Safe module unloading

in the Oberon operating system

Andreas Pirklbauer

1.6.2019

Purpose

This technical note describes a possible 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 operating 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.

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, i.e. if clients exist among the other loaded modules, a module or module group is not unloaded from the system. *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 been *unsafe* in the Oberon operating system, as removing a module from memory *may*, and in general *will*, lead to “dangling” references, i.e. references that point to module data that is no longer valid. This means that the system 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*.

---

[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. Types can be declared globally (in which case they can be exported and referenced by name in client modules) or local to a procedure (in which case they cannot be exported). Variables can be declared as global variables (allocated in the module area when a module is loaded) or as local variables (allocated on the stack when a procedure is called). Anonymous variables with no explicit name declared in the program can be dynamically allocated in the heap via the predefined procedure NEW. Procedures can be declared as global or local procedures, and can be assigned to procedure variables. Thus, in general there can be type, variable, procedure and procedure variable references from static or dynamic objects of other modules to static or dynamic objects of the modules to be unloaded. However, only dynamic type references and static and dynamic procedure variable references need to be checked during module unloading for the following reasons: First, static type, variable or procedure references from other modules can only refer by name to types, variables or procedures declared in the modules to be unloaded. Such references are already handled via their import/export relationship during module unloading (if clients exist, a module or module group is never unloaded) and therefore don't need to be checked separately. Second, dynamic pointer references from global or dynamic pointer variables of other modules to dynamic objects reachable by the modules to be unloaded should not be checked, as they should not prevent module unloading. In the Oberon system, such references will be handled by the garbage collector during a future garbage collection cycle, i.e. heap records reachable by the just unloaded modules and other still loaded modules will not be collected, whereas heap records that were reachable only by the unloaded modules will be collected – as they should. Thus, the handling of pointer references is delegated to the garbage collector. Finally, pointer variable references to statically declared objects are only possible by resorting to low-level facilities and should be avoided – and, in fact, be disallowed (pointers should point exclusively to anonymous variables allocated when needed during program execution).*

[4] *See http://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/Historical-notes-on-module-unloading-in-Oberon.pdf*

Implementing safe module unloading

In Experimental Oberon, a revision of FPGA Oberon[5], *all* possible types of references to a loaded module or module group are checked as follows prior to module *unloading* (Figure 1):

- If clients exist among the 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 containing the references that caused the removal to fail. If, however, the *force* option /f is specified[6], the modules are initially removed (only) from the *list* of loaded modules, without releasing their associated memory. Such "hidden" modules are later physically removed from *memory*, as soon as there are no more references to them from anywhere in the system.

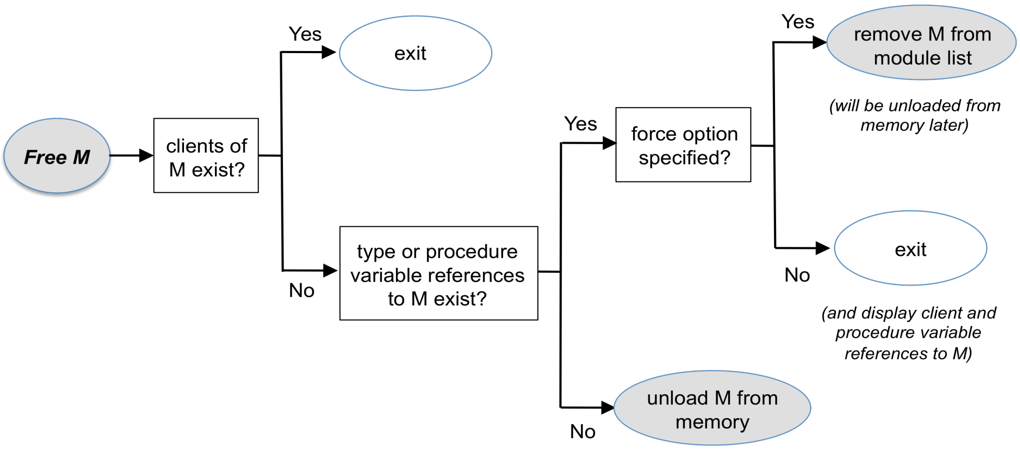

Figure 1: Safe module unloading in Experimental Oberon

Removing a module only from the *list* of loaded modules 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]. Unloading a module from *memory* frees up the memory area previously occupied by the module block[8]. To automate the unloading of no longer referenced *hidden* module data, a command *Modules.Collect* has been 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. The tool command *System.Collect* also invokes *Modules.Collect*.

