# Safe module unloading

# in the Oberon operating system

Andreas Pirklbauer

1.9.2019

## Purpose

This technical note describes an experimental implementation of *safe* module unloading for the Oberon operating system, as realized in Experimental Oberon[1], a revision of FPGA Oberon[2].

## Module unloading in the Oberon system

In the Oberon system, there exist three possible types of references to a loaded module M[3]:

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

In most Oberon implementations, only *client* references are checked prior to module unloading. *Type* and *procedure variable* references are usually not checked, although various approaches are typically employed to address the case where such references exist[4]. As a result, module unloading has traditionally left an Oberon system in an *unsafe* state, which will become *unstable* the moment another module loaded later *overwrites* a previously released module block and other modules still contain *dangling* references to its *type descriptors* or *procedures*. In addition, unloading of module *groups* with references only among themselves is usually not supported.

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

[2] http://www.projectoberon.com

[3] An Oberon module can be viewed as a container of types, variables and procedures, where variables may be procedure-typed. Once loaded, modules may be interrelated with each other in a variety of ways. First, 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 ("client references"). Second, anonymous variables with no explicit name declared in a module are allocated in the dynamic space (heap) when needed during program execution using the predefined procedure NEW. Such "dynamic" objects may be "reachable" by multiple named pointer variables, possibly declared in different modules. Third, in the language Oberon pointer types are "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 ("type references"). The concept of "pointer binding", first implemented in the language Algol W and adopted in Pascal, Modula-2 and Oberon, is absolutely crucial for guaranteeing type safety, as it permits the validation of pointer values at the time of compilation without loss of run-time efficiency and also enables run-time type checks on dynamic objects. Finally, assignments to procedure-typed variables lead to references in the form of installed procedures ("procedure variable references"). However, only client references, type references from dynamic objects and procedure variable references from either static or dynamic objects to the modules to be unloaded need to be checked prior to module unloading. By contrast, pointer references to dynamic objects should not prevent module unloading, as such objects are not viewed as "belonging" to a single module. Pointer references will be handled by the Oberon garbage collector during a future garbage collection cycle (dynamic objects that previously were reachable only by the just unloaded modules will be automatically collected). Similarly, pointer references to static objects (e.g., global variables) in the modules to be unloaded are only possible by resorting to low-level facilities and should be avoided – and, in fact, be disallowed (pointer variables should point exclusively to anonymous variables allocated dynamically in the heap when needed during program execution).

[4] See the appendix for historical notes on module unloading in the Oberon system.

## 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 Experimental Oberon, this is done as follows (Figure 1):

- If clients exist among other loaded modules, a module or module group is never unloaded.
- If no client, type or procedure variable references to a module or module group exist in the remaining modules or data structures, it is unloaded and its associated memory is released.
- If no clients, but type or procedure variable references exist, the module *unload* command takes no action by default and merely displays the names of the modules and the types of references that caused the removal to fail. If, however, the *force* option /f is specified[5], the modules are initially removed only from the *list* of loaded modules, without releasing their associated memory. Such "hidden" modules will later be physically removed from *memory* using the command *Modules.Collect*[6], as soon as there are no more references to them.

