The Experimental Oberon System

Andreas Pirklbauer

1.9.2017

Experimental Oberon[1] is a revision of the Original Oberon[2] operating system. It contains a number of enhancements including continuous fractional line scrolling with variable line spaces, multiple logical displays, enhanced viewer management, safe module unloading and a minimal version of the Oberon system building tools. Some of these modifications are purely of experimental nature, while others serve the explicit purpose of exploring potential future extensions, for example to add support for touch display devices.

1. Continuous fractional line scrolling with variable line spaces

Continuous fractional line scrolling has been added to the viewer system, enabling completely smooth scrolling of displayed texts with variable lines spaces and dragging of entire viewers with continuous content refresh. Both *far* (positional) scrolling and *near* (pixel-based) scrolling are realized[3]. To the purist, such a feature may represent an "unnecessary embellishment" of Oberon, but it is simply indispensable if the system is to support touch display devices where a mouse is absent and viewers may not have scrollbars. In such an environment, continuous scrolling is the only acceptable way to scroll and presents a more natural user interface. As a side effect, the initial learning curve for users new to the system is *considerably* reduced.

2. Multiple logical display areas ("virtual displays")

The Oberon system was designed to operate on a *single* abstract logical display area which is decomposed into a number of vertical *tracks*, each of which is further decomposed into a number of horizontal *viewers*. Experimental Oberon adds the ability to create *several* such display areas (or *displays* for short) on the fly and to seamlessly switch between them. Thus, the extended conceptual hierarchy of the display system consists of the triplet *(display, track, viewer)*, and consequently the underlying base module *Viewers* exports procedures to add and remove *displays*, to open and close *tracks* within displays and to open and close individual *viewers* within tracks. There are no restrictions on the number of displays, tracks or viewers that can be created. Focus viewers and text selections are separately defined for each display. This scheme naturally maps to systems with multiple *physical* monitors. It can also be used to realize fast context switching, for example in response to a swipe gesture on a touch display.

The command *System.OpenDisplay* opens a new logical display, *System.CloseDisplay* closes an existing one. *System.ShowDisplays* lists all open displays, *System.ThisDisplay* shows the display *id* and *name* of the current display, *System.SetDisplay* switches between displays, and *System.SetDisplayName* assigns a new name to an existing display.

---

[1] *http://www.github.com/andreaspirklbauer/Oberon-experimental*
[2] *http://www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html (Original Oberon, 2013 Edition); see also http://www.projectoberon.com*
[3] *The system automatically switches back and forth between the two scrolling modes based on the horizontal position of the mouse pointer.*

The command *System.Clone*, displayed in the title bar of every menu viewer, opens a new logical display on the fly *and* displays a copy of the initiating viewer *there*. The user can easily toggle between the two copies of the viewers (i.e. switch *displays*) with a single mouse click[4].

Alternatively, the user can activate the command *System.Expand*, also displayed in the title bar of every menu viewer, to expand the viewer "as much as possible" by reducing all other viewers in the track to their minimum heights, and switch back to any of the "compressed" viewers by clicking on *System.Expand* again in any of their (still visible) title bars.

3. Enhanced viewer management

The basic viewer operations *Change* (in modules *Viewers* and *MenuViewers*) and *Modify* (in modules *MenuViewers* and *TextFrames*) have been generalized to handle *arbitrary* viewer modifications, including pure vertical translations (without changing the viewer's height), adjusting the top bottom line, the bottom line and the height of a viewer with a single *viewer change* operation, and dragging multiple viewers around with a single *mouse drag* operation.

Several viewer message types (e.g., *ModifyMsg*) and message identifiers (e.g., *extend, reduce)* have been eliminated, further streamlining the overall type hierarchy. The remaining message types and identifiers now have a single, well-defined purpose. For example, restoring a viewer is accomplished exclusively by means of a *restore* message identifier.

In addition, a number of viewer message types that appeared to be generic enough to be made generally available to *all* viewer types have been merged and moved from higher-level modules to the base module *Viewers*, resulting in fewer module dependencies in the process. Most notably, module *TextFrames* no longer depends on module *MenuViewers*, making it possible to (recursively) embed text frames into *other* types of frames or composite viewers, for example a viewer consisting of an *arbitrary* number of text, graphic or picture frames.

4. Safe module unloading

The semantics of module *unloading* has been refined as follows. If clients exist, a module or module group is never unloaded. If no clients *and* no 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 references exist, the user has the *option* to *initially* remove the module or module group only from the *list* of loaded modules, without releasing its associated memory[5]. Such *hidden* modules are later *physically* removed from *memory* as soon as there are no more references to them[6]. To achieve this automatic removal of no longer referenced module data, a new command *Modules.Collect* has been added to the Oberon background task handling garbage collection[7]. It checks *all* possible combinations of module *subgroups* among the *hidden* modules for outside clients and references.

