Streamlining symbol files in Oberon

Andreas Pirklbauer

1.4.2024

Overview

This technical note presents both a simplification and improvement of the handling of import and export[1] for the *Project Oberon 2013* system, which is a reimplementation of the original *Oberon* operating system on an FPGA development board in 2013, as published at *www.projectoberon.com.* If you use the *Extended Oberon*[2] system, the improvements presented in this document are already implemented by default.

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 1970s 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. *Re-export conditions*: Imported types may be *re-exported* and their *imports* may be hidden.
2. *Recursive data structures*: Pointer declarations may *forward reference* a record type.
3. *Cyclic references:* Record and pointer types may contain cyclic references among themselves.
4. *Module alias names*: A module can be imported under a different (alias) name.
5. *Hidden record fields:* Offsets of non-exported pointer fields are needed for garbage collection;
   offsets of non-exported procedure variable fields are needed for module reference checking.

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 [1-14]. 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 *postorder* traversal [4, 5, 7].
- Oberon compilers in the 1990s used either a *postorder* or *preorder* traversal of the symbol table [8-12].
- The Oberon on ARM compiler in 2008 used a *fixup* technique for types in symbol files [13].
- The Project Oberon 2013 compiler uses *preorder* traversal and a *fixup* technique for types [14].

As with the underlying languages, all these re-implementations and refinements of the handling of import and export (and the associated 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.

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. 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 [13]:

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

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 excerpt above correctly states that types must always be defined before they are referenced during compilation. Consequently, one might infer the need for fixups to handle forward or cyclic references. However, an alternative method, which is outlined in [9] and [12] and implemented in Project Oberon 2013, arranges the symbol file such that a unique type reference consistently *precedes* the corresponding type description. This is always possible, even in the presence of forward or cyclic references. The basic idea is that the linear description of an *enclosing* object *P* may open a *bracket* to contain the complete description of a *component* object *R* (for example a referenced record of a pointer type, a record field of a record type or a parameter of a procedure type). The description of the enclosing object *P* then resumes after closing the bracket. Importantly, if the description of *R* refers back to the enclosing object *P*, the reference number of *P* is used, thereby resolving the cyclic reference.

The example above of a forward pointer declaration is now linearized as follows, without any fixups:

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

Here, the pointer declaration *P* obtains a reference number #14 which precedes its actual type description, while the record declaration *R* with reference number #15 is *embedded* in the linear description of *P*. This eliminates the need for fixups. *R* refers back to the description of *P* using reference number #14.

As a result of this approach, all type references within symbol files inherently take the form of *backward* references. This design choice also simplifies the process of reconstructing the symbol table data structure during import, and makes it straightforward to ensure that types are always *defined* (i.e. inserted into the symbol table) before they are *referenced* (see *ORB.OutType* and *ORB.InType*):

```
PROCEDURE OutType(VAR R: Files.Rider; t: Type);
  ...
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); ...
    ELSIF t.form = Record THEN OutType(R, t.base); ...
    ELSIF t.form = Proc THEN OutType(R, t.base); ...
    END
  END ;
  ...
END OutType;
```

```
PROCEDURE InType(VAR R: Files.Rider; thismod: Object; VAR T: Type);
  ...
BEGIN Read(R, ref);
  IF ref < 0 THEN T := typtab[-ref]  (*type was already read*)
  ELSE NEW(t); T := t; typtab[ref] := t; ...
    Read(R, form); t.form := form;
    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;
```

One can easily verify that in Project Oberon 2013 types are *always* already “fixed up” with the right value by slightly modifying procedure *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 error message will *never* be printed when importing a module. A formal proof can of course easily be constructed. It rests on the observation that a type’s *reference* number is written to the symbol file *before* the remaining type description and also before any other types or objects refer to this type.

Code that can be omitted 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
  ...
  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 ;
  ...
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;
```

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

In Project Oberon 2013, compilation of module *C* below leads to a compilation error.

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

MODULE M0;
  IMPORT M;                 (*import type M.A and type alias name M.B*)
  VAR a*: M.A;              (*re-export type M.A, but the type name M.B is (incorrectly) written in PO 2013*)
END M0.

MODULE C;
  IMPORT M0, M;             (*first re-import type M.A via M0 and then directly import type M.B from M*)
  VAR c: M.A;               (*compile time error in Project Oberon 2013 if explicit import of M were allowed*)
END C.
```

The first reason is that the explicit import of module *M* (from which the type *M.A* has previously been re-imported via *M0*) in module *C* is not allowed in Project Oberon 2013 (“invalid import order”).

