Streamlining symbol files in Oberon

Andreas Pirklbauer

1.5.2024

This technical note presents a simplification and improvement of the handling of import and export[1] for the *Project Oberon 2013* system (*www.projectoberon.com*), as implemented in *Extended Oberon*.

Code that can be omitted in Project Oberon 2013

Project Oberon 2013 arranges the symbol file such that a unique type reference consistently *precedes* the corresponding type description. 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. As a result of this approach, all type references within symbol files inherently take the form of *backward* references and the following code (in red) in *ORB.Import* and *ORB.Export* 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;
```

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

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;
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;              (*compilation 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* in module *C* is not allowed in Project Oberon 2013, as module *C* has previously re-imported the type *M.A* via *M0*, resulting in an *invalid import order* error.

But even if permitted, this scenario would trigger a compilation error during compilation of module *C* due to the unknown type *M.A* when processing the declaration of the global variable *c*. The issue arises from the import process in module *M0*, where the imported type alias *M.B* is correctly identified as the previously re-imported type *M.A*. However, the subsequent assignment *obj.type.typobj := obj* in *ORB.Import* incorrectly redirects the back-pointer for *M.B* to the newly imported type alias object *M.B* instead of leaving it as pointing to *M.A*. Consequently, when *M.A* is re-exported by module *M0*, it is misrepresented as *M.B* in the symbol file of *M0*, rendering the original type *M.A* inaccessible to clients like 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 follow:

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

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* re-imported modules from the global namespace, allowing them to coexist with global identifiers and module alias names of explicitly imported modules:

```
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
    ...
```

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

The Oberon language report defines aliased module imports 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*. In our implementation, we have adopted the following interpretation of this definition:

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

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 combinations *obj.orgname # name* and *obj.name # orgname*, where *obj* denotes an existing module in the module list of the symbol table. Not checking the combination *obj.orgname = name* allows the second import *M1 := M2* with *name = M1* in module *A1*. Not checking the combination *obj.name = orgname* allows the second import *M1 := M0* with *orgname = M0* in module *A2*.

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*)
        IF decl THEN (*explicit import by declaration*)
          IF obj.rdo THEN ORS.Mark("mult def")
          ELSE obj.name := name; obj.rdo := TRUE  (*convert obj to explicit import*)
          END
        END
      END
    ELSE ORS.Mark("mult def")
    END ;
    RETURN obj
  END ThisModule;
```

Propagate imported export numbers of type descriptor addresses to client modules

The Project Oberon 2013 implementation does not support type tests or type guards on types re-imported solely via other modules. This doesn't pose an issue since only explicitly imported types can be referenced by name in client modules anyway.

But our implementation allows explicitly importing a module *M* even after types of *M* have previously been re-imported, as outlined in the next section. In such cases, the previously re-imported module is *converted* to an explicitly imported one in the compiler's symbol table. To enable type tests and type guards on types declared in such converted modules, 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:

```
  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, thereby enabling type tests and type guards on such types.

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*. Project Oberon 2013 has chosen *self-contained symbol files* to implement the re-export mechanism. But it does not allow an explicit import of a module *M* after types of *M* have previously been re-imported via other modules.

Our implementation removes this limitation by propagating the *original* reference number of a re-exported type *t*, denoted as *t.orgref*, across the module hierarchy and by using this reference number to initialize the compiler's *type translation table*[2] for the module prior to its explicit import. It can be summarized as follows:

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

[2] *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.*

- If the same client *C* later also *explicitly* imports module *M*, we start by initializing the compiler's type translation table for module *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).

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

  Case A: 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*. Since named types are written to symbol files *before* variables and procedures that might 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*.

  Case B: If the type *t* has *not* previously been re-imported via other modules, we create and insert a new type object for *t* into the object list of module *M*. This is the regular (and frequent) case.

The following code excerpts show a possible implementation of this scheme:

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 new type object in 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
```

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
```

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. Our implementation writes this reference and the type name immediately *after* its reference number, but *before* the type description to the symbol file of the re-exporting module. Recall that a type may refer to itself:

```
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 types defined in *M* are re-imported via an intermediate module. In this situation, procedure *ORB.InType* of Project Oberon 2013 is recursively called for the *re-imported* types as follows:

```
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);
    ...
    Files.ReadString(R, modname);  (*code to read the module anchor*)
    IF modname[0] # 0X THEN  (*re-import*)
      ...
      T := obj.type;                        (*changes t.base one level up in the recursion*)
      ...
   END
  END
END InType;
```

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

```
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; t.mno := thismod.lev; t.orgref := ref;
    IF ref > 0 THEN  (*named type*)
      Files.ReadString(R, modname); (*code to read the module anchor*)
      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 ...
    END
  END
END InType;
```