lead to an “undef” compile time error when processing the declaration of the global variable *c*. The reason is that during the explicit import of module *M* in *M1*, the imported alias type *M.B* is (correctly) recognized as being the same as the previously imported type *M.A*, but the assignment *obj.type.typobj := obj* in *ORB.Import* makes the back-pointer for the type *M.B* (which is the same as type *M.A*) point to the type object of the just imported *alias* type *M.B* instead of leaving it as is, i.e. pointing to the *original* type object of *M.A*.

Consequently, when the imported original type *M.A* is later *re-exported* by *M1*, the alias type name *M.B* is written to the symbol file of *M1* instead of the original type name *M.A*, effectively making *M.A* unavailable to clients of *M1*.

To solve this issue, we have replaced the following code in *ORB.Import*:

```
t.typobj := obj
```

with:

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

which establishes the *typobj* back-pointer only if it doesn't exist yet. This ensures that an imported alias type always points to the *original* imported type object, not another imported alias type object.

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 object (no matter whether the original type is imported from another module or declared in the module currently being compiled).

Propagate imported export numbers of type descriptor addresses to client modules

We replaced the following code in *ORB.OutType* from the *Project Oberon 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*) are re-exported to client modules correctly, 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, as only *explicitly* imported types can be referred to *by name* in clients. However, our implementation *allows* explicitly importing a module *after* types of the module have previously been re-imported via other modules. In such a case, the previously re-imported module is *converted* to an explicitly imported one in the compiler's symbol table (see *ORB.ThisModule*). Thus, we must now make sure that if a type *M.T* is exported by the declaring module *M* and re-exported by another module *M1*, the export number of the type descriptor address in the *declaring* module *M* is propagated across the entire module hierarchy.

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

In Project Oberon 2013, compilation of modules *M1* and *M2* below leads to a name conflict with the re-imported module name *M*:

```
MODULE M;
  TYPE T* = RECORD (*…*) END ;
END M.

MODULE M0;
  IMPORT M;
  VAR t*: M.T;                (*re-export type M.T*)
END M0.

MODULE M1;
  IMPORT M0;                  (*re-import type M.T*)
  VAR M: INTEGER;             (*name conflict with globally declared identifier in Project Oberon 2013*)
END M1.

MODULE M2;
  IMPORT M := M0;             (*name conflict with module alias name in Project Oberon 2013*)
END M2.
```

Since the indirectly imported module *M* cannot be referred to *by name* in in modules *M1* and *M2*, it *should* not lead to name conflicts there. We have therefore decided to *hide* module names, which are only re-imported (but not explicitly imported) from the global namespace during compilation. This allows them to coexist with other identifiers of the same name in the importing module, i.e.

- *Globally* declared identifiers of the importing module will not interfere with re-imported modules.
- *Module* alias names of explicitly imported modules can be used without conflict.

This is implemented by *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 guards of various loops in procedures *ORB.NewObj*, *ORB.thisObj* and *ORB.ThisModule*.

Of course, if a re-imported module *M* is subsequently also explicitly imported, it will be converted to an explicitly imported one in the compiler's symbol table (the field *obj.rdo* will be set to TRUE).

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.

Interpretation #1:

- *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*

In our implementation, we have adopted this interpretation. It implies that the statement *IMPORT M := M1* imports module *M1* and associates the local name *M* with it, i.e. the identifier *M* is used as usual, but the file with name *M1* is read. It also implies that an imported object *x* can be accessed only via a *single* qualified identifier *M.x* and allows reusing the original module name *M1*.