But even if it *were* allowed, it would lead to an “undef” compilation error when processing the declaration of the global variable *c*. The reason is that during the explicit import of module *M* in *M0*, the imported type alias name *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 *M.B* point to the type object of the just imported type alias name *M.B* instead of leaving it as pointing to the *original* type object of *M.A*.

Consequently, when the imported original type *M.A* is later *re-exported* by *M0*, the type alias name *M.B* is written to the symbol file of *M0* instead of *M.A*, effectively making the original type name *M.A* unavailable to clients of *M0* (such as module *C*).

To solve this issue, it suffices to replace 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 type alias name always points to the *original* imported type object, not another imported type alias object. This is the same type of precaution as in *ORP.Declarations*, where type aliases declared in the module being compiled are initialized as follows:

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

which also makes sure that an type alias name 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).

As this example shows, special care must be taken to handle cases where *imported* or *re-imported* types have type alias names associated with them. Consider the following scenario:

```
MODULE M;
  TYPE A* = RECORD END ;         (*exported original type*)
    B = A;                       (*non-exported type alias name of an exported original type A*)
    C* = B;                      (*exported type alias name of (another alias of) an exported original type A*)
    D = RECORD END ;             (*non-exported original type, considered local to module M*)
    (*E* = D;*)                  (*export of E not allowed, because the original type D is not exported*)
END M.

MODULE M0;
  IMPORT M;
  TYPE F* = M.C;                 (*type alias name of an imported type alias name M.C, resolves to M.A*)
  VAR c*: M.C;                   (*original type M.A re-exported, not the type alias name M.C*)
    f*: F;                       (*original type M.A re-exported, as well as the type alias name F*)
END M0.

MODULE C;
  IMPORT M0, M;                  (*original type M.A re-imported via M0 and explicitly imported from M*)
  VAR c*: M.C;                   (*type alias name M.C of an original type M.A directly imported from M*)
END C.
```

In our implementation, type alias names are handled as follows:

- A *type alias name* is just an additional *name* for an existing type; it does *not* define a new type. A type can have multiple alias names. If an alias name refers to another alias, it resolves to the original type.

- If the original type is declared in the same module as a type alias name, the type alias name can only be exported if the original type is also exported. In the above example, exporting the type alias name *C* in *M* is allowed because the *original* type *M.A* is an exported global type[3]. However, the type alias declaration *E* = D* in *M* would not be allowed, because the original type *M.D* is not exported.

- If the original type is an imported type, it will be *re-exported* as a result of a type alias declaration. For example, exporting the type alias name *F* in *M0* causes the original type *M.A* to be re-exported by *M0*.

- When a type is *re-exported*, its *original* type name will be included in the symbol file of the re-exporting module. However, this name cannot be used in clients unless the client also *explicitly* imports it from the module in which it is defined (a re-exported original type name only serves to uniquely identify the re-exported type and its declaring module, in order to ensure that multiple occurrences of the same type during the import process are mapped to the same type node within the compiler's symbol table).

- Original type names and type alias names can always be *directly* imported from the modules in which they are defined. For example, module *C* directly imports the type alias name *M.C* from *M*. But when *C* exports the global variable *c,* the original type *M.A* will be re-exported, not the alias type name *M.C*.

- Exported original types are always written to symbol files *before* any of their type alias names. This ensures that when a named type *M.A* is first re-imported via an intermediate module *M0* and then also explicitly imported by a client *C*, the *first* occurrence of *M.A* in the symbol file of *M* is identified as the original type (in which case no new object node needs to be created).

- Any subsequent occurrences of *named* types referring to an original type in the symbol file of *M* are then identified as *type alias names* and will (correctly) lead to the creation of new object nodes (one for each alias name) in the symbol table of the compiler.

- Type alias names must always refer to their *original* type objects in the symbol table of the compiler in order to avoid incompatibilites during *type checking*, regardless of whether the original type has been declared in the module currently being compiled or in an imported or re-imported module.

