Safe module unloading and module finalization

in the Oberon operating system

Andreas Pirklbauer

31.12.2020

Overview

Most implementations of the Oberon[1] operating system only check for *client* references prior to module unloading. Other types of module references are usually not checked, although various approaches are typically employed to alleviate the situation where such references do exist. In addition, the unloading of *groups* of modules with (potentially cyclic) references only among themselves and *module finalization* are usually not supported. See the appendix for historical notes on module unloading in the Oberon operating system.

Checking only some, but not all references to a module prior to unloading it leaves a system in an *unsafe* state, which may – and generally will – become *unstable* the moment another module loaded later overwrites a previously released module block.

This technical note describes an implementation of *completely safe* unloading of modules and module groups and *module finalization*, as realized in *Extended Oberon*[2], 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*.

Module references in the Oberon system

An Oberon module can be viewed as a container of constants, types, variables and procedures, where variables may be procedure-typed. Global objects with an explicit name declared in a module are allocated in the *module area* when the module is loaded. If exported, such *static objects* may be *referenced by name* in client modules. Anonymous variables with no explicit name declared in a module are allocated in a *dynamic space* called the *heap* when needed during program execution through invocation of the intrinsic procedure NEW. Such *dynamic objects* may be *reachable* by named pointer variables declared in any of the loaded modules.

In the Oberon system, there exist several possible types of references to a loaded module M:

1. *Client references* exist when other loaded modules import module M. Client modules may refer by name to exported constants, types, variables or procedures declared in module M.

2. *Type references* exist when *type tags* (addresses of type descriptors) in dynamic objects reachable by other loaded modules refer to descriptors of types declared in module M[3].

---

[1] *Wirth N. and Gutknecht J.:The Oberon System, Computer Science Report 88, ETH Zürich (1988).*
[2] *http://www.github.com/andreaspirklbauer/Oberon-extended*
[3] *In the programming language Oberon, pointer types are said to be "bound" to their base types, giving rise to additional (hidden) references from dynamic objects to descriptors of types declared in any of the loaded modules. The concept of "pointer binding", first implemented in the language Algol W and later adopted in Pascal, Modula-2 and Oberon, is essential for guaranteeing type safety, as it permits the validation of pointer values at the time of compilation without loss of run-time efficiency and in addition enables run-time t y p e c h e c k s on dynamic objects.*

3. *Procedure variable references* exist when procedure variables in static or dynamic objects reachable by other loaded modules refer to procedures declared in module M.

4. *Pointer variable references to static module data* exist when pointer variables in static or dynamic objects reachable by other loaded modules refer to *static* objects declared in module M. Such references are only possible by resorting to low-level facilities and should be avoided (pointer variables should point exclusively to anonymous variables allocated in the *heap* when needed during program execution). But since they *can* in theory exist and the required metadata (locations of all pointer variables) is already present for other reasons[4], a specific implementation may opt to also check for them prior to module unloading.

Note that pointer variable references *to* dynamic objects in the heap reachable by the modules to be unloaded are *not* considered as *module references* and therefore *should* not prevent module unloading. The Oberon garbage collector will handle such references during a future garbage collection cycle (heap objects that *were* reachable *only* by previously unloaded modules will automatically be collected).

Implementing safe module unloading

In order to make module unloading *safe*, *all* possible types of references to a loaded module or module group must be checked. In Extended Oberon, this is done as follows (see Figure 1):

- If clients exist among other loaded modules, a module or module group is never unloaded.
- If no references to the module or group of modules to be unloaded exist in the remaining modules and data structures, it is unloaded *and* its associated memory is released.
- If no clients, but type, procedure or pointer variable references to static module data of the specified modules exist, the module *unload* command takes no action by default and merely displays the names of the modules and the references that caused the removal to fail.
- If, however, the *force* option /f is specified in the module *unload* command, the modules to be unloaded are initially removed only from the *list* of loaded modules, without releasing their associated memories *(module blocks)*. Such "hidden" modules are later physically removed from *memory* using the command *Modules.Collect* (or the command *System.Collect* which also invokes *Modules.Collect*), as soon as no more references to these modules exist.

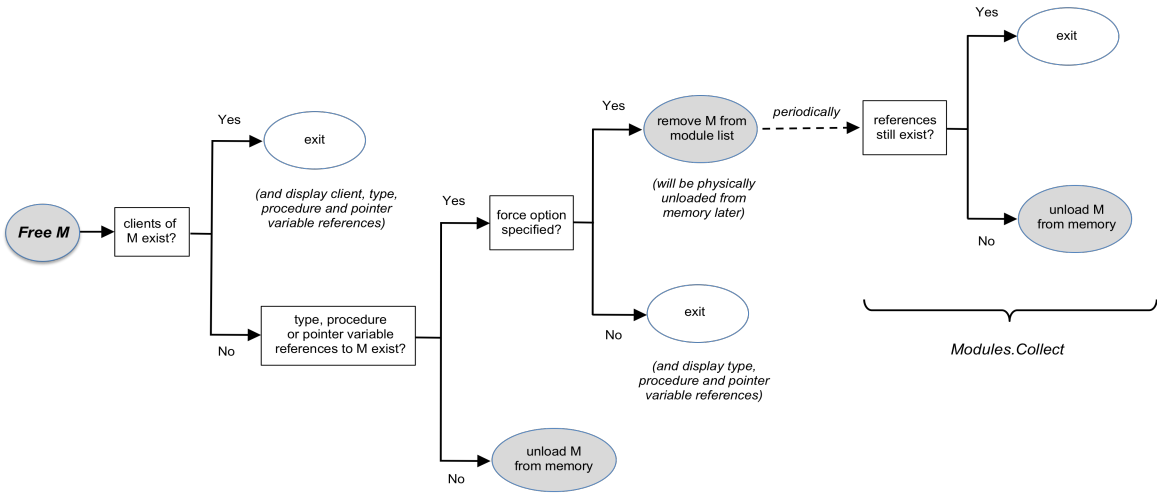

Figure 1: Safe module unloading in Extended Oberon

[4] *The Oberon garbage collector uses the locations of global pointer variables as the roots of graphs of heap objects to be traversed during the mark phase.*

Removing a module only from the module *list* amounts to *renaming* it, with the implication that a newer version of the same module with the same (original) name can be reloaded again, without having to unload (from memory) earlier versions that are still referenced by other modules[5]. By contrast, unloading a module from *memory* physically frees up the memory area previously occupied by its *module block*[6]. In Extended Oberon, this is only possible when no more module references of *any* kind to the module block exist from *anywhere* in the system.