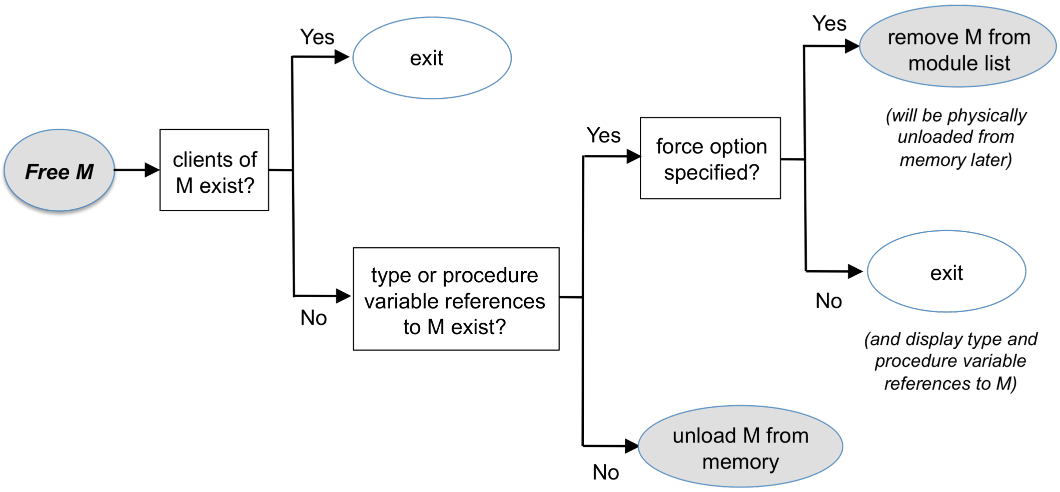


**Figure 1:** Safe module unloading in Experimental Oberon

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 name) can be reloaded again, without having to unload (from memory) earlier versions that are still referenced[7]. By contrast, unloading a module from *memory* frees up the memory area previously occupied by the module block[8]. In Experimental Oberon, this is only possible when no references to the module exist.

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. 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.

---

[5] *The force option /f must be specified at the* e n d *of the list of modules to be unloaded, e.g., System.Free M1 M2 M3/f*

[6] *The tool command System.Collect also invokes Modules.Collect*

[7] *Modules removed only from the list of loaded modules are marked with an asterisk in the output of the command System.ShowModules. Commands of such "hidden" modules can be accessed by either specifying their module number or their (modified) module name, both of which are displayed by the command System.ShowModules. In both cases, the corresponding command text must be enclosed in double quotes. If a module M carries module number 14, for instance, one can activate a command M.P also by clicking on the text "14.P". Typical use cases include hidden modules that still have background tasks installed which can only be removed by activating a command of the hidden modules themselves, or hidden modules that still have open viewers. 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 two pixel lines), replace the module name by its module number and enclose the modified command text in double quotes. Although this is somewhat clumsy, it at least enables the user to close the viewer. An alternative approach is to provide a "Close" command that also accepts the marked viewer as argument (using procedure Oberon.MarkedViewer).*

[8] *In FPGA Oberon and Experimental Oberon, the module block includes the module's type descriptors. In some other Oberon implementations, such as Ceres-Oberon, type descriptors are not stored 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 Experimental Oberon, no such extra precautionary measure is necessary, as module blocks that are still referenced are only removed from the module list, but not from memory.*

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 modules, even if a newer version of the same module has been reloaded in the meantime[9]. 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, it is unloaded *as a whole*. This can be used to remove module groups with *cyclic* references[10]. 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 that these unloading strategies amount to *heuristics* and may tend to unload rather large parts of the system, which may not be desired. 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.ShowRefs* and *System.ShowGroupRefs* can be used to identify all modules containing references to a given module or module group.

### Implementation aspects

To check for references, the module *unload* command *System.Free* first *selects* the modules to be unloaded using procedure *Modules.Select* and then invokes procedure *Modules.Check*. Client references are checked first. This is accomplished by simply verifying whether *unselected* modules import *selected* modules[11]:

```
PROCEDURE FindClients*(proc: ImpHandler; VAR res: INTEGER);
  VAR mod, imp, m: Module; p, q: INTEGER; continue: BOOLEAN;
BEGIN res := noref; m := root; continue := proc # NIL;
  WHILE continue & (m # NIL) DO
    IF (m.name[0] # 0X) & m.selected & (m.refcnt > 0) THEN mod := root;
      WHILE continue & (mod # NIL) DO
        IF (mod.name[0] # 0X) & ~mod.selected THEN p := mod.imp; q := mod.cmd;
          WHILE p < q DO imp := Mem[p];
            IF imp = m THEN INC(res, proc(mod, imp, continue)); p := q ELSE INC(p, 4) END
          END
        END ;
        mod := mod.next
      END
    END ;
    m := m.next
  END
END FindClients;
```

---

*[9] 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.*

*[10] 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 construct 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 Experimental Oberon – adopting the approach chosen in Original Oberon and FPGA Oberon – 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 were allowed to be loaded, Experimental Oberon would handle them correctly upon unloading, i.e. if no external clients or references exist, such a module group would simply be unloaded as a whole.*

*[11] 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.*

If clients exist among the *unselected* modules, no further action is taken and the module *unload* command exits. If no clients exist, *type* and *procedure variable* references from dynamic objects in the heap are checked next[12]. This is accomplished using a conventional *mark-scan* scheme:

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

During the *mark* phase, heap objects reachable by all named global pointer variables of *other* currently loaded modules are marked (line 5), 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 unloaded. The subsequent *scan* phase (lines 6, 10), implemented as a separate procedure *Scan* in module *Kernel*[13], scans the heap sequentially, unmarks all *marked* objects and checks whether the *type tags* of these objects point to descriptors of *types* and whether *procedure variables* declared in those same objects refer to *procedures* declared in the modules to be 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 modules before initiating the *scan* phase *once* (used for module unloading), or whether the *scan* phase should be initiated for *each* unselected module *individually* (can be used for identifying references from each unselected module).

