Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Robert Griesemer

On the Linearization of Graphs and Writing Symbol Files

Cuno Pfister (ed.)
Beat Heeb
Josef Templ

Oberon Technical Notes

March 1991

156

Address of the authors:

Computersysteme
ETH–Zentrum
CH–8092 Zurich, Switzerland

e–mail: griesemer@inf.ethz.ch
pfister@inf.ethz.ch

On the Linearization of Graphs and Writing Symbol Files

Robert Griesemer

Abstract

The linearization of general graphs represented by pointer and record data structures is a problem often arising in computer programs. Whenever a graph has to be stored on or transmitted over a sequentially organized carrier, a form of linearization is used. A simple algorithm for this purpose is presented and a special application – the writing of symbol files as required by modern language compilers – is described in more detail.

Keywords: *Linearization, Graphs, Symbol Files.*

1. Linearization of Graphs

General graphs occur in various forms in computer science, sophisticated data structures may often be interpreted as graphs. Whenever such a data structure has to be stored on a file or has to be transmitted over a network, the linearization problem occurs. For special forms of non–linear data structures (e.g. trees) well–known solutions exist and are comprehensively described in the basic literature. Nevertheless, for more general data structures the wheel has to be reinvented and literature is difficult to find [ReMö86]. The problem has a twofold nature: firstly, the graph has to be linearized by means of a *write algorithm*, and secondly, it has to be rebuilt out of its linear description by a *read algorithm*. In the following, *linearization* means writing a graph to a file.

1.1 Preconditions

First of all, we remember that every general (undirected) graph may be represented by a *directed graph*, by means of having two directed edges instead of an undirected one. Hence, we concentrate on directed graphs only. Secondly, we consider only *rooted* and *connected* graphs; i.e. graphs with a special *root* node and a path from the root to every other node. Unconnected graphs or graphs with several root nodes are easily extended such, that they obey the above preconditions. Thirdly, as an (academic) restriction only finite graphs are considered.

In computer programs, graphs may be represented in various forms, depending on the available notational support (the programming language) and the way the data structure is used (the program). Here we consider only graphs described in terms of pointers and records; i.e. every graph node is represented by a record and every edge is represented by a pointer. Note that possible information attached to the edges of a graph (*labeled edges*) may be interpreted as additional node data.

The nodes of such a graph may not all have the same type; i.e. the amount of data in every node may be different and the number of directed edges to other nodes may vary. It depends on the application and available language how different nodes are specified. For the sake of simplicity we identify each node with a positive *tag*–number (> 0), so that every node type is clearly determined by its tag and vice versa. Then, every node contains a certain amount *M(tag)* of data and (at most) a certain number *N(tag)* of directed edges to other nodes. Hence, a general graph node and its edges may be described in the following way (written in Oberon [Wi88]):

```
TYPE
    Node = POINTER TO NodeDesc;
    NodeDesc = RECORD
        tag: INTEGER;                       (* determines node type; tag > 0 *)
        data: ARRAY Mtag OF INTEGER;        (* M depends on the node type (tag) *)
        link: ARRAY Ntag OF Node            (* N depends on the node type (tag) *)
    END;
```

Note that any data structure represented in any way in computer memory may be written to a file by simply writing out sequentially the corresponding memory area. Reading requires "only" the data to be read into the same memory area (otherwise pointers would be incorrect). But normally this is an inappropriate solution for several reasons: often the data structure is distributed over the entire available memory and writing would lead to an immense waste of space. For reading, the destination memory addresses must be specified which is almost never possible and a file created this way is inherently not portable (to another computer system).

Hence, writing of graphs requires pointers to be transformed into another form of reference and vice versa for reading. In addition, reading and writing should be as efficient as possible considering both memory and time; e.g. each node description should appear once and only once on the file.