To automate the process of unloading no longer referenced *hidden* module data from memory, the command *Modules.Collect* is included in the background task handling garbage collection[7]. It checks all possible combinations of *k* modules chosen from *n* hidden modules for references to them, and removes those module subgroups from memory that are no longer referenced.

In sum, module unloading does not affect any past or future references, as module data is kept in memory for exactly as long as necessary and removed from it as soon as possible.

For example, older versions of a module's code can still be executed if they are referenced by static or dynamic procedure variables in other loaded modules, even if a newer version of the same module has been reloaded in the meantime[8].

Type descriptors also remain accessible to other modules for exactly as long as needed. This covers the important case where a structure rooted in a variable of base type T declared in a base module M contains elements of an extension T' defined in a client module M', which is unloaded. Such elements typically contain both *type* references (type tags) and *procedure variable* references (installed handlers) referring to M'.

If a module *group* is to be unloaded and there exist references *only* within this group, the group is unloaded *as a whole*. This can be used to remove module groups with *cyclic* references[9]. It is also possible to release *entire subsystems* of modules. The command *System.FreeImports* attempts to unload the specified modules and all their direct and indirect *imports*. It may be used for conventional module stacks with a single top module, for example a compiler. By contrast, the command *System.FreeClients* unloads the specified modules and all their direct and indirect *client*s. This variant may be used for module stacks written in the object-oriented programming style with their typically inverted module hierarchy, for example a graphics editor.

Note, however, that these unloading strategies amount to *heuristics* and may tend to unload rather large parts of the system, which may not be desired. Therefore, the recommended way to unload modules is to use the *base* command *System.Free* with a *specific* set of modules provided as parameters. For added convenience, the tool commands *System.ShowReferences* and *System.ShowGroupReferences* can be used to identify all modules containing references to a given module or module group.

---

[5] Modules removed only from the module list are marked with an asterisk in the output of the command System.ShowModules. Commands of such "hidden" modules can be accessed by either specifying the module number or the modified module name. Typical use cases include hidden modules that have still open viewers or installed background tasks. If the command to close a viewer is displayed in the viewer's menu bar, the user can manually edit the command text (by clicking within its bottom 2 pixel lines) and prepend an asterisk to the module name or replace it with the module number. A more convenient alternative approach is to provide a "Close" command that also accepts the marked viewer as argument, using procedure Oberon.MarkedViewer.

[6] In Project Oberon 2013 and Extended Oberon, the module block also contains the module's type descriptors. In some other Oberon implementations, such as Ceres-Oberon, type descriptors are not placed in the module block, but are allocated in the heap at module load time in order to persist them beyond the lifetime of their associated modules. In Extended Oberon, no such extra precaution is necessary, as module blocks that are still referenced are only removed from the module list, but not from memory.

[7] Note that the garbage collector itself must also include the global pointer variables of hidden modules as roots of graphs of heap objects to be traversed, as they could still be referenced, for example when a procedure of a hidden module is called or when global pointer variables of a hidden module are accessed.

[8] If an older version of a module's code accesses global variables (of itself or of other modules), it will automatically access the "right" version of such variables – as it should.

[9] In Oberon, cyclic references can be created by pointers or procedure variables, whereas cyclic module imports are normally not allowed. However, through a tricky sequence of compilation and editing steps, it is actually possible to c o n s t r u c t cyclic module imports, which cannot be detected by the compiler. But even though they can be created, such modules are not allowed to be loaded, as the module loader of Extended Oberon – adopting the approach chosen in Original Oberon and Project Oberon 2013 – would enter an endless recursion, eventually leading to a stack overflow or out-of-memory condition – a totally acceptable solution for such an artificially constructed case. But even if modules with cyclic module imports w e r e allowed to be loaded, Extended Oberon would handle them correctly upon unloading, i.e. if no external clients or references exist, such a module group would simply be unloaded a s a w h o l e.

Implementation aspects

In our implementation, the module *unload* command *System.Free* first has to *select* the module or module group to be unloaded using an auxiliary procedure *Modules.Select* before invoking the main reference checking procedure *Modules.Check*. Client references are always checked first. This is as simple as verifying whether *unselected* modules import *selected* modules[10]:

```
 1  PROCEDURE FindClients*(client: ClientHandler; VAR res: INTEGER);
 2    VAR mod, imp, m: Module; p, q: INTEGER; continue: BOOLEAN;
 3  BEGIN res := noref; m := root; continue := client # NIL;
 4    WHILE continue & (m # NIL) DO
 5      IF (m.name[0] # 0X) & m.selected & (m.refcnt > 0) THEN mod := root;
 6        WHILE continue & (mod # NIL) DO
 7          IF (mod.name[0] # 0X) & ~mod.selected THEN p := mod.imp; q := mod.cmd;
 8            WHILE p < q DO (*imports*) imp := Mem[p];
 9              IF imp = m THEN INC(res, client(mod, imp, continue)); p := q ELSE INC(p, 4) END
10            END
11          END ;
12          mod := mod.next
13        END
14      END ;
15      m := m.next
16    END
17  END FindClients;
```