Finally, the check for *procedure variable* references is also performed for all *global* procedure variables, whose offsets in the module's data section are obtained from an array in the module's *meta* data section, headed by the link *mod.pvr* in the module descriptor (see below):

```
PROCEDURE FindStaticRefs*(proc: RefHandler; VAR res: INTEGER);
  VAR mod: Module; pref, pvadr, r: LONGINT; continue: BOOLEAN;
BEGIN res := noref; mod := root; continue := proc # NIL;
  WHILE continue & (mod # NIL) DO
    IF (mod.name[0] # 0X) & ~mod.selected THEN
      pref := mod.pvr; pvadr := Mem[pref];
      WHILE continue & (pvadr # 0) DO r := Mem[pvadr];
        INC(res, proc(pvadr, r, mod.name, continue));
        INC(pref, 4); pvadr := Mem[pref]
      END
    END ;
    mod := mod.next
  END
END FindStaticRefs;
```

Procedures *FindClients*, *FindDynamicRefs* and *FindStaticRefs* are expressed as *generic* object traversal schemes, which accept *parametric* handler procedures that are (directly or indirectly) called for each encountered object with the *source* and *destination* of the (potential) reference itself passed as actual parameters. This allows these procedures to be used for *other* purposes

[12] *In FPGA Oberon, only records can be allocated in the heap. In Experimental Oberon, fixed-length and open arrays can also be dynamically allocated.*
[13] *The original procedure Kernel.Scan (implementing the scan phase of the Oberon garbage collector) has been renamed to Kernel.Collect – analogy to procedure Modules.Collect.*

as well, for example to *enumerate* all references to any given group of modules. In order to omit in module *Kernel* any reference to the list of modules managed in module *Modules*, procedure *Scan* is also expressed in *generic* form. The following is a simplified version of this scheme[14]:

```
1   PROCEDURE Scan*(type, proc: Handler; s: ARRAY OF CHAR; VAR resType, resProc: INTEGER);
2     VAR offadr, offset, p, r, mark, tag, size: LONGINT; continue: BOOLEAN;
3   BEGIN p := heapOrg; resType := 0; resProc := 0; continue := (type # NIL) OR (proc # 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 type # NIL THEN INC(resType, type(p, tag, s, continue)) END ;
10            IF continue & (proc # NIL) THEN offadr := tag + 16; offset := Mem[offadr];
11              WHILE offset # -1 DO (*skip pointers*) INC(offadr, 4); offset := Mem[offadr] END ;
12              INC(offadr, 4); offset := Mem[offadr];
13              WHILE continue & (offset # -1) DO (*procedures*) r := Mem[p+8+offset];
14                INC(resProc, proc(p+8+offset, r, s, continue));
15                INC(offadr, 4); offset := Mem[offadr]
16              END
17            END
18          END
19        END
20      END ;
21      INC(p, size)
22    UNTIL p >= heapLim
23  END Scan;
```

This scheme calls *parametric* handler procedures for individual elements of each marked *heap* object rather than directly checking whether it contains type or procedure variable references to the modules to be unloaded. Procedure *type* is called with the address of (the *type tag* of) the heap object as the *source* of the potential reference to be checked, followed by the address of the referenced type descriptor as its *destination* (line 9). Procedure *proc* is called for each procedure variable in the heap object with the address of the procedure variable as the *source*, followed by the address of the referenced procedure as the *destination* (line 14). The results of these calls are *separately* added up for each handler and returned in the variable parameters *resType* and *resProc*. By convention, a result of zero means that no reference has been found. Apart from that, handler procedures are free to define the semantics of these result parameters. For example, they may decide to enumerate *all* references or stop when the *first* one is found.