---

[4] By comparison, the Original Oberon commands System.Copy and System.Grow create a copy of the original viewer in the s a m e (and only) logical display area – System.Copy opens another viewer in the same track of the display, while System.Grow extends the viewer's copy over the entire column or display, lifting the viewer to an "overlay" in the third dimension.

[5] Removing a module from the list of loaded modules amounts to renaming it, allowing another module with the same name to be (re-)loaded again. Modules removed o n l y 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 Oberon 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). In this case, the command (with the module number used instead of the module name) can be activated from anywhere in the system, after first designating the viewer to be closed as operand by placing the "star" marker in it.

[6] Removing a module from memory frees up the memory area previously occupied by the module block. In Original Oberon 2013 on RISC and in Experimental Oberon, this includes the module's type descriptors. In some other Oberon implementations, such as Original Oberon on Ceres, type descriptors are not stored in the module block, but allocated dynamically 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, because module blocks are removed only from the l i s t of loaded modules, if they are still referenced by other modules. Thus, type descriptors can safely be stored in the (static) module blocks.

[7] The command Modules.Collect can also be manually activated at any time. Alternatively, one can invoke the command System.Collect which includes a call to Modules.Collect.

Thus, 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. For example, older versions of a module's code can still be executed if they are referenced by procedure variables in other modules, even if a newer version of the module has been reloaded in the meantime. Type descriptors also remain accessible to other modules for as long as needed[8].

If a module *group* is to be unloaded and there exist clients or references *only* within this group, the group is unloaded *as a whole*. Both *regular* and *cyclic* module imports and references *within* a module group are allowed and will *not* prevent the unloading of that group[9].

There are two *base* variants of the module *unload* command: *System.Free* and *System.Unload*. The default command *System.Free* implements the semantics described above. The command *System.Unload* differs from *System.Free* only in that it does *not* hide the modules from the list of loaded modules, if references to them exist in the remaining modules. This gives the user the *option* to *select* the desired behavior in that case.

It is also possible to release *entire subsystems* of modules. The command *System.FreeImports* attempts to unload the specified modules together with all their direct and indirect *imports*[10]. 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 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. The additional subvariants *System.FreeRemovableImports* and *System.FreeRemovableClients* unload the largest *subset* of modules that has *no* outside clients. The corresponding variations of the command *System.Unload* are analogous.

Note that these unloading strategies amount to *heuristics* and may tend to unload rather large parts of the system, which may not be desired. Therefore, the recommended way to unload modules is to use the *base* commands *System.Free* and *System.Unload* with a *specific* set of modules provided as parameters. To help identify valid module subsets that can be unloaded, the commands *System.ShowImports, System.ShowClients, System.ShowRemovableImports* and *System.ShowRemovableClients* are provided.

When the user attempts to *unload* a module or module group, clients are checked first. If clients exist among the other modules, no further action is taken. If no outside clients exist, references are checked. References that *need* to be checked prior to module unloading include *type tags* (addresses of type descriptors) in *dynamic* objects allocated in the heap reachable by all other loaded modules pointing to descriptors of types declared in the modules to be unloaded, and *procedure variables* installed in *static or dynamic* objects of other modules referring to *static* procedures declared in the modules to be unloaded[11].

---

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

[9] *Cyclic references can be created by pointers or procedure variables. Cyclic module imports are normally not allowed in Oberon. However, through a tricky sequence of compilation and editing steps, it is in fact possible to construct cyclic module imports, which cannot be detected by the compiler. But such modules are not allowed to be loaded in Experimental Oberon, as its module loader – adopting the approach chosen in Original Oberon – would simply enter an endless recursion, eventually leading to a stack overflow or out-of-memory condition – an acceptable solution for such an artificial case. But even if modules with cyclic module imports were allowed to be loaded, Experimental Oberon would be able to handle them correctly.*

[10] *The Oberon system itself (module System and all its direct and indirect imports) is of course excluded from the list of modules to be unloaded.*

[11] *An Oberon module can be viewed as a container of types, variables and procedures, where variables can be procedure-typed. Types are statically declared as global types (in which case they can be exported and therefore also be referenced by other client modules) or as types 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), or allocated in a dynamic space called the heap via the predefined procedure NEW. Thus, in general there can be type, variable or procedure references from static or dynamic (heap) objects of other modules to static or dynamic objects of the specified modules to be unloaded. However, only dynamic type and static and dynamic procedure references need to be checked during module unloading, for the following reasons. First, static type and variable references from other loaded modules can only refer by name to types or variables declared in the modules to be unloaded. Such references are 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 variable references from global or dynamic pointer variables of other modules to dynamic heap objects reachable by the modules to be unloaded don't need to be checked either, as such references are allowed to exist, i.e. don't prevent module unloading. Such references will be handled by the garbage collector during a future garbage collection cycle as follows: heap records reachable by the unloaded modules and other (still loaded) modules will not be collected, while heap records that were reachable only by the just unloaded modules will be collected – as they should. Finally, pointer variable references to static objects (declared as global variables) are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).*