If clients exist among the *unselected* modules, no further action is taken and the module *unload* command exits. If no clients exist, *type*, *procedure* and *pointer variable* references from objects allocated in the heap[11] are checked next, using a conventional *mark-scan* scheme:

```
 1  PROCEDURE FindDynamicReferences*(typ, ptr, pvr: RefHandler;
 2    VAR resTyp, resPtr, resPvr: INTEGER; all: BOOLEAN);
 3    VAR mod: Module;
 4  BEGIN mod := root;
 5    WHILE mod # NIL DO
 6      IF (mod.name[0] # 0X) & ~mod.selected THEN Kernel.Mark(mod.ptr);
 7        IF ~all THEN Kernel.Scan(typ, ptr, pvr, mod.name, resTyp, resPtr, resPvr) END
 8      END ;
 9      mod := mod.next
10    END ;
11    IF all THEN Kernel.Scan(typ, ptr, pvr, "", resTyp, resPtr, resPvr) END
12  END FindDynamicReferences;
```

During the initial *mark* phase, objects reachable by all named global pointer variables of some or all *other* loaded modules are marked (line 6), thereby excluding objects reachable *only* by the modules to be unloaded. This automatically recognizes module *groups* and ensures that when a module group is referenced *only* by itself, it can still be safely unloaded. An additional boolean parameter *all* allows the caller to indicate whether the *mark* phase should first mark all heap objects reachable by *all* other loaded modules before initiating the *scan* phase *once* (used for module unloading), or whether *both* the mark and the scan phase should be initiated for *each* unselected module (used for identifying references from each unselected module *individually*).

The subsequent *scan* phase (line 7, 11), implemented as a separate procedure *Scan* in module *Kernel*[12], scans the heap sequentially, unmarks all *marked* objects and checks whether their *type*

[10] *Mem stands for the entire memory and assignments involving Mem are expressed as SYSTEM.GET(a, x) for x := Mem[a] and SYSTEM.PUT(a, x) for Mem[a] := x.*
[11] *In Project Oberon 2013, only r e c o r d s can be allocated using the predefined procedure NEW. In Extended Oberon, fixed-length and open a r r a y s can also be dynamically allocated.*
[12] *The original procedure Kernel.Scan (implementing the scan phase of the Oberon garbage collector) has been renamed to Kernel.Collect – in analogy to procedure Modules.Collect.*

tags point to descriptors of *types* and whether *procedure* or *pointer variables* declared in those same objects refer to *procedures* or *static objects* of the modules to be unloaded.

In order to omit in module *Kernel* any reference to the list of modules rooted in module *Modules*, procedure *Scan* is expressed as a *generic* object traversal scheme, which accepts *parametric* handler procedures that are called for each encountered object with the *source* and *destination* of the potential reference as parameters. The following is a simplified version of this scheme[13]:

```
 1  PROCEDURE Scan*(typ, ptr, pvr: Handler; s: ARRAY OF CHAR; VAR resTyp, resPtr, resPvr: INTEGER);
 2    VAR offadr, offset, p, r, mark, tag, size, pos, len, elemsize, blktyp: LONGINT; continue: BOOLEAN;
 3  BEGIN p := heapOrg; resTyp := 0; resPtr := 0; resPvr := 0; continue := (typ # NIL) OR (ptr # NIL) OR (pvr # NIL);
 4    REPEAT mark := Mem[p+4];
 5      IF mark < 0 THEN (*free*) size := Mem[p]
 6      ELSE (*allocated*) tag := Mem[p]; size := Mem[tag];
 7        IF mark > 0 THEN (*marked*) Mem[p+4] := 0;
 8          IF continue THEN
 9            IF typ # NIL THEN INC(resTyp, typ(p, tag, s, continue)) END ;
10            IF continue & ((ptr # NIL) OR (pvr # NIL)) THEN offadr := tag + 16; offset := Mem[offadr]
11              WHILE continue & (offset # -1) DO (*pointers*)
12                IF ptr # NIL THEN r := Mem[p+8+offset]; INC(resPtr, ptr(p+8+offset, r, s, continue)) END ;
13                INC(offadr, 4); offset := Mem[offadr]
14              END ;
15              IF continue & (pvr # NIL) THEN INC(offadr, 4); offset := Mem[offadr]
16                WHILE continue & (offset # -1) DO (*procedures*) r := Mem[p+8+offset];
17                  INC(resPvr, pvr(p+8+offset, r, s, continue));
18                  INC(offadr, 4); offset := Mem[offadr]
19                END
20              END
21            END
22          END
23        END
24      END ;
25      INC(p, size)
26    UNTIL p >= heapLim
27  END Scan;
```

where the handler procedures *typ*, *ptr* and *pvr* are invoked as follows for each marked object:

- Procedure *typ* is called with the address of (the *type tag* of) the heap object as the *source* of the potential reference that is to be checked, followed by the address of the referenced type descriptor as its *destination* (line 9).
- Procedure *ptr* is called for each *pointer variable* in the heap object with the address of the pointer variable passed as the *source*, followed by the address of the referenced object passed as the *destination* (line 12).
- Procedure *pvr* is called for each *procedure variable* in the heap object with the address of the procedure variable passed as the *source*, followed by the address of the referenced procedure passed as the *destination* (line 17).

The results of these calls are *separately* added up for each handler and returned via the variable parameters *resTyp, resPtr* and *resPvr*. By convention, a result of zero indicates that no reference has been found. Apart from that, handler procedures are free to define the semantics of their return values. An additional boolean variable parameter *continue* allows a handler procedure to

[13] *The simplified version scans only r e c o r d blocks allocated in the heap via NEW(p), where p is a POINTER TO RECORD. The full implementation also covers a r r a y blocks.*

indicate to the caller that it is no longer to be called (lines 8, 10, 11, 15, 16). The scan process itself continues, but only to *unmark* the remaining marked heap objects (line 7).