An additional boolean variable parameter *continue* allows the handler procedures to indicate to the caller that they are no longer to be called (lines 8, 10, 13). Note that the scan process itself continues, but only to *unmark* the remaining marked heap objects (line 7).

Procedure *Modules.Check* uses procedures *FindClients, FindDynamicRefs* and *FindStaticRefs* by passing its own private handler procedures *HandleClient* and *HandleRef*. These procedures merely set the parameter *continue*, depending on whether a reference has been found or not.

In passing we note that procedure *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 correctly handle visible *and* hidden modules in the data structure managed by module *Modules*.

*[14] The simplified version scans only record blocks allocated via NEW(p), where p is a POINTER TO RECORD. The Experimental Oberon implementation also covers array blocks.*

## Implementation prerequisites

In order to make the outlined validation pass possible, type descriptors of dynamic objects in the heap[15] and descriptors of global module data in static module blocks contain the *locations of all procedure variables* within the described object, adopting an approach employed in an earlier implementation of the Oberon system (MacOberon)[16]. The resulting run-time representations of a heap record with its associated type descriptor and a module block are shown in Figure 2 (the *method table* used to implement Oberon-2 style *type-bound procedures* is not discussed here).

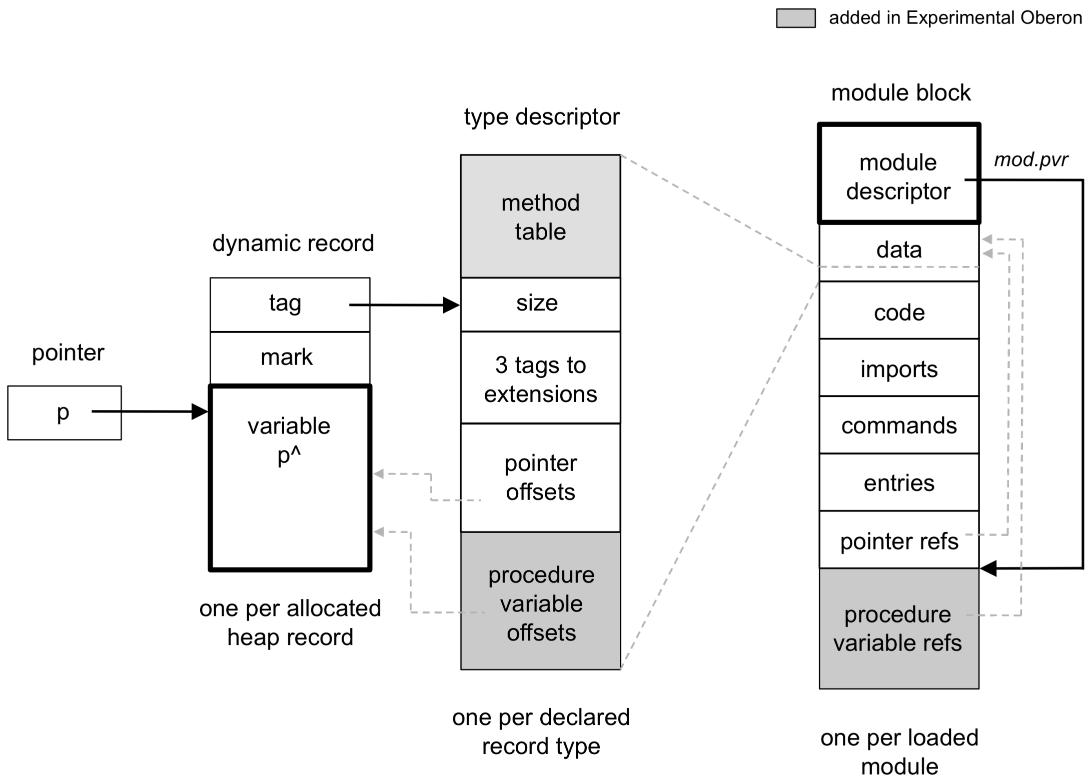


**Figure 2:** Run-time representation of a dynamic record with its type descriptor and a module block in Experimental Oberon

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 entire system for possible procedure variable references to the modules to be unloaded[17].

