Historical Notes on Module Unloading in the Oberon System

Andreas Pirklbauer

21.2.2017

In Oberon, references to a loaded module can be in the form of *type tags* (addresses of type descriptors) in *dynamic* objects of other modules pointing to descriptors of types declared in the specified module, or in the form of *procedure variables* installed in *static or dynamic* objects of other modules referring to procedures declared in that module[1].

There exist two essentially different interpretations of the *semantics* of module unloading: schemes where *past* references *must* remain unaffected, and schemes that explicitly *allow* invalidating past references.

In the former, module unloading can be viewed as an implicit mandate to *preserve* module data in memory, as long as references to it exist. One could of course always simply exit the unload command with an error message, if such references are detected. The user, however, may then be "stuck" with loaded 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 placing the referenced module data in a "safe" location before unloading the associated module or, alternatively, by removing the module only from the *list* of loaded modules *without* releasing its associated memory. The latter solution effectively amounts to "renaming" the module[3], with the effect that a new version of the module with the same original name can be reloaded again. This approach has been chosen in MacOberon[4]. Since modules are *never* physically removed from memory, the problem of dangling references is avoided, as they simply cannot exist. However, it can also lead to higher-than-necessary memory usage, if, for example, the same module is repeatedly loaded and unloaded (which is typical during development), but may be deemed appropriate on production systems that use virtual memory with demand paging.

To address the issue of eventually running out of memory, a further refinement consists of initially removing a module only from the list of loaded modules (as in MacOberon), but *in addition* automatically (or manually) releasing its associated memory as soon as there are no more references to it. This is the approach chosen in Experimental Oberon. A variant of it was also used in one of the later versions of SparcOberon[5].

The second possible interpretation of *unloading* a module explicitly *allows* invalidating past references. This means that physically removing a loaded module from memory may lead to *dangling references* in the form of *type tags* pointing to descriptors of types declared in

[1] An Oberon module can be viewed as a container of types, procedures and variables, where variables can be allocated in the global module area (when a module is loaded), on the stack (as local variables, when a procedure is called) or on the heap (via the predefined procedure NEW, with pointer variables subsequently referring to them). Thus, in general there can be type, procedure or pointer (variable) references from static or dynamic objects of other loaded modules to static or dynamic objects of the module to be unloaded. However, only dynamic type and static and dynamic procedure references need to be checked. Static type references from other modules referring to types declared in the specified module don't need to be checked, as these are already handled via their import/export relationship (if clients exist, a module is never unloaded). Pointer references from static or dynamic pointer variables of other modules to dynamic objects of the specified module don't need to be checked, as these are handled by the garbage collector. Pointer references to static objects are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).

[2] The module M' to be unloaded may, for example, have inserted an object of an extended type T' in a data structure rooted in a base module M exporting a base type T.

[3] 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).

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

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

the unloaded module or in the form of *procedure variables* referring to procedures declared in that module[6]. A variety of approaches have been employed in different implementations of the Oberon system to cope with the introduced dangling references.

On systems that feature a memory management unit (MMU) 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. Thus, access to an unloaded module, for example via dangling procedure variables pointing to its code block, results in a trap on Ceres-1 and Ceres-2, but goes undetected on Ceres-3, which does not use virtual memory. One disadvantage of this solution is that it requires special hardware support, which may not be available on all target systems.

On systems that do *not* use virtual memory, such as Ceres-3 or RISC[7], there are several alternative ways to tackle the problem of dangling references.

First, one could *allow* module unloading only if *procedure references* exist, but not if *type references* are present[8]. To handle procedure references, one could check for them and simply set them to a *dummy* procedure, when a module is removed from memory, thereby *preventing* a run-time error when such “fixed-up” procedures are called. This approach has been implemented in an earlier version of Experimental Oberon, but was later discarded, mainly because the resulting effect on the behavior of the system would be difficult to predict or detect by the user. It would usually become “visible” only through the *absence* of an action – such as mouse tracking, if the unloaded module contained a viewer handler, for instance. An alternative approach would be to use *indirection* for *external* calls using a so-called *link table*, which would be reset to *dummy* entries upon module unloading, thereby *causing* a run-time error when such entries are invoked. Although this solution does not require reference checking for procedures (as the link table now provides the translation mechanism), it would make the system less efficient, as external procedure calls are slowed down due to the extra indirection. However, on systems that provide efficient hardware support for external procedure calls, it may be considered as a potential option[9].