Finally, the check for procedure and pointer variable references to static module data of the modules to be unloaded is also performed for all *global* pointer and procedure variables of all other loaded modules, whose offsets in their corresponding module's data section are obtained from two arrays in their module's *meta* data section, headed by the links *mod.ptr* and *mod.pvr:*

```
 1  PROCEDURE FindStaticReferences*(ptr, pvr: RefHandler; VAR resPtr, resPvr: INTEGER);
 2    VAR mod: Module; pref, pvadr, r: LONGINT; continue: BOOLEAN;
 3  BEGIN resPtr := noref; resPvr := noref; mod := root; continue := (ptr # NIL) OR (pvr # NIL);
 4    WHILE continue & (mod # NIL) DO
 5      IF (mod.name[0] # 0X) & ~mod.selected THEN
 6        IF ptr # NIL THEN pref := mod.ptr; pvadr := Mem[pref];
 7          WHILE continue & (pvadr # 0) DO (*pointers*) r := Mem[pvadr];
 8            INC(resPtr, ptr(pvadr, r, mod.name, continue));
 9            INC(pref, 4); pvadr := Mem[pref]
10          END
11        END ;
12        IF pvr # NIL THEN pref := mod.pvr; pvadr := Mem[pref];
13          WHILE continue & (pvadr # 0) DO (*procedures*) r := Mem[pvadr];
14            INC(resPvr, pvr(pvadr, r, mod.name, continue));
15            INC(pref, 4); pvadr := Mem[pref]
16          END
17        END
18      END ;
19      mod := mod.next
20    END
21  END FindStaticReferences;
```

We note that the procedures *FindClients, FindDynamicReferences* and *FindStaticReferences* are also expressed as *generic* object traversal schemes, which – similar to procedure *Scan* – accept parametric handler procedures that are invoked for each encountered object. This allows these procedures to be used for other purposes as well, for example to *enumerate* all references to a given module or module group. The main reference checking procedure *Modules.Check* uses these procedures by passing its own (private) handler procedures *HandleClient* and *HandleRef*, which merely set the boolean variable parameter *continue*, depending on whether a reference to any of the selected modules has been found or not. We remark that *Modules.Check* is not only called when a module is unloaded by the *user*, but also by the Oberon *background task* that removes no longer referenced *hidden* module data from memory. Thus, it must be able to handle both visible *and* hidden modules in the data structure rooted in module *Modules*.

Implementation prerequisites

In order to make the outlined validation pass possible, type descriptors of *dynamic* objects in the heap[14] and descriptors of global module data in the *static* module blocks now also contain the *locations of all procedure variables* (in addition to the locations of all *pointer* variables, which are already present for other reasons) within the described object, adopting an approach employed in one of the earlier implementations of the Oberon system (MacOberon)[15].

The descriptors also contain the offsets of *hidden* (not exported) procedure variables, enabling the module *unload* command to check *all* static and dynamic procedure variable locations in the

[14] *See chapter 8.2 of the book Project Oberon 2013 Edition for a description of an Oberon type descriptor. In essence, it contains information that is s h a r e d by all dynamically allocated records of the same t y p e, such as its size, information about type extensions and the offsets of pointer and procedure variable fields.*
[15] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

entire system for possible procedure variable references to the selected modules to be unloaded. The resulting run-time representations of a heap record with its associated type descriptor and a static module block are shown in Figure 2 (the *method table* used to implement Oberon-2 style *type-bound procedures* is not discussed here).

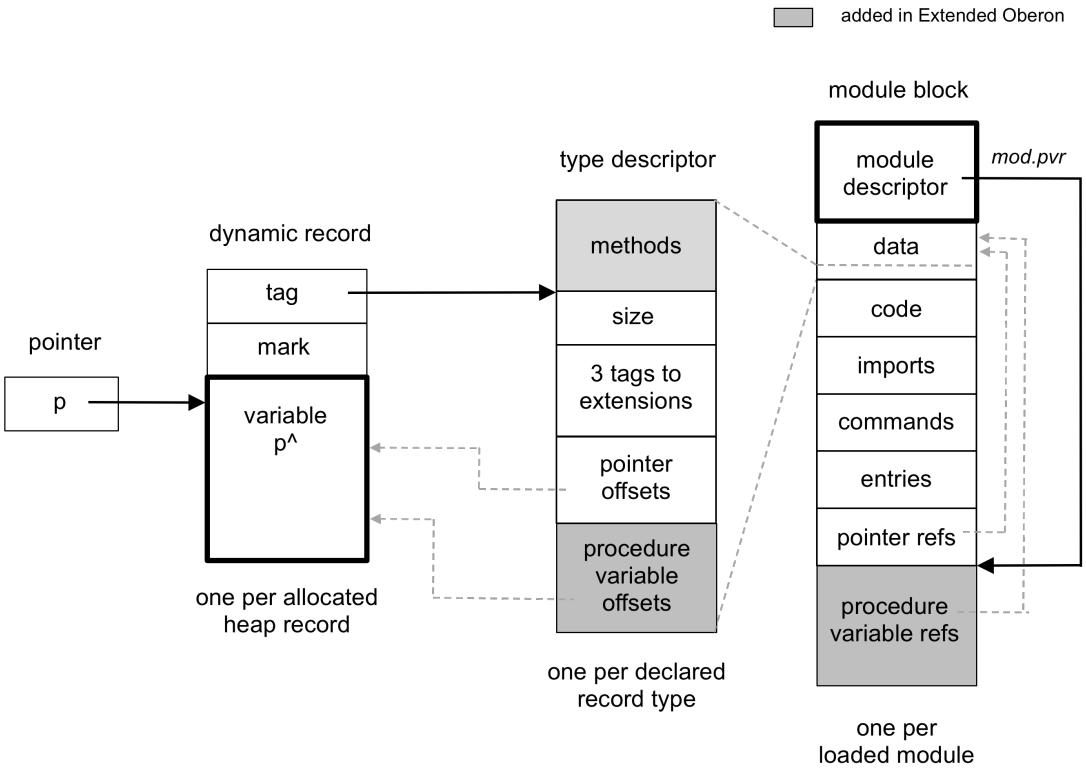

Figure 2: Run-time representation of a dynamic record with its type descriptor and a module block in Extended Oberon