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 FPGA Oberon, hidden *pointers*, although not exported and therefore invisible in client modules, are included in symbol files because their offsets are needed for garbage collection. Similarly, in Experimental Oberon, hidden *procedure variables* are included in symbol files because their offsets are needed for reference checking prior to module unloading, as shown in the following example:

[15] *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 type, such as its size, information about type extensions and the offsets of pointer and procedure variable fields.*
[16] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*
[17] *If only procedures declared in the same module were allowed to be assigned to global procedure variables, the meta data section headed by the link mod.pvr would not be necessary.*

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

Here the global variable *r* declared in module *M1* is of an imported record type, which contains a *hidden* procedure variable field *r.p*. 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 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* must be known at run time. This is achieved as follows:

- The offset of the hidden field *p* of the record type *M0.R* is included in the symbol file of *M0.*
- The location of the procedure variable *M1.r.p* within the data section of *M1* (computed as the sum of the start address of *M1.r* and the offset of *M0.R.p*) is included in the object file of *M1.*
- 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 in FPGA Oberon). 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*[18]. This choice is reflected in the conditions used to find *all* pointers and procedure variables in various procedures of the compiler[19], as illustrated in the following code excerpt of its symbol table handler (module ORB):

```
TYPE Ptrs* = {Pointer, NilTyp};                 (*NilTyp = hidden pointer variable*)
  Procs* = {Proc, NoTyp};                       (*NoTyp = hidden procedure variable*)

PROCEDURE FindHiddenFields(VAR R: Files.Rider; typ: Type; off: LONGINT);
  VAR fld: Object; i, s: LONGINT;
BEGIN
  IF typ.form IN Ptrs THEN Write(R, Fld); Write(R, 0); Files.WriteNum(R, off)
  ELSIF typ.form IN Procs THEN Write(R, Fld); Write(R, 0); Files.WriteNum(R, -off-1)
  ELSIF typ.form = Record THEN fld := typ.dsc;
    WHILE fld # NIL DO FindHiddenFields(R, fld.type, fld.val + off); fld := fld.next END
  ELSIF typ.form = Array THEN s := typ.base.size;
    FOR i := 0 TO typ.len-1 DO FindHiddenFields(R, typ.base, i*s + off) END
  END
END FindHiddenFields;
```

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

## 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[20].

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 to them in the *scan* phase. Note, however, that reference checking *stops* when the *first* reference is detected and that 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.

## Alternatives considered

Alternative #1:

In the conventional mark-scan scheme outlined above, the initial *mark* phase first marks the heap objects reachable by *other* loaded modules and the *scan* phase subsequently checks for references from the *marked* objects to the modules to be unloaded. In order to improve the efficiency of this scheme, one might be tempted to check for references directly *during* the *mark* phase and simply *stop* marking heap objects as soon as the first reference has been found. While this has the potential to be more efficient, it would lead to a number of complications.

First, we note that the pointer rotation scheme used during the *mark* phase temporarily modifies not only the *mark* field, but also the *pointer variable* fields of the encountered heap objects. In the Oberon system, the Deutsch-Schorr-Waite graph marking algorithm is used[21]. This scheme essentially establishes a *return path* used during future visits of a node, effectively replacing the stack of procedure activations by a stack of marked nodes, i.e. the return path is encoded *in* the heap data structure itself, thereby avoiding recursion and, consequently, the consumption of additional stack space during the heap traversal. Thus, one cannot simply exit the *mark* phase as soon as a reference has been found, but would also need to undo all pointer modifications made up to that point. The easiest way to achieve this is by letting the *mark* phase run to completion. But this would neutralize the desired performance gain.

---

*[20] Adding procedure variable offsets to type descriptors is, strictly speaking, not necessary, as the compiler could 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 additional fields in type descriptors. We 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 FPGA Oberon). 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 would be 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 appears negligible, and therefore the additional effort required to implement this refinement would be hard to justify indeed.*

*[21] Herbert Schorr and William M. Waite, "An efficient machine-independent procedure for garbage collection in various list structures", CACM, 10(8):501-506, August 1967.*

Second, if one wants to omit in module *Kernel* any reference to the data structure managed by module *Modules*, one would also need to express the *mark* phase as a *generic* heap traversal scheme accepting parametric handler procedures for reference checking, similar to the generic heap *scan* scheme outlined above. Such a generalization would open up the possibility for an erroneous (or malicious) handler procedure to prematurely end the *mark* phase, which in turn may leave the heap in an inconsistent and potentially irreparable state. In sum, one seems well advised not to interfere with the *mark* phase.

Finally, one would still need a *scan* phase to *unmark* the already marked portion of the heap.