To check clients and references, the module *unload* command first *selects* the modules to be unloaded using procedure *Modules.Select*, and then invokes procedure *Modules.Check*[12].

```
    PROCEDURE Check*(VAR res: INTEGER); (*whether clients or references to selected modules exist*)
 1   VAR mod, imp, m: Module; continue: BOOLEAN;
 2     pref, pvadr, r: LONGINT; p, q, impcnt, resType, resProc: INTEGER;
 3  BEGIN res := 0; mod := root;
 4   WHILE (mod # NIL) & (res = 0) DO
 5     IF (mod.name[0] # 0X) & mod.selected & (mod.refcnt > 0) THEN m := root; impcnt := 0;
 6       WHILE m # NIL DO (*count clients within selected modules*)
 7         IF (m.name[0] # 0X) & (m # mod) & m.selected THEN p := m.imp; q := m.cmd;
 8           WHILE p < q DO imp := Mem[p];
 9             IF imp = mod THEN (*m imports mod*) INC(impcnt); p := q ELSE INC(p, 4) END
10           END
11         END ;
12         m := m.next
13       END ;
14       IF mod.refcnt # impcnt THEN res := 1 END
15     END ;
16     mod := mod.next
17   END ;
18   IF res = 0 THEN mod := root;
19     WHILE mod # NIL DO (*mark dynamic records reachable by all other modules*)
20       IF (mod.name[0] # 0X) & ~mod.selected THEN Kernel.Mark(mod.ptr) END ;
21       mod := mod.next
22     END ;
23     Kernel.Scan(ChkSel, ChkSel, resType, resProc); (*check dynamic type and procedure references*)
24     IF resType > 0 THEN res := 2 ELSIF resProc > 0 THEN res := 3
25     ELSE mod := root; continue := TRUE;
26       WHILE continue & (mod # NIL) DO (*check static procedure references*)
27         IF (mod.name[0] # 0X) & ~mod.selected THEN
28           pref := mod.pvar; pvadr := Mem[pref];
29           WHILE continue & (pvadr # 0) DO r := Mem[pvadr];
30             IF ChkSel(r, continue) > 0 THEN res := 4 END ;
31             INC(pref, 4); pvadr := Mem[pref]
32           END
33         END ;
34         mod := mod.next
35       END
36     END
37   END
    END Check;
```

Clients are checked by verifying whether the number of clients of each module *within* the group of selected modules matches its *total* number of clients (lines 3-17). References from *dynamic* objects allocated in the heap are checked using a conventional *mark-scan* scheme. During the *mark* phase (lines 18-22), heap records reachable by all *named* global pointer variables of *all other* loaded modules are marked (line 20), thereby excluding records reachable *only* by the specified modules themselves. This ensures that when a module or module group is referenced *only* by itself, it can still be unloaded. The subsequent *scan phase* (line 23), implemented as a separate procedure *Scan* in module *Kernel*[13], scans the heap sequentially, unmarks all *marked* records and checks whether the *type tags* of the marked records point to descriptors of types in the *selected* modules to be unloaded, and whether *procedure variables* declared in these records refer to global procedures declared in those same modules. The latter check is then also performed for all *static* procedure variables of all other loaded modules (lines 25-35).

[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.*
[13] *The original procedure Kernel.Scan (implementing the scan phase of the Oberon garbage collector) has been renamed to Kernel.Collect, in analogy to Modules.Collect.*

In order to omit in module *Kernel* any reference to the module list rooted in module *Modules*, procedure *Kernel.Scan* is expressed as a *generic* heap scan scheme, which calls *parametric* handler procedures for individual elements of each *marked* record, instead of *directly* checking whether these records contain *type* or *procedure* references to the modules to be unloaded.

```
    PROCEDURE Scan*(type, proc: Handler; VAR resType, resProc: INTEGER);
 1   VAR p, r, mark, tag, size, offadr, offset: LONGINT; continue: BOOLEAN;
 2  BEGIN p := heapOrg; resType := 0; resProc := 0; continue := (type # NIL) OR (proc # NIL);
 3   REPEAT mark := Mem[p+4];
 4    IF mark < 0 THEN (*free*) size := Mem[p]
 5    ELSE (*allocated*) tag := Mem[p]; size := Mem[tag];
 6     IF mark > 0 THEN Mem[p+4] := 0; (*unmark*)
 7      IF continue THEN
 8       IF type # NIL THEN INC(resType, type(tag, continue)) END ; (*call type for type tag*)
 9       IF continue & (proc # NIL) THEN offadr := tag - 4; offset := Mem[offadr];
10        WHILE continue & (offset # -1) DO r := Mem[p+8+offset];
11         INC(resProc, proc(r, continue)); (*call proc for each procedure variable*)
12         DEC(offadr, 4); offset := Mem[offadr]
13        END
14       END
15      END
16     END
17    END ;
18    INC(p, size)
19   UNTIL p >= heapLim
    END Scan;
```

In this scheme, the parametric procedure *type* is called with the *type tag* of the heap record as argument (line 8), and procedure *proc* is called for each procedure variable declared within the same record with (the address of) the *procedure* itself as argument (line 11). The results of the handler calls are *separately* added up for each handler 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 7, 9, 10)[14].