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

[5] *http://www.projectoberon.com*
[6] *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*
[7] *Modules removed only from the list of loaded modules, but not from memory, 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 2013 on RISC and in Experimental Oberon, the module block includes the module's type descriptors. In some other Oberon implementations, such as Oberon on Ceres, type descriptors are not stored in the module block, but are dynamically 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 special precaution is necessary, as module blocks are removed only from the list of loaded modules and not from memory, if they are still referenced by other modules. Thus, type descriptors can safely be stored in the (static) module blocks.*

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 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[10]. Such elements typically contain both *type* references (type tags) and *procedure variable* references (handler procedures) referring to M’. This is common in the Oberon viewer system, for example, where M is module *Viewers*.

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[11]. 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. Thus, the recommended way to unload modules is to use the 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 unload a module group, the module *unload* command *selects* the modules to be unloaded using procedure *Modules.Select* and then invokes procedure *Modules.Check*. Clients are checked first. This is done by verifying whether *unselected* modules import *selected* modules[12]:

```
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 – as it should.

[10] Note that for t y p e references, it is actually possible to know at compile time, whether a module M may potentially lead to references from other modules at run time (but not whether it actually will): If M does not declare record types which are extensions T’ of an imported type T, records declared in M cannot be inserted in a data structure rooted in a variable of an imported type T – precisely because they are not extensions of T (in the Oberon language, the assignment p := p’ is allowed only if the type of p’ is the same as the type of p or an extension of it).

[11] 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 in fact 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 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 w e r e 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 – as it should.

[12] 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 loaded modules, no further action is taken and the *unload* command exits. If no clients exist, *type* and *procedure variable* references are checked next. References from *dynamic* objects in the heap are checked 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 initial *mark* phase, heap records reachable by all *named* global pointer variables of *all other* loaded modules are marked (line 5), thereby excluding records reachable *only* by the specified modules themselves. This automatically recognizes module *groups* and ensures that when a module or module group is referenced *only* by itself, it can still be unloaded. The *scan phase* (line 6 or 10), implemented as a separate procedure *Scan*[13] in module *Kernel*, scans the heap sequentially, unmarks all *marked* records and checks whether the *type tags* of the marked records point to descriptors of types declared in the *selected* modules, and whether *procedure variables* declared in these heap records refer to procedures declared in those same modules.

An additional boolean parameter *all* allows the caller to specify whether the *mark* phase should first mark all heap records that are reachable by *all* other loaded modules before initiating the *scan* phase *once* (used for module unloading), or whether the *scan* phase should be initiated for each module separately, after marking the heap records reachable by it (used for enumerating the references to each module *individually*).

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;
```

Note that the procedures *FindClients*, *FindDynamicRefs* and *FindStaticRefs* are expressed as *generic* object traversal schemes which accept *parametric* handler procedures that are called for each encountered object. This allows these procedures to be used for *other* purposes as

[13] *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.*

well, for example to *count* or *enumerate* the client, type or procedure variable references to *any* given (loaded) module or module group.

In order to omit in module *Kernel* any reference to the module list rooted in module *Modules*, procedure *Kernel.Scan* is also expressed as a *generic* heap scan scheme. 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 record rather than *directly* checking whether these records contain type or procedure variable references to the modules to be unloaded. Procedure *type* is called with the memory address of the *heap record* itself as the *source* of the reference followed by the *type tag* (address of the type descriptor) as its *destination* (line 9). Procedure *proc* is called for each procedure variable declared in the same record 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 handler calls are *separately* added up for each handler procedure and returned in the variable parameters *resType* and *resProc*.