[3] *In Oberon, only global types can be exported.*

In passing, we note the following general *export rules*, as realized in our implementation:

- Type alias names can only be exported if the *original* type is exported or is itself imported (see above).
- Array types or variables of an array type can only be exported if the array *base* type is exported.
- Record fields can only be exported if the type of the *field* and the type of the *record* are exported.
- Extended record types *can* be exported even if its *base* type is not exported.
- Pointer types or pointer variables *can* be exported even if the pointer base type is not exported.
- Parameters of exported procedures must be of exported data types.
- Anonymous variables cannot be exported.
- String constants can be exported as read-only (Extended Oberon only).

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.
```

To solve this issue, we *hide* module names which are only re-imported, but not explicitly imported, from the global namespace. This allows them to coexist with global identifiers of the importing module and module alias names of explicitly imported modules. This is implemented by *skipping* over re-imports (identified as *~obj.rdo*) in various loops that traverse the list of identifiers rooted in the global variable *ORB.topScope*:

```
PROCEDURE NewObj*(VAR obj: Object; id: ORS.Ident; class: INTEGER);
  VAR new, x: Object;
BEGIN x := topScope;
  WHILE (x.next # NIL) & ((x.next.name # id) OR (x.next.class = Mod) & ~x.next.rdo) DO
    x := x.next
  END ;
  ...

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

PROCEDURE ThisModule(name, orgname: ORS.Ident; decl: BOOLEAN; key: LONGINT): Object;
  VAR mod: Module; obj, obj1: Object;
BEGIN obj1 := topScope;
  IF decl THEN obj := obj1.next;   (*search for alias, obj.class = Mod implicit*)
    WHILE (obj # NIL) & ((obj.name # name) OR ~obj.rdo) DO obj := obj.next END
    ...
```

If a re-imported module is later also explicitly imported, it is converted to an explicitly imported one in the symbol table, to ensure that its name can no longer co-exist with module alias names or global identifiers.

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, for example:

Interpretation #1:

- *It is module M1 that is imported, not M*
- *The module alias name M renames module M1 (i.e. the original name M1 can be reused)*
- *A module can only have a single module alias name*

Interpretation #2:

- *It is module M1 that is imported, not M*
- *A module alias name M is just an additional name for M1 (i.e. the original name M1 cannot be reused)*
- *A module can have multiple module alias names*

In our implementation, we have adopted interpretation #1. This 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* denotes an existing module in the module list.

Not checking the combination *obj.orgname = name* allows the second import *M1 := M2* with *name = M1* in module *A1* (when processing this import, we skip over the first import *M0 := M1* with *obj.orgname = M1*).

Not checking the combination *obj.name = orgname* allows the second import *M1 := M0* with *orgname = M0* in module *A2* (when processing this import, we skip over the first import *M0 := M1* with *obj.name = M0)*.

This leaves us with checking the *other* two combinations *obj.name # name* and *obj.orgname # orgname*:

```
PROCEDURE ThisModule(name, orgname: ORS.Ident; decl: BOOLEAN; key: LONGINT): Object;
  VAR mod: Module; obj, obj1: Object;
BEGIN obj1 := topScope;
  IF decl THEN  (*explicit import by declaration*)
    obj := obj1.next;  (*search for alias*)
    WHILE (obj # NIL) & ((obj.name # name) OR ~obj.rdo) DO obj := obj.next END
  ELSE obj := NIL
  END ;
  IF obj = NIL THEN obj1 := obj1.next;  (*search for module*)
    WHILE (obj # NIL) & (obj.orgname # orgname) DO obj1 := obj; obj := obj1.next END;
    IF obj = NIL THEN (*insert new module*) ...
    ELSE (*module already present*) ...
    END
  ELSE ORS.Mark("mult def")
  END ;
  RETURN obj
END ThisModule;
```

Interpretation #2 would imply that an imported object *x* can be accessed through *multiple* identifiers *M.x* and *M1.x*, and would *not* allow reusing the original module name *M1* (i.e. it remains valid). This is similar to a *type alias name*, where a type can have additional names and the original type name remains valid. Under this definition, modules *A1-A3* above would be illegal, whereas module *B1* would become legal.

We have decided *not* to adopt interpretation #2 for the following reasons:

- Accessing the same imported object through several *different* identifiers is confusing and is considered bad programming style. If used at all, a single module alias name 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 would become even more complex due to the need to having to manage a *list* of alias objects. We believe this to be unnecessary complexity.

Propagate imported export numbers of type descriptor addresses to client modules

The Project Oberon 2013 implementation does not allow type tests or type guards on *re-imported* types. To make them possible, we have replaced the following code in *ORB.OutType*:

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

with this code:

```
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 or type guards on such types when they are subsequently re-imported.