1.2 The Write Algorithm

The write algorithm resembles closely a naive recursive mark–algorithm for garbage collectors, actually it is nearly the same task to be done: all nodes have to be traversed and marked; marking is necessary to avoid a second traversal and to refer to the node's first occurence. While traversing the graph its structure is written to a file. Hence, the node data structure has to be extended by a *mark* field, which initially must be zero:

```
TYPE
    Node = POINTER TO NodeDesc;
    NodeDesc = RECORD
        mark: INTEGER;                      (* used for linearization; initially = 0 *)
        tag: INTEGER;                       (* determines node type; tag > 0 *)
        data: ARRAY Mtag OF INTEGER;        (* N depends on the node type (tag) *)
        link: ARRAY Ntag OF Node            (* M depends on the node type (tag) *)
    END;
```

We say a node q is *marked* iff q.mark > 0. The *WriteNode* procedure (see below) traverses the graph in *pre–order*, starting with the root node. Whenever an unmarked node is encountered, the node is numbered and thus marked (line 4), the node tag and data are written (lines 4 and 5) and its successor nodes are traversed (line 6). Note that a node must be numbered before its subtrees are traversed because they may refer to it. Numbering is done using the counter *NofNodes* which is one initially and is then incremented by one with every processed node. Hence, NofNodes is always greater than zero. Whenever an already marked node is traversed, only its negative node number is written out as reference number (line 2). If the node doesn't exist (i.e. q = NIL) zero is written out (line 1):

```
    VAR
       NofNodes: INTEGER;
 (* node number; initially = 1 *)

    PROCEDURE WriteNode(q: Node);
       VAR i: INTEGER;
    BEGIN
1      IF q = NIL THEN WriteInt(0)
2      ELSIF q.mark > 0 THEN (* already marked *) WriteInt(–q.mark)
3      ELSE (* first occurence *)
4         WriteInt(q.tag); q.mark := NofNodes; INC(NofNodes);
5         i := 0; WHILE i < Mtag DO WriteInt(q.data[i]); INC(i) END;
6         i := 0; WHILE i < Ntag DO WriteNode(q.link[i]); INC(i) END
7      END
    END WriteNode;
```

During execution of WriteNode with argument *root* all nodes are numbered in the order of their first occurence. References to already written nodes are the negative node numbers. While reading the file this information will be used to reconstruct the graph (see 1.3 The Read Algorithm). Before WriteNode can be called a second time for the same graph or parts of it, the preconditions have to be established again, i.e. the nodes must be unmarked. When using time–stamps, no such unmark pass is necessary (due to an idea of J. Templ, see [PfiHeTe91]: *A Symmetric Solution to the Load/Store Problem*). The WriteNode procedure leads to the following file structure (in EBNF, terminal symbols are described in double quotes):

```
Graph      =  NodeRef | NoNode | NodeDesc.
NodeRef    =  "negative node number (< 0)".
NoNode     =  "0".
NodeDesc   =  "node tag (> 0)" "node data" {Graph}.
```

It is obvious that the algorithm terminates for all graphs: there is only a finite number of nodes and every procedure call works on at least one node. The recursion is stopped when an already marked node and hence a loop is found (line 2) and the algorithm returns to its predecessor node. In line 4 the assignment q.mark := NofNodes is executed only if q.mark = 0, i.e. each node is numbered at most once (because NofNodes > 0). On the other hand, since after numbering of the node q^ (line 4) WriteNode numbers all subgraphs of q^ (line 6), each node is numbered at least once.

In combination we may conclude that each node is marked exactly once. Finally, because each node's data is written iff the node was numbered immediatly before (line 5), it becomes clear that each node is written only once.

Note that WriteNode is a *one–pass algorithm* and that no additional data structures except NofArgs and the procedure activation stack are required. The run time complexity is O(n) where n is the number of edges in the graph. As we mentioned above, WriteNode resembles closely the mark phase of a mark–and–sweap garbage collector, therefore it is possible to transform the algorithm into a completely iterative form (e.g. if stack space is critical). The analoguous transformations for a simple garbage collector are described in [PfiHeTe91].