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

Procedure *Modules.Check* uses procedures *FindClients, FindDynamicRefs* and *FindStaticRefs* by passing its private handler procedures *HandleClient* and *HandleRef*.

We emphasize 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 written such that it can correctly handle *both* visible *and* hidden modules in the module data structure rooted in module *Modules*.

---

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

Implementation prerequisites

In order to make the outlined validation pass possible, type descriptors of *dynamic* objects[15] allocated in the *heap* as well as the descriptors of *global* module data located in *static* module blocks contain a list of *procedure variable offsets*, adopting an approach employed in one of the earlier implementations of the Original Oberon system (MacOberon)[16]. The resulting run-time representation of a *dynamic* record and its associated type descriptor is shown in Figure 2 (the *method table* used to implement Oberon-2 style *type-bound procedures* is not discussed here).

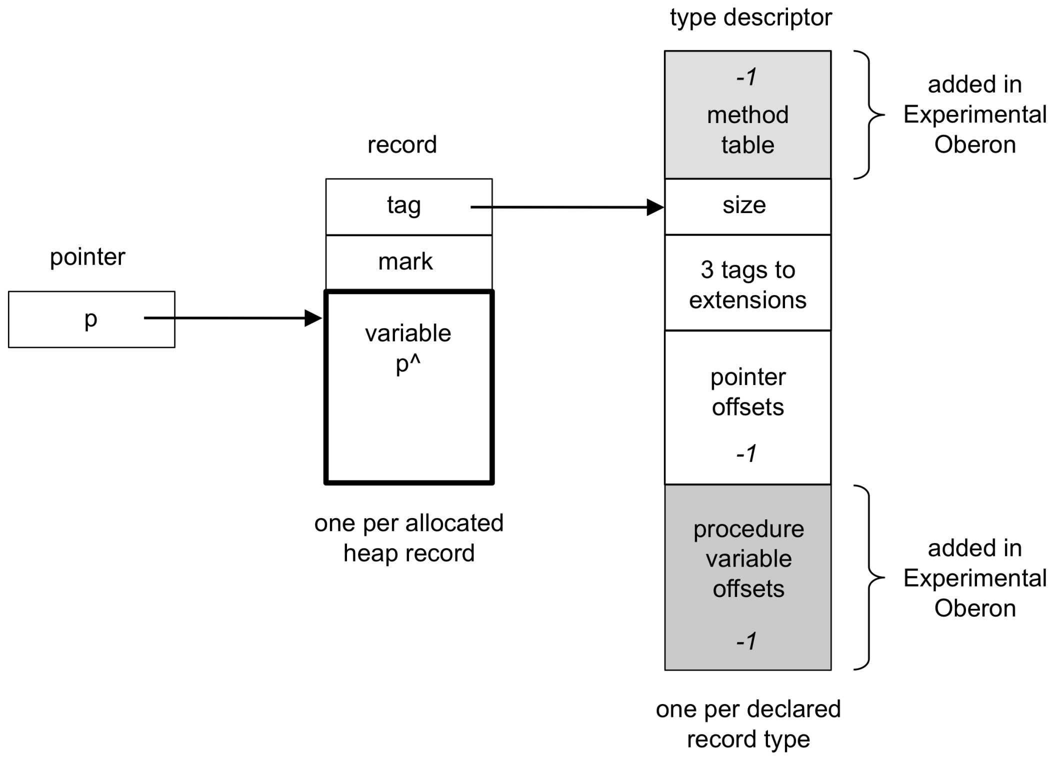

Figure 2: Run-time representation of a dynamic record and its type descriptor in Experimental Oberon

The descriptors also contain the offsets of *hidden* (not exported) procedure variables, enabling the module *unload* command to check *all* possible procedure variable locations in the entire system for possible procedure variable references to the modules to be unloaded.

To make the offsets of hidden *procedure variables* in exported record types available to client modules, symbol files also include them. An importing module may, for example, declare a global variable of an imported record type, which contains *hidden* procedure variables, or declare a record type that contains or extends an imported one. We recall that in Original Oberon, 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[17]. Similarly, in Experimental Oberon, the locations of visible *and* hidden procedure variables are needed for reference checking during module unloading, as shown in the following example.