Alternative #2:

Another possible variant would be to treat all static and dynamic *procedure variables* and *type tags* like *pointers* (which is what they essentially are) and the *procedures* and *type descriptors* referenced by them like *records* during the *mark* phase of reference checking. This could easily be accomplished by making procedures and type descriptors *look* like records, which in turn could be achieved by making them carry a *type tag* and a *mark field*.

These additional fields would be inserted as a prefix to (the code section of) *each* declared procedure and (the type descriptor of) *each* declared record type in the module block. All static *procedures* would share a common “procedure descriptor” and all *type descriptors* a common “meta-type descriptor”. These descriptors would contain no tags to extensions and no pointer or procedure variable offsets. Consequently, they can be represented by one and the same shared *global* meta descriptor, which could be stored at a fixed location in the module area for example. The resulting run-time representation of *procedures* and *type descriptors* is shown in Figure 3.

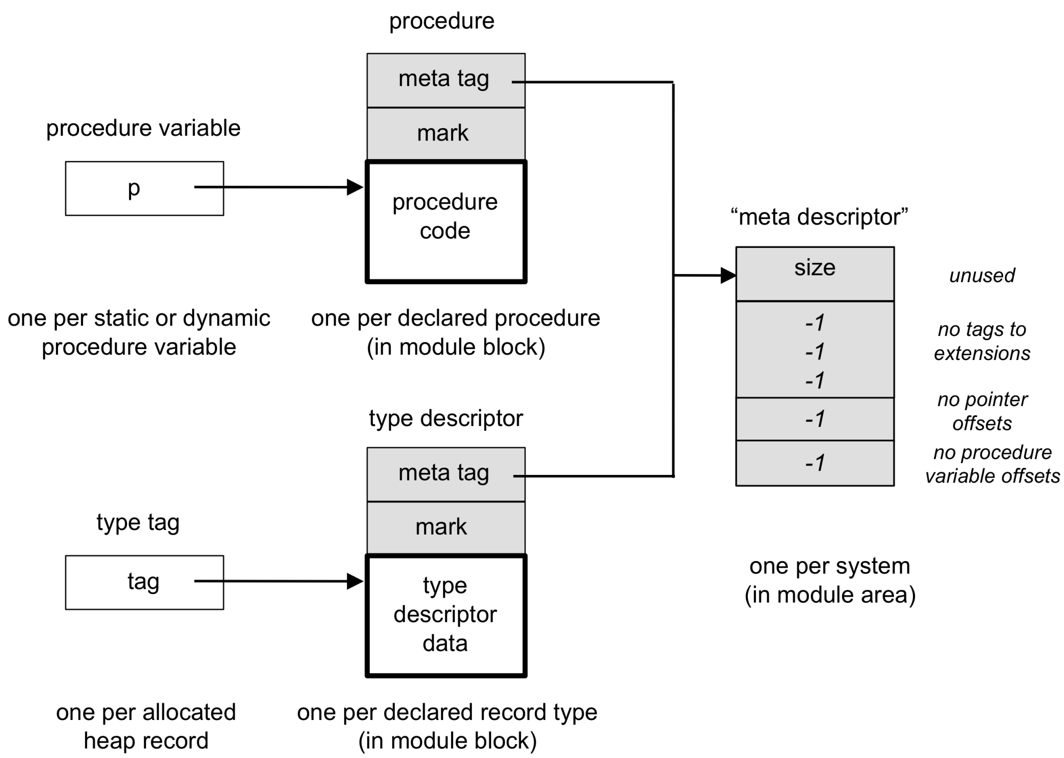


**Figure 3:** Procedure variable and type tag interpreted as *pointer*, and procedure and type descriptor interpreted as *record*

Alternative #3:

A simpler variant would be to treat procedure variables and type tags as *special cases* during the *mark* phase, eliminating the need for a *meta* tag field as well as the shared *meta* descriptor. The resulting run-time representation of *procedures* and *type descriptors* is shown in Figure 4.