To make the offsets of hidden *procedure variables* in exported record types available to clients, symbol files also include them. An importing module may, for example, declare a variable of an imported record type, which contains *hidden* procedure variable fields, or declare a record type, which contains (or extends) an imported type. We recall that in Project Oberon 2013, hidden *pointers*, although not exported and therefore invisible in client modules, are already included in symbol files because their offsets are needed for garbage collection. Similarly, in Extended Oberon, hidden *procedure variables* are included in symbol files because their offsets are needed for reference checking prior to module unloading, as illustrated in the following example:

```
1  MODULE M0;
2    TYPE P* = PROCEDURE; R* = RECORD p: P END ;          (*hidden record field p, visible only in M0*)
3    PROCEDURE Init*(VAR r: R; p: P); BEGIN r.p := p END Init;   (*install procedure p in hidden field r.p*)
4  END M0.
5
6  MODULE M1;
7    IMPORT M0;
8    VAR r: M0.R;                                         (*global variable r with hidden field r.p*)
9    PROCEDURE Init*(p: M0.P); BEGIN M0.Init(r, p) END Init;
10 END M1.
11
12 MODULE M2;
13   IMPORT M1;
14   PROCEDURE P; BEGIN END P;
15   PROCEDURE Clear*; BEGIN M1.Init(NIL) END Clear;      (*clear reference from M1.r.p*)
```

```
16    PROCEDURE Init*; BEGIN M1.Init(P) END Init;          (*create reference from M1.r.p to M2.P*)
17  END M2.
18
19  M2.Clear    System.Free M2 ~     unloading sucessful (as no references to M2 exist)
20  M2.Init     System.Free M2 ~     unloading failed (procedures of M2 in use in global procedure variables of M1)
```

Here the global record variable *r* declared in module *M1* (line 8) is of an imported record type *M0.R*, which contains a *hidden* procedure variable field *r.p*, visible only in M0. If another module *M2* installs a procedure *M2.P* in this field (line 16), a hidden global procedure variable reference from *M1.r.p* to *M2.P* is created, with the intended effect that *M2* can no longer be unloaded (line 20). In order to be able to check for such hidden references, the location of *M1.r.p* within the module block of *M1* must be known at run time. This is accomplished as follows:

- The offset of the *hidden* record field *p* of the record type *M0.R* is also included in the *symbol* file of *M0*, when M0 is compiled.
- The location of the procedure variable *M1.r.p* within the data section of *M1* (computed as the sum of the start address of the global record *M1.r* and the offset of the hidden field *M0.R.p* imported from M0) is included in the *object* file of *M1*, when M1 is compiled.
- The location of *M1.r.p* is transferred from the object file of *M1* to the corresponding array in the *meta* data section in the module block, headed by the link *mod.pvr*, when *M1* is loaded.

Hidden *pointers* are included in symbol files without their names, and their (imported) type in the symbol table is of the form *ORB.NilTyp* (as with Project Oberon 2013). Hidden *procedure variables* are also included in symbol files without their names, but their (imported) type in the symbol table is of the form *ORB.NoTyp*[16]. This choice is reflected in the conditions used to find *all* pointers and procedure variables in various procedures of the compiler[17], as illustrated in the following code excerpt of its symbol table handler (module ORB):

```
1   TYPE Ptrs* = {Pointer, NilTyp};          (*NilTyp = hidden pointer variable*)
2     Procs* = {Proc, NoTyp};                (*NoTyp = hidden procedure variable*)
3
4   PROCEDURE FindHiddenFields(VAR R: Files.Rider; typ: Type; off: LONGINT);
5     VAR fld: Object; i, s: LONGINT;
6   BEGIN
7     IF typ.form IN Ptrs THEN Write(R, Fld); Write(R, 0); Files.WriteNum(R, off)
8     ELSIF typ.form IN Procs THEN Write(R, Fld); Write(R, 0); Files.WriteNum(R, -off-1)
9     ELSIF typ.form = Record THEN fld := typ.dsc;
10      WHILE fld # NIL DO FindHiddenFields(R, fld.type, fld.val + off); fld := fld.next END
11    ELSIF typ.form = Array THEN s := typ.base.size;
12      FOR i := 0 TO typ.len-1 DO FindHiddenFields(R, typ.base, i*s + off) END
13    END
14  END FindHiddenFields;
```

The reader my wonder whether on systems, where Oberon-2 style *type bound procedures* are implemented, it is necessary to also check for hidden *methods*, in addition to hidden *pointer* and *procedure* variables. The answer is *no*, because in order to *redefine* a type-bound procedure, the redefining module must always *import*, either directly or indirectly, the module containing the method being overwritten. Since *client* references are always checked prior to module unloading, no additional check for “method references” is necessary.