The reader may think that this change is unnecessary, since one cannot perform a type test on re-imported types (only *explicitly* imported types can be referred to *by name* in client modules).

But our implementation allows importing a module *M* even after its types of *M* have previously been re-imported via other modules, as explained in the next section. In such cases, the previously re-imported module is simply *converted* to an explicitly imported one in the compiler's symbol table. This enables type tests on all explicitly imported types (see *ORB.ThisModule*). Therefore, it's important to ensure that if a type is exported by a module *M* and subsequently re-exported by another module, the export number of its type descriptor address in its declaring module *M* is propagated correctly from the beginning.

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 *T* from one module *(M)* 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.

Two common approaches to implement this 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 types M.T0, M.T1 and M.T2 from M directly*)
  VAR t0*: M.T0;            (*re-export type M.T0 to clients of M0*)
END M0.

MODULE M1;
  IMPORT M;                 (*import types M.T0, M.T1 and M.T2 from M directly*)
  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 M0*)
    M1;                     (*re-import types M.T0 and M.T1 via 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 M directly*)
    M0,                     (*re-import type M.T0 via M0*)
    M1;                     (*re-import types M.T0 and M.T1 via M1*)
END D.

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

A robust implementation must correctly handle *named* types in every possible scenario, including explicit imports and re-imports from multiple symbol files, or any combination thereof. The principal requirement is to ensure that, regardless of how many times a particular type is encountered during the import process and irrespective of the order of the various (re-)imports of this type, it consistently maps to a *single* node in the symbol table of the compiler. Failure to achieve this can lead to incompatibilites during *type checking*. Recall that in typical implementations, type equality is determined by comparing pointers referencing type descriptors in the symbol table. This is made possible by the Oberon language definition, which specifies equivalence of types on the basis of names rather than structure.

In the above example, the type *M.T0* is imported and re-imported as follows:

- 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 direct 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.

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

But it does not cover the case where a type is *first* re-imported via other modules (once or several times) and *subsequently* explicitly imported from its declaring module (as 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 the
  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 consists of propagating the *original* reference number of each re-exported type *t* across the module hierarchy. This eliminates the need to search for each explicitly imported type within the compiler's symbol table. Instead, the original reference number, denoted as *t.orgref*, is used to initialize the compiler's *type translation table*[4] for the module prior to its explicit import. It is propagated as follows:

- 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 the declaring module *M*). This effectively marks the beginning of the chain of re-exports and subsequent re-imports of this particular type[5].

[4] *The compiler's type translation table (typtab) for a module M is a table containing references to all types of M that already exist in the object list of M.*
[5] *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 re-imports of individual types of M.*

- If *t* is re-exported by *M0*, a reference number for the type *"M.T in M0"* is assigned *(t.ref)* 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 module *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 the re-exporting module *M0*).

Note that the field *t.orgref* is only written to symbol files if the type *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 an intermediate module *M0* and a client module *E* 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 *E*, together with its original reference number (*t.orgref*) in its declaring module *M*.

- If the same client *E* later also *explicitly* imports *M*, we start by initializing the compiler's type translation table for *M* with all types of *M* that have previously been re-imported via other modules, using the original reference numbers in their 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*.

- For convenience, we also *mark* each previously re-imported type (e.g., by temporarily inverting the sign of its module number) during this initialization phase. This will allow us to easily detect, whether a type read from the symbol file of *M* has previously been re-imported via *other* modules.

- If module *E* then reads a type *t* from the symbol file of *M*, there are two cases:

  – If the type *t has* previously been re-imported via other modules, we reuse the already existing type, while continuing to read the type information of *t* from the symbol file of *M*. In the above example, this is the case when module *E* reads the types *M.T0* and *M.T1* from the symbol file of *M*. Note that only *named* types can ever be re-exported. Since named types are written to symbol files before variables and procedures that may refer to them, we know that the object *class* must be *Typ* in this case and therefore no additional data needs to be read from the symbol file of *M*.

  – If the type *t* has *not* been re-imported via other modules before, we create and insert a new type object for *t* into the object list of *M*. In the above example, this is the case when module *E* reads the type *M.T2* from the symbol file of *M*. This is the regular (and frequent) case.

