Historical notes on module unloading in the Oberon system

Andreas Pirklbauer

1.9.2018

In the Oberon system[1], there exist three possible types of references to a loaded module M:

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

It is easy to convince oneself that these are indeed the *only* types of references that *need* to be checked prior to unloading module M[2]. In most implementations of the Oberon system, *client* references are checked prior to module unloading, i.e. if clients exist among the remaining loaded modules, a module or module group is not unloaded.

As for the remaining two types of possible references (*type* and *procedure* references), there exist two essentially different interpretations of the semantics of module unloading:

- Schemes that explicitly *allow* invalidating past type and procedure references
- Schemes where past type and procedure references remain *unaffected*

Schemes that explicitly allow invalidating past references

In such schemes, 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memory *may*, and in general *will*, lead to "dangling" references, i.e. references that point to module data that is no longer valid. Such references can be in the form of *type tags* (addresses of type descriptors) in *dynamic* objects or in the form of *procedure variables* installed in *static* or *dynamic* objects[3].

An important use case is when a structure rooted in a variable of base type T declared in a base module M (for example module *Viewers*) contains elements of an extension T' defined in a client module M' (for example a graphics editor), which is then unloaded. Such elements

---

[1] *http://www.projectoberon.com*

[2] *An Oberon module can be viewed as a container of types, variables and procedures, where variables can be procedure-typed. Types can be declared as g l o b a l types (in which case they can be exported and referenced in client modules) or as types l o c a l to a procedure (in which case they cannot be exported). Variables can be statically declared as g l o b a l variables (allocated in the module area when a module is loaded) or as l o c a l variables (allocated on the stack when a procedure is called), or they can be dynamically allocated in the h e a p via the predefined procedure NEW. Thus, in general there can be type, variable or procedure references from static or dynamic objects of other modules to static or dynamic objects of the specified modules to be unloaded. However, only d y n a m i c t y p e references and s t a t i c and d y n a m i c p r o c e d u r e references need to be checked during module unloading, for the following reasons. First, s t a t i c type and variable references from other loaded modules can only refer by n a m e to types or variables 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, d y n a m i c variable references from global or dynamic p o i n t e r variables of other modules to d y n a m i c objects reachable by the modules to be unloaded don't need to be checked either, as such references s h o u l d not prevent module unloading. In the Oberon operating 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 a n d other still loaded modules are not collected, while heap records that w e r e reachable o n l y by the just unloaded modules w i l l be collected – as they should. Thus, the handling of dynamic pointer references is delegated to the garbage collector. Finally, p o i n t e r variable references to s t a t i c objects declared as global variables are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).*

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

typically contain both *type* references (type tags) and *procedure* references (installed handler procedures) that still refer to M'.

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

A variety of approaches have been used in various implementations of the Oberon system to cope with the introduced dangling *type* or *procedure* references:

1. The easiest way to cope with dangling references is to simply *ignore* them. This is the approach chosen in FPGA Oberon on RISC, where the memory associated with a module to be unloaded is *always* released (unless clients exist) without taking any further precautions. But this leaves the system in an *unsafe* state. It will become *unstable* the very 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.

2. Another approach, which can however be used only for *procedure* references, is to identify *all* procedure references to the module M to be unloaded and make them refer to a "dummy" procedure, preventing a run-time error, when such "fixed up" procedures are called later. This simple solution was tested in an earlier version of Experimental Oberon, but was later discarded, mainly because the resulting effect on the *overall* behavior of the system would be essentially impossible to predict (or even detect) by the end 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 mouse tracking if the unloaded module contained a viewer handler, for example.

3. The same effect can also be achieved by using *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 table – which the caller consults for every procedure *call*, in order to obtain the actual memory location of the called procedure. On systems that use such a link table, one can simply set the *link table entries* for all referenced *procedures* of a module to be unloaded to *dummy* entries, instead of modifying each individual procedure *call* anywhere in the system.

   We note that using a link table to implement indirection for procedure calls is only viable on systems that provide *efficient* hardware support for it. It has been used in some of the earlier versions of the Oberon system on Ceres computers, which were based on the (now defunct) NS32000 processor. Early versions of this processor featured efficient *hardware* support for the link table in the form of a *call external procedure* (CXP k) instruction (where k is the index of the link table entry of the called procedure), which sped up the process of activating external procedures significantly[4]. Later versions of the NS processor, however, internally re-implemented the same processor instruction using microcode. This negatively impacted its performance – so much that it became slower than the *regular branch to subroutine* (BSR) instruction[5]. Therefore, the CXP k instruction – and with it the *link table* – were no longer used in later versions of Ceres-Oberon, for example on Ceres-3.

---

[4] 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 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.

[5] The internal re-implementation of the CXP instruction using microcode in later versions of the NS processor followed the general industry trend 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) in the 1980s and 1990s, 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.

4. On systems that use a *memory management unit* to perform virtual memory management, such as on Ceres-1 or Ceres-2, another possible approach is to simply *unmap* the module space of an unloaded module from virtual memory, thereby *invalidating* future references to it. *After* that, a dangling reference points to a now unallocated page, and consequently any attempt to access this page, for example via *type* or *procedure* references, results in a *trap* on Ceres-1 and Ceres-2, thereby preventing a system crash.

   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 have existed or still exist – until a *trap* occurs. Only then they know.