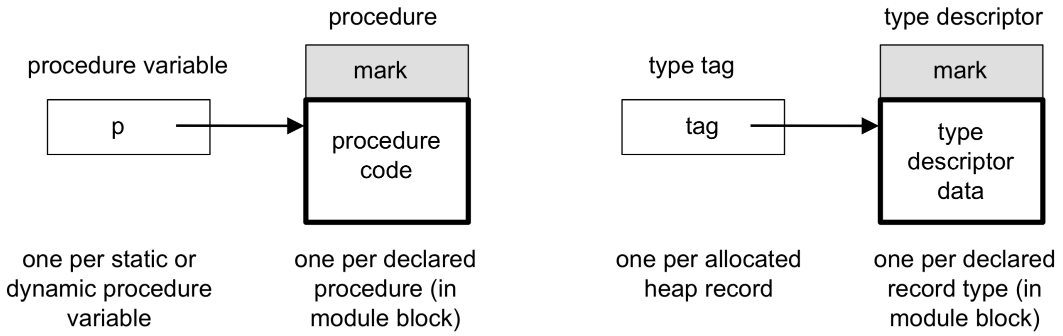


**Figure 4:** Procedures and type descriptors with an additional *mark* field

With these preparations, the *mark* phase could be suitably *extended* to also include *procedure variables* and *type tags* in the list of "pointers" to be traversed. In sum, the mark phase would not only "touch" all reachable objects in the *heap*, but also in the *module blocks*.

While this technique appears appealing at first, a few points are worth mentioning. First, the *extended* mark phase requires an extra *mark* field to be inserted as a prefix to *each* procedure and *each* type descriptor in the module block. Given that most procedures and type descriptors are never referenced, this appears to be overkill. One could therefore decide to add the *mark* field only to *module descriptors* rather than to individual *procedures* or *type descriptors* during the *mark* phase. While this would make the check whether any of the selected modules is referenced a trivial task, it would render the *mark* phase more complex, as it would now need to *locate* the module descriptor belonging to a given procedure or type descriptor (on systems where additional meta-information is present in the run-time representation of loaded modules, such as the locations of procedures and type descriptors in module blocks or *back pointers* from each object of a module to its module descriptor, the mark phase would be simpler; however, neither FPGA Oberon nor Experimental Oberon offer such "metaprogramming" facilities).

Second, one would still need to mark *all* objects, for the same reason as outlined above, i.e. one cannot simply exit in the middle of the *mark* phase without additional action.

Third, one would also still need an extra *scan* phase to unmark the marked portion of the heap.

Finally, a comparison of the code required to implement the various alternatives showed that our solution is *by far* the simplest. The combined implementation cost of *all* modifications to the runtime representation of type descriptors and descriptors of global module data, the object and symbol file formats and the module loader, is only about 15 source lines of code[22], while the *reference checking* phase itself amounts to less than 75 lines (the *total* implementation cost to realize *safe module unloading*, including the ability to unload module *groups* and the automatic collection of no longer referenced hidden modules, amounts to 250 source lines of code).

[22] *See procedures ORB.InType, ORB.FindHiddenFields, ORG.BuildTD, ORG.Close and Modules.Load.*

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

In most Oberon implementations, only *client* references are checked prior to module unloading. *Type* and *procedure variable* references are usually not checked, although various approaches are typically employed to address the case where such references exist. These approaches can be broadly grouped into two main categories: (a) schemes that explicitly allow invalidating references, and (b) schemes where all references must remain unaffected.

*a. Schemes that explicitly allow invalidating references*

In such schemes, unloading a module from memory *may*, and in general *will*, lead to "dangling" *type* and *procedure variable* references pointing to module data that is no longer valid[23]. 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* or *procedure variable* references:

1. On systems that use a *memory management unit* to perform virtual memory management, such as Ceres-1 or Ceres-2, 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* or a *procedure variable* references point to a now *unallocated* page, and consequently any attempt to access such a page later will 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, after having unloaded it, they still don't know whether references from other loaded modules still exist – until a *trap* occurs. But then it is usually 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.

   Note that if the language Oberon featured the concept of module *finalization*, a module could 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 could be used to close any viewers that may have been opened or delete any data elements that may have been created by a module to be unloaded, for example. However, *finalization* has, regrettably, never been introduced as a language feature in Oberon.

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

2. On systems that do *not* use virtual memory, such as Ceres-3 or FPGA Oberon on RISC, the easiest way to cope with dangling references is to simply *ignore* them, i.e. 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 will become *unstable* the moment another module loaded later *overwrites* the previously released module block *and* other loaded modules still refer to its *type descriptors* or *procedures*. 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 Experimental Oberon, but was later discarded, mainly because the resulting effect on the *overall* behavior of a running system is essentially impossible to predict (or even detect) by the user. 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”, 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, instead of locating and modifying each individual procedure *call* anywhere in the system.