The following code excerpts show 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);
FOR i := Record+1 TO maxTypTab-1 DO typtab[i] := NIL END ;
obj := thismod.dsc;  (*initialize typtab with already re-imported types*)
```

```
  WHILE obj # NIL DO
    typtab[obj.type.orgref] := obj.type;  (*initialize typtab*)
    obj.type.mno := -obj.type.mno;  (*mark type as re-imported*)
    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
```

In passing, we note that Project Oberon 2013 allows a maximum of 15 modules to be imported by any single module. This typically doesn't pose any issues, as it aligns with the good programming practice of structuring the module hierarchy in a way that only a small number of modules are imported.

However, this upper limit also includes modules from which types are (only indirectly) *re-imported*. These modules don't necessarily have to be explicitly listed in the import statement; their imports can remain hidden. Therefore, in deep module hierarchies, there may arise a desire to lift this restriction. To address this need, our implementation increases the maximum number of modules that can be directly or indirectly imported from 15 to 63, providing greater flexibility in managing complex module structures.

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

When implementing the re-export mechanism through *self-contained symbol files*, it is essential to include in the type description a reference to the module in which a re-exported type was originally defined. This reference, known as the *module anchor*, typically includes the module name and key of the respective module. When combined with the type's name, it forms a unique *identifier* for the re-exported type[6].

In contrast to Project Oberon 2013, our implementation writes this unique identifier immediately *after* the type's reference number, but *before* the actual type description to the symbol file of the re-exporting module. This approach corresponds to postulate #5 in [5]. It guarantees that no other types can appear between the type's *reference* number and its *identification*, thereby ensuring that the code also works in the presence of cyclic references *among re-imported types*.

Recall that a type may refer to itself, as illustrated 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) p3*: P3 END ;           (*cyclic reference through type extensions*)
  END M.
```

Consider the case where the types defined in module *M* are re-imported by a client module *C* via an intermediate module *M0*. 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 type *R2*, sets *t.base* to *R1* via its variable parameter *T*.

[6] *For types declared in the currently compiled module, there is no need for an explicit module anchor, as it is implicitly handled by the export mechanism.*

If the module anchor of the declaring module *M* were stored in the symbol file of the re-exporting module *M0 after* the type description of *R1* (as in Project Oberon 2013), the code to read this module anchor 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);
  (*Project Oberon 2013 implementation*) ...
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, changes t.base*)
    ...
    (*code to read the module anchor*)
    Files.ReadString(R, modname);
    IF modname[0] # 0X THEN (*re-import*)
      ...
      T := obj.type;                                 (*changes t.base one level up in the recursion*)
      ...
    END
  END
END InType;
```

Note that the code handling re-imports may *change* the field *t.base* one level up in the recursion via the variable parameter *T* to an entry in the compiler's type translation table or an existing entry in the object list of the declaring module *M*. While this doesn't actually pose an issue, as the code discards the subtree rooted in *t* if it detects that this type has previously been read from another symbol file, our preference is to establish the data structure for re-imported module and type objects in the compiler's symbol table *before* entering any recursion. This prevents *T* from temporarily holding an incorrect value and allows building the data structure for re-imported modules in the compiler's symbol table *before* any of their types are added.

In our implementation, we have therefore decided to move the code to read and write the module anchor of re-imported types to the beginning of procedures *ORB.InType* and *ORB.OutType*, e.g.,

```
PROCEDURE InType(VAR R: Files.Rider; thismod: Object; VAR T: Type);
  (*Extended Oberon implementation*)...
BEGIN Read(R, ref);
  IF ref < 0 THEN T := typtab[-ref]  (*already read*)
  ELSE NEW(t); T := t; t.mno := thismod.lev; t.orgref := ref;
    IF ref > 0 THEN  (*named type*)
      (*code to read the module anchor*)
      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 NEW(obj); (*insert new type object in object list of mod*)
          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
      ELSIF typtab[ref] # NIL THEN T := typtab[ref]  (*already re-imported*)
      END ;
      typtab[ref] := T
    END ;
    Read(R, form); t.form := form;
    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;
```

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. Crelier R. *Separate Compilation and Module Extension*. ETH Zürich Dissertation No. 10650 (1994).
13. Wirth N. *An Oberon Compiler for the ARM Processor*. Technical note (December 2007, April 2008), www.inf.ethz.ch/personal/wirth
14. Wirth N., Gutknecht J. *Project Oberon 2013 Edition*, www.inf.ethz.ch/personal/wirth