1.3 The Read Algorithm

The read algorithm reconstructs a (sub–) tree from its description on a file. Each node block in the file starts with an identification tag. According to the EBNF file structure there are three cases to distinguish: the node tag may be zero, negative or positive (excluding zero).

```
    VAR
       NofNodes: INTEGER;
(* node number; initially = 1 *)
       NodeTab: ARRAY MaxNodes OF Node;   (* node table; NodeTab[0] = NIL *)

    PROCEDURE ReadNode(VAR q: Node);
       VAR tag, i: INTEGER;
    BEGIN
1      ReadInt(tag);
2      IF tag <= 0 THEN (* node reference *) q := NodeTab[–tag]
3      ELSE (* first occurence *)
4         NEW(q, tag); NodeTab[NofNodes] := q; INC(NofNodes);
5         i := 0; WHILE i < M_tag DO ReadInt(q.data[i]); INC(i) END;
6         i := 0; WHILE i < N_tag DO ReadNode(q.link[i]); INC(i) END
7      END
    END ReadNode;
```

The first and second case are handled together by initializing the (otherwise unused) NodeTab entry zero to NIL. Because an empty subtree is identified by a zero tag, q then automatically becomes NIL (line 2). A node that occurs for the first time is identified by a positive number which is also its node tag. Then, such a node is created using NEW and stored in a global node table NodeTab (line 4). Note again that this assignment must be done before any subtrees are read (they may potentially refer to it). After creation of the new node its data and its subtrees are read (line 5 and 6). A negative node tag refers to an already read node, with the node number –tag. This node is obtained from the NodeTab array (line 2).

It should be clear that this algorithm rebuilds the original graph, as each line in ReadNode has its counterpart in WriteNode and each call of WriteNode during writing implies a corresponding call of ReadNode during reading. Clearly this algorithm is also O(n) where n is the number of edges in the graph. A temporary node array of size n is used for reading, this might be a drawback in case of very large graphs. A 2–pass algorithm which needs temporary storage only for nodes which are referenced more than once is described in the appendix.

2. Writing Symbol Files

Today's modular programming languages like Modula–2, Ada, Oberon and others allow an application to be programmed in several more or less independent parts, so–called *modules* (or packages). Such a module typically defines data structures and provides operations (procedures) on them, these *exported objects* may be used (*imported*) by other modules without knowledge of their implementation.

Therefore, the information about a module's interface has to be available to the compiler in an efficient way: in Modula–2 and Oberon the necessary data is recorded on so–called *symbol file*s. For historical reasons and to avoid confusion we use the term *symbol file* for any linear data structure which describes a module's interface. Actually, it is not necessary that symbol files are represented by a separate file (see 2.4 Implementation Aspects).

During compilation, the information about a module's interface is held in the *symbol table* of the compiler. The symbol table can be regarded as a graph, hence writing a symbol file requires linearizing the necessary partial graph of this table. The methods described in the following have been implemented in a compiler for an experimental Oberon–like language [Gri90]. The code fragments detail only the principal structure, a complete implementation may be found in the appendix.

2.1 Structure of Symbol Tables

The symbol table of a compiler for an Oberon–like language describes the compiled *objects*, which are constants, types, variables, procedures, or modules. The table itself is built when processing *declarations*. These may be nested, so the compiler has to manage a stack of *scopes*, i.e. visibility ranges of objects. For our purposes, it is sufficient to store the objects of a scope in a linear list. More sophisticated implementations would use structures like binary trees, because objects have to be searched efficiently during compilation. In Modula–2 and Oberon only *global* objects may be exported, i.e. only objects within the global scope of the symbol table have to be considered for export (Fig. 1). Hence, writing the symbol file means linearizing the partial graph described by the exported (and therefore in some way marked) objects of this scope.