   Another disadvantage of this solution is that it requires special hardware support, which may not be available on all systems. Indeed, on Ceres-3, which does not use virtual memory, an attempt to access an unloaded module M goes undetected *initially* – until a *trap* or, worse, a *crash* occurs later. This can, and usually *will*, happen, the moment another module is loaded into the module block previously occupied by the unloaded module M *and* the overwritten data is still *referenced* by other modules[6].

5. Finally, we add the remark that for *type* references, it is actually possible to determine at *compile* time, whether a module M will *not* lead to references from other loaded modules at run time: namely, if module M does *not* declare record types which are extensions T' of an imported type T, then records declared in M *cannot* ever be inserted in a data structure rooted in a variable *v* of an imported type T – precisely *because* they are not extensions of T (in the Oberon programming 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).

   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.

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 removal (from memory) of still referenced module data.

The main issue is that the moment one *allows* modules to release their associated memory if references to them still exist, the resulting *dangling* references must be "fixed up" *somehow*, in order to prevent an almost certain system *crash* (when their module block is overwritten).

However, fixing up 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 loaded modules may *critically* depend on the removed functionality.

For example, unloading a module that contains a handler procedure of a *contents frame* may render it impossible to *close* the enclosing *menu viewer* that contains it. If the approach chosen to handle dangling procedure references involves generating a trap[7], the trap itself will actually

[6] *Of course, one could adapt the module loader to never overwrite a previously "unloaded" module block, even if a new module to be loaded would fit, but this would be a waste of memory .*
[7] *In that case, if the enclosing menu viewer attempts to send a "close" message to the sub frame by calling its handler, a trap is generated, if the module has been unloaded before.*

prevent a system crash (as intended), but the user may *still* need to reboot the system in order to recover an environment without any "frozen" parts, e.g. still open viewers[8].

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

Schemes where past references remain unaffected

The second possible interpretation of *unloading* a module consists of schemes where *past* references *must* at all times remain *unaffected*. In such schemes, module unloading can be viewed as an implicit mandate to preserve "critical module data", as long as references 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 allowing the user to *persist* any still referenced module data to a "safe" location before unloading the associated module.

   For *type* references (type tags referring to type descriptors) an easy solution exists: simply allocate type descriptors *outside* module blocks, in order to persist them beyond the lifetime of their associated module. One possibility is to allocate them in the *heap* at module load time. This has been implemented in Oberon on Ceres-3. Note that this method eliminates dangling *type* references and therefore also the *need* to check for them at run time.

   For *procedure* references (procedure variables referring to procedures) no such simple solution exists. The only way to persist procedures would be to persist the *entire* module (as procedures may *access* global module data or *call* other procedures of the same module).

   We conclude that *if* one wants to address both type *and* procedure references, one *cannot* unload the module block from memory, as long as references to it still exist[9].

3. One possible approach consists of simply *not* releasing the associated memory of a module to be unloaded, but removing it only from the *list* of currently loaded modules. This amounts to *renaming* the module[10], with the implication that a newer version of the same module with the same name can be reloaded again. Such an approach has been implemented in MacOberon[11], for example. Since the associated memory of a module is *never* released, the issue of dangling type or procedure 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 (which is typical during *development*).

   Nevertheless, such an approach may be viewed as adequate on *production* systems or on systems that use *virtual memory* with *demand paging*. On such systems, the virtual address space is (practically) unlimited and data that is accessed only infrequently is temporarily *swapped out* from main to secondary store. Thus, once a module has been removed from the module list, its module block is – over time – less likely to be accessed and hence more

[8] *The module could of course provide a "close" command, which also accepts the marked viewer as argument (using procedure Oberon.MarkedViewer), but that is not necessarily the case.*
[9] *Of course, a "mixed" variant is also possible, namely to allocate type descriptors in the heap, and preserve a module block only if procedure references exist; however, most modules that are referenced by type tags are also referenced by procedures – this is in fact the typical case for records with installed handlers. Thus, this would be an "optimization" in the wrong place.*
[10] *In a specific implementation, one might choose to make the module completely anonymous or modify the name such that one can no longer import it (e.g., by inserting an asterisk).*
[11] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

likely to be placed in secondary store by the demand paging mechanism, freeing up valuable memory resources for modules loaded later. Of course, if the *swapped out* module is accessed later by another module that still references it, it will have to be swapped back in.

Note that with the advent of large primary stores, the concept of virtual memory with demand paging has lost much of its significance. Therefore it is not necessarily used on all systems. On systems that do *not* use virtual memory (such as Ceres-3 or FPGA Oberon on RISC), other mechanisms to handle no longer referenced module data must be considered, if one wants to avoid running out of memory *eventually*.

4. A *refinement* of the approach outlined above consists of *initially* removing a module from the *list* of loaded modules (as is done in MacOberon), but *in addition* releasing its associated memory *as soon as* there are no more 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 one of the later versions of SparcOberon[12]. Of course, it also works on systems that *do* use a memory management unit to perform virtual memory management (it simply optimizes them).

In sum, schemes were past references remain *unaffected* avoid many of the complications that are inherent in schemes that explicitly *allow* invalidating past references. A (small) price to pay is to keep loaded modules 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, note that on modern computers, the amount of available memory – and consequently the amount of dynamically allocated *data* handled by loaded modules – typically *far* exceeds the size of their module blocks.

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