Second, one could *also* allow module unloading if *type references* exist in the remaining part of the system. In that case, one must assure that their type descriptors are *preserved* in memory, even if the associated module is removed. One approach is to allocate them dynamically in the *heap* at load time[10], with the effect that they survive beyond the lifetime of their associated module[11]. This solution has been implemented in Original Oberon on Ceres-3 and 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[12]. However, only *type* references can be handled this way; procedure references would still need to be handled separately[13].

[6] *If the programming language Oberon-2 is used, there can also be references to method tables (which however are typically allocated within type descriptors).*
[7] *https://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/RISC.pdf (The Design of a RISC Architecture and its Implementation with an FPGA, 2015)*
[8] *Alternatively, for type references one could introduce the additional rule that a module can be dispensed only if does not d e c l a r e any record types, which are extensions of imported types. Since this condition is known at compile time, no run-time reference checking would be needed for type references. See chapter 6.4, page 82, of the book Project Oberon (2013 Edition) – note, however, that the described condition is not actually implemented in Oberon 2013.*
[9] *The (now defunct) NS processor used in the Ceres computer was such a system. It represented an example of a complex instruction set computer (CISC), which, among other useful features of relevance to modular programming, offered the call external procedure (CXP) instruction, which both increased code density and sped up the process of activating procedures located in other modules significantly. In later versions of the NS processor, however, this instruction has been internally re-implemented using microcode, following the general industry trend of implementing only frequent, simple instructions directly with hardware, while interpreting more complex instructions using internal microcode. This decision to rearchitect the internal structure of the NS processor rendered the original performance advantage of the CXP instruction obsolete, so that in later versions of the Oberon system (e.g., on the Ceres-3 computer, which was based on a later version of the NS processor), it was no longer used – it had in fact become slower than its replacement that does not use the CXP instruction. With the advent of highly regular reduced instruction set computers (RISCs) in the early 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.*
[10] *Note that one cannot simply copy type descriptors around in memory when a module is unloaded, because their absolute memory addresses are typically used in implementations (e.g., to implement type tests and type guards). Thus, they must be placed at a f i x e d memory address at module load time.*
[11] *They would still have to be explicitely persisted w h i l e the module is loaded, to ensure that they are not garbage collected in case there happen to be no references to them.*
[12] *See chapter 8.4, page 202, of the book Project Oberon (2005 Edition)*
[13] *Preserving procedures in the same way would make no sense, as it would essentially amount to preserving the e n t i r e module (procedures may access global module data).*

Third, one could simply *ignore* the problem of dangling references and *always* remove a module block from memory *without* taking any further precautions (other than checking whether clients exist). This is the approach chosen in Original Oberon 2013 on RISC. However, it may leave the system in an unsafe state, as modules that are loaded later can at any time overwrite the freed module block, if it fits. This is of course undesirable.

Even though most of these approaches to cope with the *dangling reference* problem (or various combinations thereof) have actually been realized in different implementations of the Oberon system, we emphasize that none of them can be considered truly satisfactory. The main issue is that the moment one *allows* modules to be physically removed from memory in spite of the presence of references to them, the resulting dangling references must be "fixed up" somehow in order to prevent an almost certain system crash. 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, because other loaded modules may *critically* depend on the removed functionality.

For example, unloading a module containing 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[14]. Even though the trap will prevent a system crash (as intended), the user still needs to reboot the system in order to recover an environment without any "frozen" parts. A similar problem may occur with references to type descriptors if they are not preserved in memory after the unloading of a module. If, for example, an object of an extended type declared in a client module M' can no longer be deleted from the data structure rooted in a base module M, because the client module M' has already been unloaded, a single such reference in the base module M (which could be module *Viewers*) may suffice to render the entire system unusable[15].

For these reasons, we don't recommend *allowing* a module to be physically removed from memory, as long as references to them still exist anywhere in the system. The approaches described above to cope with *dangling references* appear to only tinker with the symptoms of a problem that would not exist, if only one disallowed the physical removal of referenced modules. As a general rule, it's best not to introduce artificial hidden mechanisms.

[14] *The enclosing menu viewer would attempt to send a "close" message to the sub frame by calling its handler – which however causes a trap if the module has been unloaded.*
[15] *One can of course always place the 'delete' operation in the base module itself, but that doesn't necessarily have to be the case.*