   Note, however, that using a link table to implement indirection for procedure calls is viable only on systems with *efficient* hardware support for it. On such systems, 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. Indirection for procedure calls via a link table was used in some of the earlier versions of the Oberon system on Ceres 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[24]. Later versions of the NS processor, however, internally re-implemented the *same* instruction using microcode, which negatively impacted its performance[25]. For this and a variety of other reasons, the *CXP k* instruction – and with it the *link table* – were no longer used in later versions of the Oberon system.

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 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 M’ *cannot* be inserted in a data structure rooted in a variable *v* of the imported type T – precisely *because* their declared types are not extensions of T (in the language Oberon, 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

[24] *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 module 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.*

[25] *The internal re-implementation of the CXP instruction using microcode in later versions of the NS processor followed the general industry trend – starting in the mid 1980s – of implementing only frequent, simple instructions directly with hardware, while interpreting more complex instructions using internal microcode. In general, with the advent of highly regular reduced instruction set computers (RISCs) around that time, the trend towards offering microprocessors providing a smaller set of simple instructions, most of them executing in a single clock cycle, combined with fairly large banks of (fast) registers, continued – and does so to this date.*

extensions T' of an imported type T. The flip side of such a rule, however, is that modules that actually declare such types can *never* be unloaded – unless, of course, other ways to safely unload such modules are implemented. Also, procedure variable references are *not* covered by this rule.

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 *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 procedure 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. A similar problem may occur with references to *type descriptors*, if they are not persisted in memory *after* unloading their associated modules.

*b. Schemes where all references must remain unaffected*

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 of course simply exit the *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 dynamically allocated heap records a simple solution exists: allocate them *outside* their module blocks in order to persist them beyond the lifetime of their associated module. One possibility is to allocate them in the *heap* itself at module load time[26]. This approach has been implemented in Ceres-Oberon, for instance. It eliminates dangling *type* references altogether and therefore also the *need* to check for them at run time, because type descriptors are unaffected by module unloading. Note that such dynamically created type descriptors can be deallocated only if (a) their associated module has already been unloaded and (b) no *regular* heap objects refer to them via their *type tags*.

   For *procedures* referenced by *procedure variables* in either static or dynamically allocated objects no such simple solution exists. The only way to "persist" procedures would be to persist the *entire* module (recall that procedures may *access* global module data or *call* other procedures declared in the same module).

[26] *Note that one cannot simply move type descriptors around in memory, as their addresses are (typically) used to implement type tests and type guards. By allocating them in the heap at module load time, one avoids the need to move them to a different location when a module is unloaded.*

We conclude that if one wants to handle type *and* procedure variable references, one *cannot* unload a module block from memory, as long as references to it still exist[27].

3. One way to automatically persist type descriptors *and* procedures consists of simply *never* releasing the module block of a module once it is loaded. Instead, when the user requests the unloading of a module, it is only removed from the *list* of loaded modules. This amounts to *renaming* the module, with the implication that a newer version of the same module (with the same name) can be reloaded again. This method has been chosen in MacOberon[28], for example. Since the associated memory of a module is never released, the issue of dangling type or procedure variable references is avoided altogether, as they simply cannot exist. However, it can also lead to higher-than-necessary memory usage if a module is repeatedly loaded and unloaded (typical during *development*). Nevertheless, such an approach may be viewed as adequate on *production* or *server* systems, where module unloading is rare.

4. A further *refinement* of the approach outlined above consists of *initially* removing a module only from the module list (as in MacOberon), but *in addition* releasing its associated memory *as soon as* there are no more *type* or *procedure variable* references to it. If this is done in an automatic fashion (for example as part of a background process), 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 Experimental Oberon. A variation of it was used in some versions of SparcOberon[29].

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 typically no need to keep multiple copies 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 module blocks holding their program code and global variables. Hence, not releasing module blocks immediately after a module *unload* operation typically has a rather negligible impact on overall memory consumption.

* * *

*[27] Of course, "mixed" variants are also possible. For example, one could allocate type descriptors in the heap at module load time (as in 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 a l s o 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 s a m e approach for both type and procedure variable references.*

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

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