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; END B1.
```

This is implemented by *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 combination *obj.orgname = name* allows the import M := M1 in:

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

Not checking the combination *obj.name = orgname* allows the imports *M := M0* and *X := M0* in:

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

Alternative interpretation #2:

- *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 alternative interpretation implies that an imported object *x* can be accessed through *multiple* identifiers *M.x* and *M1.x*, and does *not* allow reusing the original module name *M1* (i.e. *M1* remains valid). This is similar to an *alias type* in the Oberon language, where 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 interpretation #2 for the following reasons:

- Acces*sing* the same imported object through several *different* identifiers may be confusing and is considered bad programming style. If used at all, a single module alias should suffice.
- It disallows the sometimes useful ability to "swap" two modules by writing *M0 := M1, M1 := M0*.
- The already rather complex symbol table data structure in the compiler would become 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.

Write the module anchor of re-exported types before the type description to the symbol file

When re-exporting a type, our implementation writes the module anchor (name and key) and the name of the re-exported type immediately *after* the type's reference number to the symbol file, but *before* the remaining type description instead of *after* it as in Project Oberon 2013 *(ORB.InType)*.

This ensures that the code *also* works in the presence of *cyclic* references *among re-imported types*. Recall that a named type may refer to itself either directly or indirectly, either through record fields or through type *extensions*, as shown in the following example:

```
MODULE M;
  TYPE P1* = POINTER TO R1;
    P2* = POINTER TO R2;
    P3* = POINTER TO R3;
    R1* = RECORD p2*: P2 END;
    R2* = RECORD p1*: P1 END;       (*cyclic reference through record fields*)
    R3* = RECORD (R1) END;          (*cyclic reference through type extensions*)
END M.
```

Consider the case where *M* is re-imported by a module *C* indirectly via another module *M1*. In this situation, procedure *ORB.InType* is recursively called for the re-imported types *P1, R1, P2, R2* and *R1* (in this order) and the *last* call to *ORB.InType(R, thismod, t.base)* – made when reading the re-imported type *R2* – sets *t.base* to *R1* one recursion level up via its variable parameter *T*:

```
PROCEDURE InType(VAR R: Files.Rider; thismod: Object; VAR T: Type);
```

If the module anchor of *M* were stored in the symbol file of *M after* the type description of *R1* (as in Project Oberon 2013), the code to read the module anchor of *M* would also be executed *after* the recursive calls to *ORB.InType* for the re-imported types *P1, R1, P2, R2* and *R1*:

```
PROCEDURE InType(VAR R: Files.Rider; thismod: Object; VAR T: Type);
  ...
BEGIN Read(R, ref);
  IF ref < 0 THEN T := typtab[-ref]
  ELSE NEW(t); T := t;
    ...
    InType(R, thismod, t.base);          (*recursive call to InType modifies T one level up*)
    ...
    (*code to read module anchor at the end of ORB.InType*)
    Files.ReadString(R, modname);
    IF modname[0] # 0X THEN (*re-import*) ...
    ELSIF typtab[ref] # NIL THEN T := typtab[ref]   (*already re-imported*)
    END ;
    T := typtab[ref]                     (*assignment to T after the recursive calls to InType*)
    ...
```

But the code to handle re-imports may *change* the field *t.base* one level up in the recursion to an entry in the type translation table *(typtab)* via the variable parameter *T* of *ORB.InType*. Wonders of nested recursions and cyclic references among imported (or re-imported) record and pointer base types – a little subtle indeed. If the reader remains unconvinced, s/he should prepare to explore a realm of subtle intricacies ☺ In our implementation, we chose to be on the safe side by making all assignments to *T before* any recursive calls to *ORB.InType* are initiated, as shown below:

```
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; ...
    IF ref > 0 THEN  (*named type*)
      (*code to read module anchor at the beginning of ORB.InType*)
      Files.ReadString(R, modname);
      IF modname[0] # 0X THEN (*re-import*) ...
      ELSIF typtab[ref] # NIL THEN T := typtab[ref]   (*already re-imported*)
      END ;
      typtab[ref] := T
    END ;
    Read(R, form); t.form := form;
    ...
    InType(R, thismod, t.base);          (*no assignments to T after the recursive calls to InType *)
    ...
```

This effectively establishes the data structure for re-imported module and type objects in the symbol table *prior* to entering any recursion of *ORB.InType*. The variable parameter *T* is set to the returned type, while the local variable *t* acts as a temporary holder for data read from the currently loaded symbol file. They only differ when encountering a type that has already been read from *another* symbol file. In such cases, the subtree rooted in *t* is discarded upon re-reading the type

* * *

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