---

[15] *See chapter 8.2, page 109, of the book Project Oberon 2013 Edition for a detailed description of an Oberon type descriptor. A type descriptor contains certain information about dynamically allocated records that is s h a r e d by all allocated objects of the same record type (such as its size, information about type extensions and the offsets of pointer fields within the record).*
[16] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*
[17] *See chapter 12.6.2, page 41, of the book Project Oberon 2013 Edition for a description of why the offsets of hidden pointers are needed during garbage collection.*

```
MODULE M0;
  TYPE Proc* = PROCEDURE; Rec* = RECORD p*, q: Proc END ;   (*q is a hidden field*)
  PROCEDURE Q*(r: Rec); BEGIN r.q END Q;
  PROCEDURE Init*(VAR r: Rec; p, q: Proc); BEGIN r.p := p; r.q := q END Init;
END M0.

MODULE M1;
  IMPORT M0;
  VAR r: M0.Rec;
  PROCEDURE Init*(p, q: M0.Proc); BEGIN M0.Init(r, p, q) END Init;
END M1.

MODULE M2;
  IMPORT M1;
  PROCEDURE Q; BEGIN END Q;
  PROCEDURE Set1*; BEGIN M1.Init(NIL, NIL) END Set1;
  PROCEDURE Set2*; BEGIN M1.Init(NIL, Q) END Set2;
END M2.

M2.Set1  System.Free M2 ~        … can unload M2
M2.Set2  System.Free M2 ~        … can't unload M2 (as M2.Q is referenced in global procedure variable M1.r.q)
```

Here the global record *r* declared in module *M1* contains a hidden field *M1.r.q* that is accessible only in the exporting module *M0*. If another module *M2* installs a procedure *M2.Q* in this hidden record field, a procedure variable reference from *M1* to *M2* is created, with the (intended) effect that module *M2* can no longer be unloaded. In order to be able to check for such references at run time, the record field offset of the field *M1.r.q* must be available when compiling *M1*[18].

Hidden *pointers* are included in symbol files without their names, and their (imported) type in the symbol table is of the form *ORB.NilTyp*. Hidden *procedure variables* are also included without their names, and their symbol table type is of the form *ORB.NoTyp*[19]. For additional details, 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, modules now also contain an *additional* section in the module block 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 thus 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 record encountered in the *mark* phase *all* modules to be

---

[18] The compiler will include the record field offset in the object file of M1, from where the module loader will transfer it to the corresponding array in the module's meta data section of M1.
[19] This is acceptable because for record fields, the types ORB.NilTyp and ORB.NoTyp are not used otherwise.
[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 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 out" 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 declared record type rather than one per allocated heap record. Most applications are programmed in the conventional programming style, where installed procedures are rare. For example, in the Oberon operating 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 type of viewer. In sum, the benefit obtained by saving a few fields in a relatively small number of type descriptors appears negligible, and the additional effort required to implement this refinement would be hard to justify.

unloaded are checked for references during the *scan* phase. Note also that reference checking *stops* when the *first* reference is detected, and module unloading is rare except, perhaps, during development, where however the *number* of references tends to be small.

Thus, the presented solution, which was mainly chosen for its simplicity, appears to be amply sufficient for most practical purposes.

Alternatives considered

Alternative #1:

In the conventional mark-scan scheme outlined above, the initial *mark* phase first marks the heap objects reachable by *all other* loaded modules and the *scan* phase subsequently checks for references from the *marked* records to the modules to be unloaded. In order to further improve the efficiency of this scheme, one might be tempted to check for references directly *during* the *mark* phase and simply *stop* marking records as soon as the first reference is 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 records. In the Oberon operating system, the Deutsch-Schorr-Waite graph marking algorithm[21] is used for this purpose. 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. Thus, one cannot simply exit the *mark* phase when a reference is 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.