Thus, unlike in the previous example, where a procedure *M2.P* passed as a parameter *p* to a procedure *Init* of the base module *M0* is *assigned* to a procedure variable *r.p* of a base record *R*

[16] *This is acceptable because for record fields, the types ORB.NilTyp and ORB.NoTyp are not used otherwise.*
[17] *See procedures ORB.FindHiddenFields, ORG.FindRefFlds, ORG.NofRefs and ORG.FindRefs.*

through a *regular* assignment (without the need to import the base module *M0*), it is not possible to *redefine* a method *M* bound to a base type *R* without also importing, directly or indirectly, the module containing the record declaration of *R*. This is elucidated in the following example:

```
1  MODULE M0;
2    TYPE R* = RECORD END ;
3    PROCEDURE (VAR r: R) M*(); BEGIN END M;        (*method M bound to R defined in M0, redefined in M2*)
4  END M0.
5
6  MODULE M1;
7    IMPORT M0;
8    TYPE R1* = RECORD (M0.R) END ;
9  END M1.
10
11 MODULE M2;
12   IMPORT M1;
13   TYPE R2* = RECORD (M1.R1) END ;                (*reimport type M0.R0 via type M1.R1*)
14   PROCEDURE (VAR r: R2) M*(); BEGIN END M;       (*redefines method M defined in M0*)
15   PROCEDURE Go*; BEGIN END Go;
16 END M2.
17
18 M2.Go ~    System.Free M0 ~                      unloading failed (M0 imported by M1 and M2)
```

Here, the type-bound procedure *M* bound to a record type *R* declared in module *M0* is *redefined* in module *M2*. Even though *M2* does not directly import *M0*, it indirectly imports *M0*, making it impossible to unload *M0* (in addition, *M0* is also imported by *M1* of course).

Module finalization

Extended Oberon allows a module to set a *module finalization* sequence, which is executed *prior to* module unloading. It can do so either by using the language construct FINAL, as shown in the following example:

```
1  MODULE M;
2    IMPORT Texts, Oberon;
3
4    VAR W: Texts.Writer;
5
6    PROCEDURE Finalize;  (*module finalization sequence*)
7    BEGIN Texts.WriteString(W, "Executing finalization sequence of module M…");
8      Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
9    END Finalize;
10
11   PROCEDURE Start*;  (*load module*)
12   END Start;
13
14 BEGIN Texts.OpenWriter(W)
15 FINAL Finalize
16 END M.
17
18
19 ORP.Compile M.Mod/s ~     # compile module M
20 M.Start ~                 # load module M
21 System.Free M ~           # unload module M (prints "Executing finalization sequence of module M…")
```

or by calling *Modules.SetFinalizer* in its *module initialization* body, as shown below. In the latter case, the specified sequence must be a *parameterless* procedure declared in the *same* module. A standalone program cannot set a module finalization sequence.

```
MODULE M;
  IMPORT Modules, Texts, Oberon;

  VAR W: Texts.Writer;

  PROCEDURE Finalize;  (*module finalization sequence*)
  BEGIN Texts.WriteString(W, "Executing finalization sequence of module M…");
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Finalize;

  PROCEDURE Start*;  (*load module*)
  END Start;

BEGIN Texts.OpenWriter(W); Modules.SetFinalizer(Finalize)
END M.


ORP.Compile M.Mod/s ~     # compile module M
M.Start ~                 # load module M
System.Free M ~           # unload module M (prints "Executing finalization sequence of module M…")
```

Assessment of the outlined solution

An obvious shortcoming of the reference checking scheme outlined above is that it requires *additional* run-time information to be present in *all* type descriptors of *all* modules *solely* for the purpose of reference checking, which in turn is *only* needed in the relatively infrequent case of module releases. In addition, module blocks now also contain an *additional* meta data section containing the offsets of global procedure variables. However, since there exists only one type descriptor per declared record *type* rather than one per allocated heap *record*, and global procedure variables tend to be rare, the additional memory requirements are negligible[18].

Although the operation of *reference checking* may appear expensive at first, in practice there is no performance issue. It is comparable in complexity to garbage collection and therefore barely noticeable – at least on systems with small to medium sized dynamic spaces (heaps). This is in spite of the fact that for *each* heap object encountered during the *mark* phase *all* modules to be unloaded are checked for references during the subsequent *scan* phase. However, reference checking *stops* when the *first* reference is detected and module unloading is rare except, perhaps, during development.

Thus, the presented solution, which was mainly chosen for its simplicity, appears to be amply sufficient for most practical purposes. We have therefore resisted the temptation to introduce additional “optimizations”, whose benefits are often questionable.

* * *

[18] Adding procedure variable offsets to type descriptors is, strictly speaking, not necessary, as the compiler can always rearrange the memory layout of a record such that all procedure variable fields are placed in a contiguous section at the beginning of it, making their offsets implicit. Thus, it is in principle possible to realize the reference checking scheme without any additional fields in type descriptors. We have refrained from implementing this refinement for two main reasons. First, it would increase the complexity of the compiler. At first glance, it might seem that in order to rearrange a record’s field list, only a small change needs to be made to a single procedure in the compiler (ORP.RecordType in Project Oberon 2013). However, a complication arises because records may recursively contain other records, each again containing procedure variables at arbitrary offsets. While it is always possible to “flatten” such recursive record structures, it would make other record operations more complex. For example, assignments to subrecords would become less natural, as their fields would no longer be placed in a contiguous section in memory. Second, the memory savings in type descriptors are marginal, given that there exists only one type descriptor per record t y p e rather than one per allocated heap r e c o r d. Most applications are (or should be) programmed in the conventional programming style, where installed procedures are rare. For example, in the Oberon system, the object-oriented programming style is restricted to the viewer system, which provides distributed control in the form of installed handlers – of which there is usually only one per t y p e of viewer. In sum, the benefit obtained by saving a few fields in a relatively small number of type descriptors is negligible, and therefore the additional effort required to implement this refinement would be hard to justify indeed.

Appendix: Historical notes on module unloading in the Oberon operating system

Most implementations of the Oberon system only check for *client* references prior to unloading a module. Other types of references are usually not checked, although various approaches are typically employed to alleviate the situation where such references do exist.