Procedure *Modules.Check* uses this generic *heap scan* scheme by passing a private handler procedure *ChkSel*, which merely checks whether the argument supplied by *Kernel.Scan* (either a type tag or a procedure variable) references *any* of the modules to be unloaded[15] and *stops* checking for references as soon as the *first* such reference is found, while indicating that fact to the caller. The scan process itself continues, but only to *unmark* the remaining marked records (line 6). We note that *Modules.Check* is not only called when a module is unloaded by the *user*, but also by the Oberon *background task* that removes no longer referenced *hidden* module data from memory. Thus, it must be able to handle *both* visible and hidden modules.

In order to make the outlined validation pass possible, type descriptors of *dynamic* objects[16] in the *heap* as well as the descriptors of *global* module data in *static* module blocks have been extended with a list of *procedure variable offsets*, adopting an approach employed in one of the earlier implementations of the Oberon system (MacOberon)[17], whose run-time representation of a dynamic record and its associated type descriptor is shown in Figure 1.

---

[14] *If o n e of the two handler procedures sets continue to FALSE, procedure Kernel.Scan will stop calling b o t h of them. This somewhat artificial restriction could be lifted if needed.*
[15] *This necessitates an extra iteration over all modules. On systems that offer metaprogramming facilities such as persistent type and procedure objects collected in libraries, additional meta-information may be present in the run-time representation of modules, such as the locations of procedures and type descriptors in modules. In such a case, a simpler solution may exist.*
[16] *See chapter 8.2, page 109, of the book Project Oberon (2013 Edition) for a detailed description of an Oberon type descriptor. It contains certain information about dynamically allocated records that is shared by all allocated objects of the same record type (such as its size, information about type extensions and the offsets of all pointer fields within the record).*
[17] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

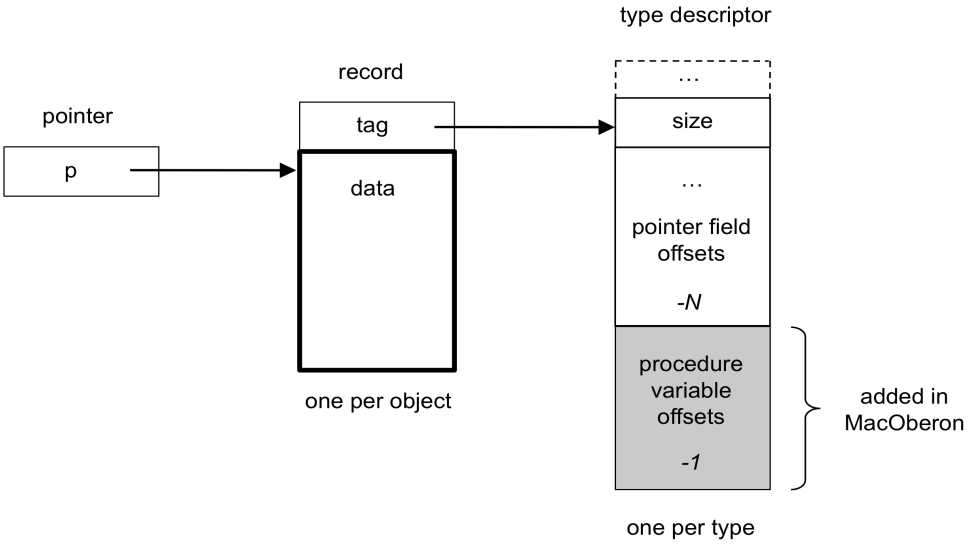

Figure 1: Run-time representation of a dynamic record and its type descriptor in MacOberon

This run-time layout would require the *scan* phase to traverse (skip over) the list of *pointer field offsets* in the type descriptor of *each* marked record. To avoid this[18], Experimental Oberon uses a slightly different run-time representation, where *procedure variable offsets* are *prepended* rather than *appended*, to the existing fields of each type descriptor, as shown in Figure 2[19].