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. While this would be straightforward to implement, such a generalization would open up the possibility for an erroneous 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 meddle 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 *procedure variables* and *type tags* like *pointers*, and *procedures* and *type descriptors* referenced by them like *records* during the *mark* phase of reference checking. This could be achieved by making procedures and type descriptors *look like* records, which could be accomplished 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 offsets. Consequently, they can be represented by one and the same shared global "meta

[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.*

descriptor", which could be stored at a fixed location within the module area for example. The resulting run-time representation of *procedures* and *type descriptors* is shown in Figure 3.

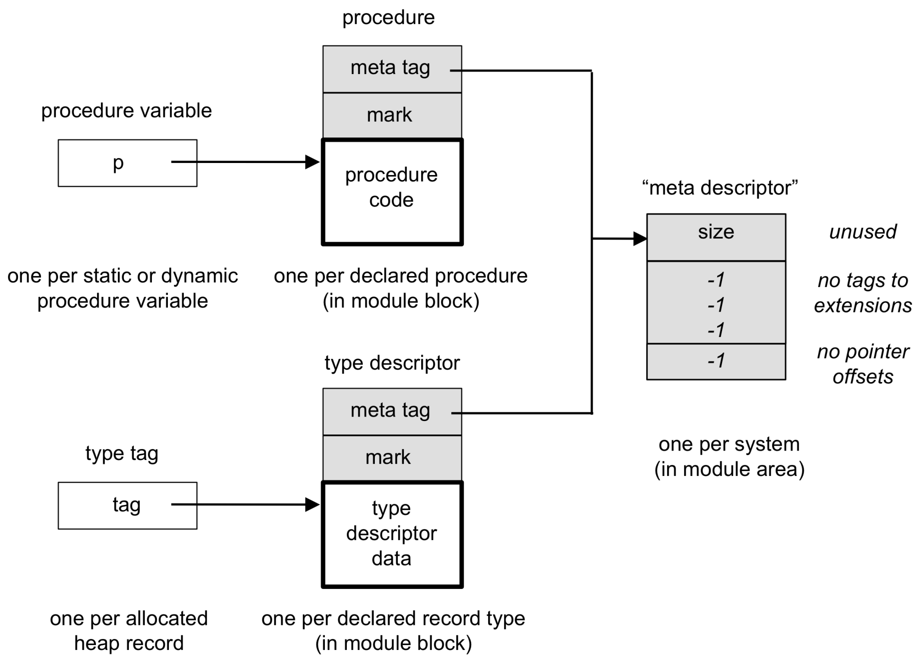

Figure 3: Procedure variable and type tag interpreted as *pointer*, and procedure and type descriptor interpreted as *record*

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.

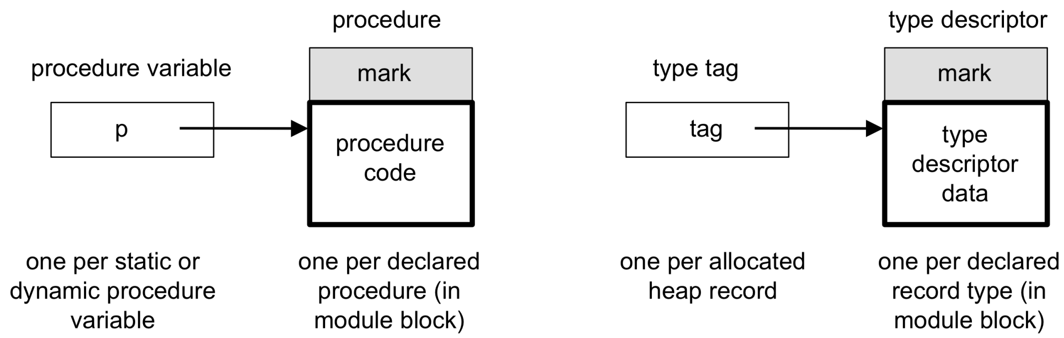

Figure 4: Procedures and type descriptors with an additional *mark* field

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

While this technique appears appealing at first glance, 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 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 2013 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 about 250 source lines of code).

* * *

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