These approaches can be broadly grouped into two main categories: (a) schemes that explicitly allow invalidating references, and (b) schemes where all past and future references must remain unaffected at all times.

a. Schemes that explicitly allow invalidating references

In such schemes, unloading a module from memory *may*, and in general *will*, lead to "dangling" *type*, *procedure* or *pointer variable* references[19] pointing to no longer valid static module data. This includes the important case where a structure rooted in a variable of base type T declared in a base module M contains elements of an extension T' defined in a client module M'. Such elements typically contain both *type* references (type tags) and *procedure variable* references (installed handlers) referring to module M'. This is common in the Oberon viewer system, where *M* is module *Viewers* and *M'* may be a graphics editor, for example. If the client module M' is unloaded and the module block previously occupied by it is overwritten by another module loaded later, any still existing references to M' become *invalid* at that moment.

Global procedure variables declared in other modules may also refer to procedures declared in module M', although this case is much less common (global procedure variables tend to be used mainly for procedures declared in the same module).

A variety of approaches have been employed in different implementations of the Oberon system to cope with the introduced "dangling" *type, procedure* or *pointer variable* references:

1. On systems that use a *memory management unit* to perform virtual memory management, such as Ceres-1 or Ceres-2 computers, one approach is to *unmap* the module space of an unloaded module from virtual memory, thereby *invalidating* any future references to it. When this is done, any still existing *type*, *procedure* or *pointer variable* references to static module data point to a now *unallocated* page, and consequently any attempt to access such a page later will (intentionally) result in a system *trap*.

   We consider this an unfortunate proposal for several reasons. First, users generally have no way of knowing *whether* it is in fact safe to unload a module, yet they are allowed to do so. Second, even after having unloaded it, they still don't know whether references from other loaded modules still exist or not – until a *trap* occurs. But then it is often too late. While the trap itself will *prevent* a system crash (as intended), the user may *still* need to restart the system in order to recover an environment without any "frozen" parts, for example displayed viewers that may have been opened by the unloaded module. Note also that this solution requires special hardware support, which may not be available on all systems.

   If the *language* Oberon or the underlying Oberon system supports the concept of (module or object) *finalization*, a module can provide code that is executed prior to module *unloading*, similar to the module *initialization* code that is executed after it has been *loaded*. Such a mechanism can be used to close any still open viewers or delete any data elements that may

[19] *If the programming language Oberon-2 is used, there can also be references to method tables (which however are typically allocated within type descriptors).*

have been created by a module to be unloaded. Finalization can be realized as a *language* feature or at the *system* level by providing a procedure that allows any module to set a finalization routine. Extended Oberon is an example of an system that supports both.

2. On systems that do *not* use virtual memory, such as Original Oberon on Ceres-3 or Project Oberon 2013 on RISC, the easiest way to cope with dangling references is to simply *ignore* them and to *always* release the memory associated with a module without any further precautions (unless, of course, clients exist). While an attempt to access an unloaded module goes undetected *initially*, the system is still left in an *unsafe* state, which may become *unstable* the moment another module loaded later overwrites the previously released module block and other modules or data elements still contain "dangling" references to its (now overwritten) *type descriptors*, *procedures* or other *static objects*. This is of course undesirable.

3. Another approach, which however can be used only for *procedure variable* references, is to identify *all* such references and make them refer to a *dummy* procedure, thereby preventing a run-time error when such "fixed up" procedures are called later. This solution requires the system to *know* the locations of all procedure variables in all static and dynamically allocated objects at run time (which can be achieved by including them in symbol and object files). It was used in an earlier version of Extended Oberon, but was later discarded, mainly because the resulting effect on the *overall* behavior of a running system is essentially impossible to predict. The fact that *some* procedure variables *somewhere* in the system no longer refer to *real* but to *dummy* procedures typically becomes "visible" only through the *absence* of some action – such as the absence of mouse tracking if the unloaded module contained a viewer handler, for instance.

4. On systems that use *indirection* for procedure calls via a so-called "link table" (where an "address" of a procedure is not a real memory address, but an *index* to this translation table, which the caller consults for every procedure *call* in order to obtain the actual memory location of the called procedure), the same effect can be achieved by setting the *link table entries* for all referenced *procedures* of the module to be unloaded to *dummy* entries, rather than locating and modifying each individual procedure *call* in the system.

   However, using a *link table* to implement indirection for procedure calls is only viable on systems that provide *efficient* hardware support for it. It was used in earlier versions of the Oberon system on Ceres-1 and Ceres-2 computers, which were based on the (now defunct) NS32000 processor. This processor featured a *call external procedure* instruction (CXP k, where k is the index of the link table entry of the called procedure), which sped up the process of calling external procedures significantly[20]. Later versions of the NS processor, however, internally re-implemented the *same* CPU instruction using microcode, following the general industry trend that started in the 1980s[21] (which has now become mainstream) of implementing only frequent, simple instructions directly with hardware, while interpreting more complex instructions using internal microcode. This internal re-implementation of the *CXP k* instruction negatively impacted its performance. For this and a variety of other reasons, this instruction – and with it the *link table* – were no longer used in later versions of Oberon.

5. In passing we note that for *type* references, it is actually possible to determine at *compile* time, whether a module *may* lead to references from other loaded modules at *run* time. The

[20] *The use of the link table also increased code density considerably (as only 8 bits for the index instead of 32 bits for the full address were needed to address a procedure in every procedure call). In addition, the link table used by the CXP instruction allowed for an expedient linking process at module load time (as there are far fewer conversions to be performed by the loader – one for every referenced procedure instead of one for every procedure call) and also eliminated the need for a fixup list (list of the locations of all external procedure references to be fixed up by the module loader) in the object file. A disadvantage is, of course, the need for a (short) link table.*

[21] *A number of computers from the 1960s and 1970s have been identified as forerunners of reduced instruction set computers (RISCs), but the modern concept dates to the 1980s.*

criteria is the following: if a module M' does *not* declare record types which are extensions T' of an imported type T, then records declared in module M' *cannot* be inserted in a data structure rooted in a variable *v* of the imported type T – precisely *because* their types are not extensions of T (in the Oberon programming language, an assignment *p := p'* is allowed only if the type of *p'* is the same as the type of *p* or an extension of it). One *could* therefore introduce a rule that a module M' can be safely dispensed *only* if it does *not* declare record types, which are extensions T' of an imported type T. The flip side of such a rule, however, is that modules that actually *do* declare such types can *never* be unloaded – unless, of course, other ways to safely unload such modules are implemented. In addition, *procedure variable* references are not covered by this rule (procedures of M' may be referenced by procedure variables declared in other modules or in dynamic objects)[22].