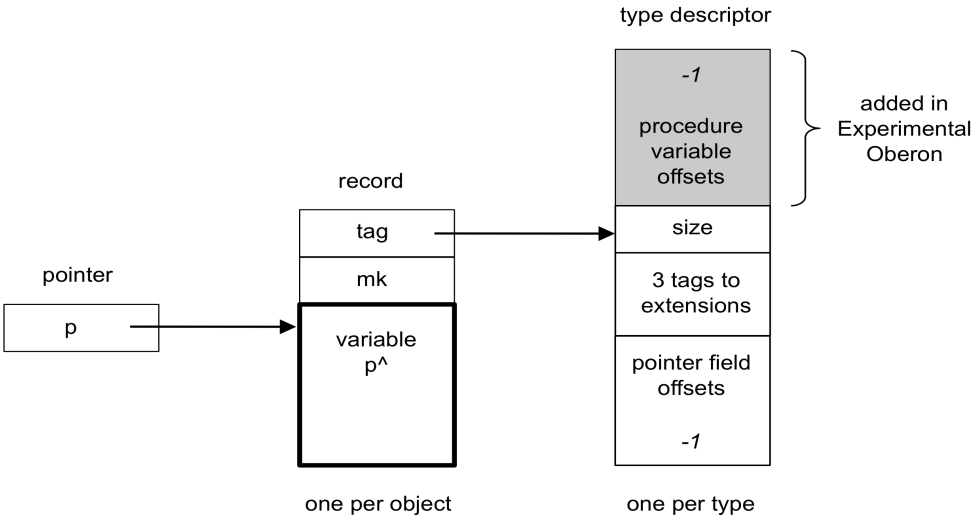

Figure 2: Run-time representation of a dynamic record and its type descriptor in Experimental Oberon

The compiler generating the modified type descriptors of dynamic objects and the descriptors of global module data, the format of the object file containing them and the module loader transferring them from object file into memory have been adjusted accordingly.

An obvious shortcoming of the reference checking scheme presented 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

[18] *The traversal could also be avoided in other ways (e.g., by storing the number of pointer field offsets at a fixed location inside the type descriptor), but we opted for a simpler solution.*
[19] *This runtime representation might come into conflict with some implementations of the Oberon-2 programming language, which typically also prepend additional run-time information to the fields in each type descriptor (namely a "method table" associated with an Oberon-2 type descriptor, which however serves a completely different purpose than the list of procedure variable offsets used for reference checking in Experimental Oberon). Thus, if Oberon-2 is to be supported in Experimental Oberon, procedure variable offsets can of course also be appended to the existing fields of each type descriptor – implementing this would require only a one-line change to the compiler (procedure ORG.BuildTD) and the scan phase of reference checking (procedure Kernel.Scan). Alternatively, one could also adapt the Oberon-2 implementation, such that the method table is allocated somewhere else (e.g., in the heap at module load time).*

module releases. However, since there exists only one type descriptor per declared record *type* rather than one per allocated heap *record*, the additional memory requirements are negligible[20]. An additional downside is that when a module cannot be removed from memory due to existing references, the user usually does not know why the removal has failed. However, since in this case the user has the option to remove the module from the module *list*, we don't consider this issue as serious. In addition, this shortcoming can easily be remedied by providing a tool command that displays the *names* of the modules containing the offending references[21].

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), even though *each* reachable heap record is traversed during the *mark* phase (as during garbage collection) and for *each* marked record *all* modules to be unloaded are checked for outside references to them during the *scan* phase. However, the number of modules unloaded with a *single* module unload operation is typically small, and reference checking *stops* when the *first* reference is detected. Furthermore, module unloading is usually rare except, perhaps, during development, where however the number of references to a module under development tends to be small. We also note that modules that manage dynamic data structures shared by client modules (such as a viewer manager) are often never unloaded at all. Thus, the presented solution, which was mainly chosen for its simplicity, appears to be amply sufficient for most practical purposes.

As an alternative to the outlined *conventional* mark-scan scheme – where the *mark* phase first marks *all* heap objects reachable by other loaded modules and the *scan* phase then checks for references from *marked* records to the modules to be unloaded – 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 several complications. First, the pointer rotation scheme employed during the *mark* phase temporarily modifies both the *mark* field and the *pointer variable* fields of the encountered heap records. As a consequence, 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 completing the *mark* phase, which in turn would undo the performance gain. Second, if one wants to omit in module *Kernel* any reference to the data structure managed by module *Modules*, one would need to express also the *mark* phase as a *generic* heap traversal scheme with parametric handler procedures for reference checking, similar to the generic heap *scan* scheme outlined above. Such a generalization would open the possibility for an erroneous handler procedure to prematurely end the *mark* phase, leaving the heap in an irreparable state. In sum, one seems well advised not to interfere with the *mark* phase. In addition, one would still need a separate *scan* phase to *unmark* the already marked portion of the heap.