Even though most of these approaches have actually been realized in various implementations of the Oberon system, we consider none of them truly satisfactory. In our view, these schemes appear to only tinker with the symptoms of a problem that would not exist, if only one adopted the rule to *disallow* the physical removal (from memory) of still referenced module data.

The main issue is that the moment one *allows* modules to release their associated memory in spite of references to them from other modules, the resulting *dangling* module references must be dealt with *somehow* in order to prevent an almost certain system *crash*. However, *fixing up* or even *invalidating* references will *always* remove essential information from the system. As a result, the run-time behavior of the modified system becomes *essentially unpredictable*, as other modules may *critically* depend on the removed functionality. For example, unloading a module that contains an installed handler of a *contents frame* may render it impossible to close the enclosing *menu viewer* that contains it, thereby leading to a system with "frozen" parts.

b. Schemes where all past and future references must remain unaffected at all times

The second possible interpretation of *unloading* a module consists of schemes where all past and future references *must* remain unaffected at all times. In such schemes, module unloading can be viewed as an implicit mandate to preserve "critical module data", as long as references to the unloaded module exist. Various possible ways to satisfy this requirement exist:

1. One could simply exit the module *unload* command with an error message, whenever such references are detected. The user, however, may then be "stuck" with modules that he can *never* unload because they are referenced by modules over which he has no explicit control.

2. But the mandate could also be fulfilled by persisting any still referenced module data to a "safe" location before unloading the associated module.

   For *type descriptors* referenced by *type tags* in dynamic records a straightforward solution exists: simply allocate them *outside* their module blocks in order to persist them beyond the lifetime of their associated modules. One possibility is to allocate them in the *heap* itself at module *load* time[23]. This approach has been implemented in Oberon on Ceres-1 and Ceres-2. It eliminates dangling *type* references altogether and therefore also the *need* to check for them at run time, because type descriptors are now unaffected by module unloading. Such dynamically allocated type descriptors can themselves be deallocated if (a) their associated module has been unloaded and (b) no *regular* heap objects refer to them via their *type tags*.

[22] *One may attempt to "record" each assignment to a procedure variable at run time by setting a flag in the referenced procedure or module, but the performance penalty would be inacceptable.*
[23] *Note that one cannot simply move type descriptors around in memory once a module has been loaded, because their addresses are (typically) used to implement type tests and type guards. By allocating them in the heap at module l o a d time, one avoids the need to move them to a different location when a module is later unloaded.*

For *procedures* referenced by *procedure variables* no such straightforward solution exists, as procedures may call other procedures or access global module data declared in the same or a different module. Thus, the only way to "persist" procedures (and other static objects) would be to persist the *entire* module.

We conclude that if one wants to handle all types of references, one *cannot* unload a module from memory, as long as references to it still exist from anywhere in the system[24].

3. One way to automatically persist type descriptors, procedures and static objects consists of simply *never* releasing the module block of a module once it has been loaded. Instead, when the user requests the "unloading" of a module, the module is only removed from the *list* of loaded modules, without releasing its associated memory. This amounts to *renaming* the module, with the implication that a newer version of the same module with the same original name can be reloaded again. This approach has been chosen in MacOberon[25], for example. Since the associated memory of a module is never released, the issue of dangling type, procedure or pointer variable references is avoided altogether, as they simply cannot exist. But it can also lead to higher-than-necessary memory consumption, if one and the same module is repeatedly loaded and unloaded (typical during *development*). Nevertheless, such an approach may be viewed as adequate on *production* systems, where module unloading tends to be rare, or on systems that use *demand paging*, where memory pages that are not frequently referenced are automatically swapped out to a secondary store. Note, however, that with the advent of large primary stores, the concept of demand paging has lost much of its significance. It is therefore best not to rely on the presence of such a mechanism.

4. A further *refinement* of the approach outlined above consists of *initially* removing a module only from the module list (as is done in MacOberon), but *in addition* releasing its associated memory *as soon as* there are no more references to its static module data. If this is done in an automatic fashion, module data is truly "kept in memory for exactly as long as necessary and removed from it as soon as possible". This is the approach chosen in Extended Oberon. A variation of it was used in some versions of SparcOberon[26].

In sum, module unloading schemes where references remain *unaffected* at all times avoid many of the complications that are inherent in schemes that explicitly *allow* invalidating references. A (small) price to pay is to keep modules loaded in memory, as long as references to them exist. However, on *production* systems, there is usually no need to keep multiple versions of the same module loaded in memory, while on *development* systems it is totally acceptable.

Finally, we note that on modern day computers the amount of available memory, and therefore also the amount of *dynamic* data that can be allocated by modules, typically far exceeds the size of the static module blocks holding their program code and global variables. Therefore, *not* releasing module blocks immediately after a module *unload* operation typically has a rather negligible impact on overall memory consumption.

* * *

[24] *Mixed variants are also possible. For example, one could allocate type descriptors in the heap at module load time (as with Ceres-Oberon), and either "fix up" procedure variable references or prevent the release of a module block if such references still exist; however, most modules referenced by type tags are also referenced by procedure variables – this is in fact the typical case for dynamic records containing installed handler procedures. Thus, it seems more natural to use the same approach for both type and procedure variable references.*

[25] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

[26] *http://e-collection.library.ethz.ch/eserv/eth:7103/eth-7103-01.pdf (SPARC-Oberon User's Guide and Implementation, 1990/1991)*