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. This could be achieved by making procedures and type descriptors *look like* records, which in turn can be accomplished by making them carry a *type tag* and a *mark field*. These additional fields

---

[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, thereby making their offsets implicit. Thus, it is in principle possible to realize the reference checking scheme without additional memory requirements in type descriptors. However, we refrain 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 (procedure ORP.RecordType). 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 operations more complex. For example, assignments to subrecords would become much less natural, because their fields would no longer be located in a contiguous section in memory. As a result, we would no longer obtain a simple and uniform implementation covering both dynamic records and global module data. Second, the memory savings in the module areas holding the type descriptors would be marginal, given that there exists only one descriptor per declared record type. In addition, we believe that most applications should be programmed in the conventional programming style, where installed procedures should be rare. For example, in Oberon, the object-oriented programming style is restricted to the viewer system, which provides distributed control in the form of installed handler procedures (of which there is typically only one per type of viewer). On balance, the benefit obtained by saving a few fields in a small number of type descriptors appears negligible and thus the additional complexity required to implement this refinement would be hard to justify.

[21] In the case of dynamic records, one can display the name of the module declaring its t y p e, which is not necessarily the module rooting the data structure holding the record itself. One could, however, modify the reference checking algorithm such that each module to be unloaded is checked i n d i v i d u a l l y for dynamic references, in which case also their data structure r o o t s would be known and therefore the names of the modules d e c l a r i n g these roots (in the form of global pointer variables) could be displayed.

would be inserted as a prefix to (the code section of) *each* procedure and *each* type descriptor stored 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 field offsets. Therefore, they can be represented by one and the same global “meta descriptor”, which could be stored at a fixed location in the module area. The resulting run-time representation of *procedures* is shown in Figure 3,

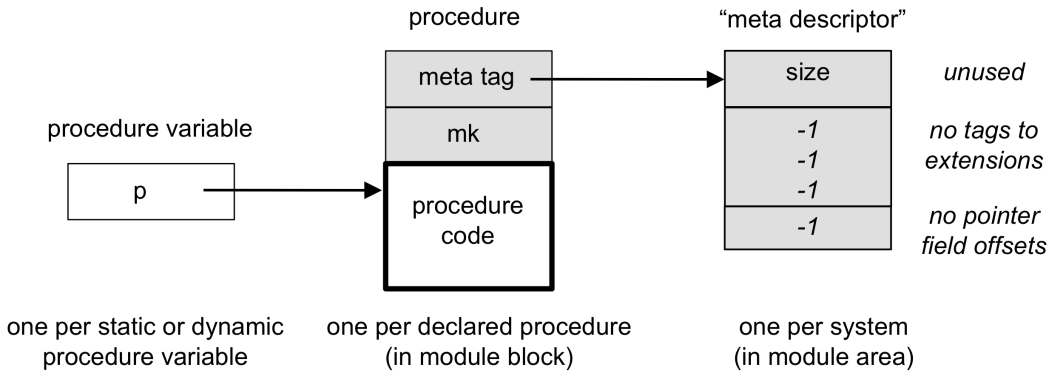

Figure 3: Procedure variable interpreted as *pointer*, and procedure interpreted as *record*

while the corresponding run-time representation of *type descriptors* is shown in Figure 4.

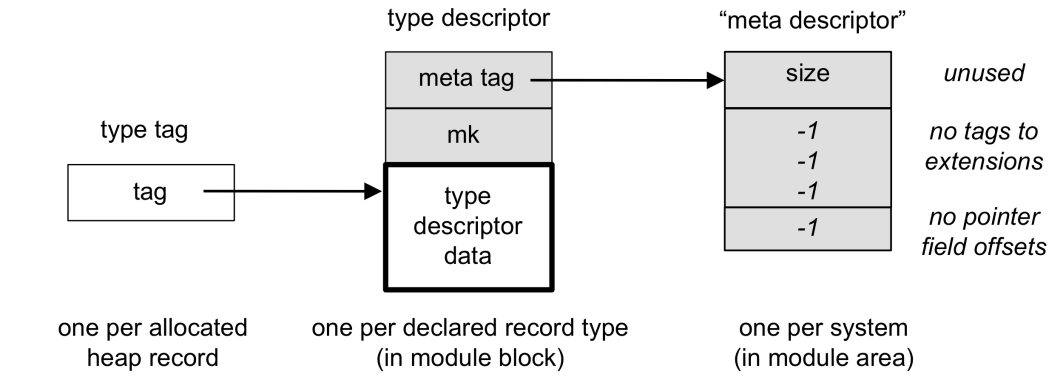

Figure 4: Type tag interpreted as *pointer*, and type descriptor interpreted as *record*

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 5.

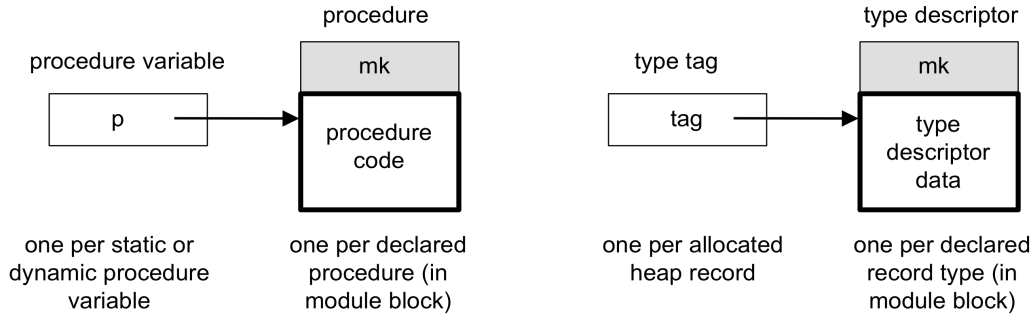

Figure 5: Procedures and type descriptors with an additional *mark* field

With these preparations, the *mark* phase of reference checking could be *extended* to include *procedure variables* and *type tags* in the list of "pointers" to be traversed for each encountered "record", which could now *also* be a declared *procedure* or a *type descriptor*. The roots used by the *mark* phase as starting points would *also* include global *procedure* variables, in addition to named global *pointer* variables. In sum, the *extended* mark phase would not only mark objects in the *heap*, but also within the *module blocks*. This simply means that after the *mark* phase *all* referenced objects are *already* marked.

While this technique appears appealing, 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* and mark *modules* 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[22]. 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. And finally, a comparison of the code required to implement the various alternatives showed that our solution is by far the simplest. The *total* implementation cost of *all* modifications to the representation of the type descriptor, the Oberon object file format and the module loader, as described earlier, is only about a dozen of lines of source code[23], while the *reference checking* phase itself amounts to less than 75 lines of code.

5. System building tools

A minimal version of the Oberon system *building tools* has been added, consisting of the two modules *Linker*[24] and *Builder*. They provide the necessary tools to establish the prerequisites for the *regular* Oberon startup process[25].

When the power to a computer is turned on or the reset button is pressed, the computer's *boot firmware* is activated. The boot firmware is a small standalone program permanently resident in the computer's read-only store, such as a read-only memory (ROM) or a field-programmable read-only memory (PROM), which is part of the computer's hardware.

In Oberon, the boot firmware is called the *boot loader*, as its primary task is to *load* a valid *boot file* (a pre-linked binary containing a set of compiled Oberon modules) from a valid *boot source* into memory and then transfer control to its *top* module (the module that directly or indirectly imports all other modules in the boot file). Then its job is done until the next time the computer is restarted or the reset button is pressed. In general, there is no need to modify the boot loader (*BootLoad.Mod*). If one really has to, one typically has to resort to proprietary tools to load the boot loader onto the specific hardware platform used.

There are currently two valid boot sources in Oberon: a local disk, realized using a Secure Digital (SD) card in Oberon 2013, and a communication channel, realized using an RS-232 serial line. The *default* boot source is the local disk. It is used by the *regular* Oberon startup process each time the computer is powered on or the reset button is pressed.

---

[22] *On systems where additional meta-information is present in the run-time representation of modules, such as the locations of procedures and type descriptors in module blocks and/or backpointers from each object of a module to its module descriptor, the mark phase could be made simpler. However, Experimental Oberon does not offer such metaprogramming facilities.*
[23] *See procedures ORG.BuildTD (+ 1 line), ORG.Close (+ 7 lines) and Modules.Load (+ 4 lines).*
[24] *The linker has been included from a different source (https://github.com/charlesap/io). It was adapted for Experimental Oberon's object file format (which includes a modified module descriptor, modified type descriptors of dynamically allocated records and an additional section in the module block holding global procedure variable references).*
[25] *Currently not implemented is a tool to prepare a disk initially – which consists of a single 'Kernel.PutSector' statement that initializes the root page of the file directory (sector 1).*

The command *Linker.Link* links a set of Oberon object files together and generates a valid boot file from them. The linker is almost identical to the regular module loader (*Modules.Load),* except that it writes the result to a file instead of loading and linking the modules in memory.

The command *Builder.Load* loads a valid boot file, as generated by *Linker.Link*, onto the *boot area* (sectors 2-63 in Oberon 2013) of the local disk, one of the two valid boot sources.

The format of the boot file is *defined* to exactly mirror the standard Oberon storage layout[26]. In particular, location 0 in the boot file (and later in memory once it has been loaded by the boot loader) contains a branch instruction to the initialization sequence of the *top* module of the boot file. Thus, the boot loader can simply transfer the boot file byte for byte from a valid boot source into memory and then branch to location 0 – which is precisely what it does[27].

In sum, to generate a new regular Oberon boot file and load it onto the local disk's boot area, one can execute the following commands *on* the system that is to be modified:

```
ORP.Compile Kernel.Mod FileDir.Mod Files.Mod Modules.Mod ~   ... compile the modules of the inner core
Linker.Link Modules ~                                       ... create a regular boot file (Modules.bin)
Builder.Load Modules ~                                      ... load boot file onto the disk's boot area
```

Note that the last command overwrites the disk's boot area of the *running* system. A backup of the disk is therefore recommended before experimenting with new *inner cores*[28].

When *adding* modules to a boot file, the need to call their initialization bodies during stage 1 of the boot process may arise, i.e. when the boot file is loaded into memory by the boot loader during system restart or reset. We recall that the boot loader merely *transfers* the boot file byte for byte from a valid boot source into memory, but does not call the module initialization sequences of the just transferred modules (this is, in fact, why the *inner core* modules *Kernel*, *FileDir* and *Files* don't *have* module initialization bodies – they wouldn't be executed anyway).

The easiest way to add a new module *with* a module initialization body to a boot file is to move its initialization code to an exported procedure *Init* and call it from the top module of the modules in the boot file. This is the approach chosen in Original Oberon, which uses module *Modules* as the top module of the *inner core*. An alternative solution is to extract the starting addresses of the initialization bodies of the just loaded modules from their module descriptors in memory and simply call them, as shown in procedure *InitMod*[29] below (see chapter 6 of the book *Project Oberon* for a detailed description of the format of an Oberon *module descriptor* in memory; here it suffices to know that it contains a pointer to a list of *entries* for exported entities, the first one of which points to the initialization code of the module itself).

```
    PROCEDURE InitMod(name: ARRAY OF CHAR);
1     VAR mod: Modules.Module; body: Modules.Command; w: INTEGER;
2   BEGIN mod := Modules.root;
3     WHILE (mod # NIL) & (name # mod.name) DO mod := mod.next END ;
4     IF mod # NIL THEN SYSTEM.GET(mod.ent, w);
5       body := SYSTEM.VAL(Modules.Command, mod.code + w); body
6     END
    END InitMod;
```

[26] *See chapter 8.1, page 103, of the book Project Oberon (2013 Edition) for a detailed description of Oberon's storage layout.*
[27] *After transferring the boot file, the boot loader also deposits some additional key data in fixed memory locations, to allow proper continuation of the boot process.*
[28] *When using an Oberon emulator (https://github.com/pdewacht/oberon-risc-emu) on a host system, one can simply make a copy of the directory containing the disk image.*
[29] *Procedure InitMod could be placed in modules Oberon or Modules (note: the data structure rooted in the global variable Modules.root is transferred as part of the boot file).*

In the following example, module *Oberon* is chosen as the new top module of the *inner core*, while module *System* is configured to be the new top module of the *outer core*.

```
    MODULE Modules;                          ... old top module of the inner core, now just a regular module
 1   IMPORT SYSTEM, Files;
 2   ...
 3  BEGIN Init                                ... no longer loads module Oberon (as in Original Oberon)
    END Modules.

    MODULE Oberon;                           ... new top module of the inner core, now part of the boot file
 1   IMPORT SYSTEM, Kernel, Files, Modules,
 2     Input, Display, Viewers, Fonts, Texts;
 3   …
 4  BEGIN                                     ... boot loader will branch to here after transferring the boot file
 5   InitMod(„Modules“);                      ... must be called first (establishes a working file system)
 6   InitMod(„Input“);
 7   InitMod(„Display“);
 8   InitMod(„Viewers“);
 9   InitMod(„Fonts“);
10   InitMod(„Texts“);
11   …
12   Modules.Load(„System“, Mod);            ... load the outer core using the regular Oberon loader
13   Loop                                    ... transfer control to the Oberon central loop
    END Oberon.
```

The following commands build a modified *inner core* for this new module configuration and load it onto the local disk's *boot area*:

```
ORP.Compile Kernel.Mod FileDir.Mod Files.Mod Modules.Mod ~
ORP.Compile Input.Mod Display.Mod Viewers.Mod ~
ORP.Compile Fonts.Mod Texts.Mod Oberon.Mod ~     ... compile the modules of the modified inner core
Linker.Link Oberon ~                             ... create a new regular boot file (Oberon.bin)
Builder.Load Oberon ~                            ... load the boot file onto the local disk's boot area
```

This module configuration reduces the number of stages in the regular Oberon *boot process* from 3 to 2, thereby streamlining it somewhat, at the expense of extending the *inner core*. If one prefers to keep the *inner core* minimal (as one should), one could also choose to extend the *outer core* instead, for example by including module *System* and all its imports. This in turn would have the disadvantage that the viewer complex and the system tools are "locked" into the *outer core*. However, an Oberon system without a viewer manager hardly makes sense, even in closed server environments. As an advantage of the latter approach, we note that such an extended *outer core* can more easily be replaced on the fly (by unloading and reloading a suitable *subset* of modules), as module *Oberon* no longer invokes the module loader itself.

* * *