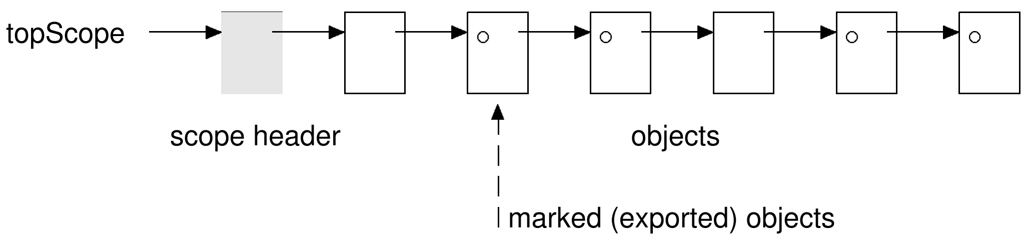


Fig. 1 Global Scope

Each node represents an object of the compiled program, containing all the necessary information about it, such as its name, the object kind, possibly an address and its type. Object types itself may have a very complex structure and are further described using a *Struct* data structure, which again may refer to other Struct nodes [Wi85, Cre90]. Hence objects and types are described with two different record structures:

```
CONST
(* object modes *)
    Undef* = 0; Scope = 1; Const* = 2; Type* = 3; Var* = 4; VarPar* = 5; XVar* = 6;
    IVar* = 7; Field* = 8; LProc* = 9; XProc* = 10; IProc* = 11; SProc* = 12; Mod* = 13;

(* type forms *)
    Char* = 2; Bool* = 3; SInt* = 4; Int* = 5; LInt* = 6; Set* = 7; Real* = 8;
    LReal* = 9; Cmplx* = 10; LCmplx* = 11; NilTyp* = 13; NoTyp* = 14;
    Proc* = 15; String* = 16; Array* = 17; DynArr* = 18; Record* = 19; Pointer* = 20;

TYPE
    Name* = ARRAY 32 OF CHAR;
    Object* = POINTER TO ObjectDesc;
    Struct* = POINTER TO StructDesc;

    ObjectDesc* = RECORD
        link*, next*: Object;           (* objects are chained using next *)
        name*: Name;
        typ*: Struct;
        marked*: BOOLEAN;               (* marked objects are exported *)
        mode*: INTEGER;
(* identifies object kind *)
        mnolev*: INTEGER;               (* module numbers are <= 0 *)
        ...                             (* object specific data *)
    END;

    StructDesc* = RECORD
        ref: INTEGER;                   (* used as mark field *)
        form*: INTEGER;                 (* identifies structure kind *)
        obj*: Object;                   (* points to type object if it exists *)
        len*, size*: LONGINT;
        base*: Struct;                  (* result–, element–, base– or pointee type
*)
        link*: Object       (* record scope *)
    END;
```

Struct nodes describe the type of an object (array, record, pointer, procedure); a few types are predefined (e.g. Char, Integer, Real). Because types may be recursively defined, the resulting data structure may contain cycles. Many objects may have the same type and therefore refer to the same struct node.

2.2 Export

The graph linearization algorithm in the form described in section 1 actually works for different node kinds (determined by the tag field) but requires that all nodes are described using the same record. However, enforcing this precondition would have a far too strong impact on the structure of a compiler. As we have actually two different record types for object and type description, the adequate solution is to have two sets of read and write procedures, one set for objects and one for types. This distinction is even more justified by the fact that objects are stored in a simple linear list and are referenced only once (with the exception of type and module objects) while type descriptors may be referenced several times and potentially belong to cycles. References from struct nodes to their type objects are handled directly in the WriteStruct procedure. Modules are never marked for export and written using a special procedure. As a consequence, the write procedure which traverses the object scope does not have to take care of objects already traversed and therefore no marking is necessary. Hence, the write procedure degenerates to a simple list traversal while for the corresponding read procedure no temporary array is necessary. Using the above definitions, the WriteObjects procedure can be written as follows (primitive operations like WriteInt and WriteName may be found in the appendix section):

```
PROCEDURE WriteObjects(obj: Object);
BEGIN
   WHILE obj # NIL DO
      IF obj.marked THEN
         WriteInt(obj.mode);
         IF (obj.mode # Type) OR (obj.typ.obj # obj) THEN (* no–type or alias type *)
            WriteName(obj.name)
         ELSE (* other type *) Write(0X)
         END;
         WriteStruct(obj.typ);
         IF obj.mode = Const THEN (* write const value *)
         ELSIF obj.mode = LProc THEN (* write parameter list *)
            ...
         END
      END;
      obj := obj.next
   END;
   WriteInt(Undef) (* termination tag *)
END WriteObjects;
```

For exported objects, the mode which describes the object kind, the object's name and its type are written. As an optimization, the name of types is only written if it concerns *alias types*. Otherwise, the name will be written by the WriteStruct procedure (given below). Then, depending on the object mode, additional information is written (e.g. the value of constants or the parameter list of procedures).

The WriteStruct procedure closely resembles the Write algorithm described in 1.2 but is slightly complicated by the fact that *named types* (i.e. types for which a type object exists) have to be handled correctly. Remember, that it is not correct to simply call the WriteObjects procedure, because WriteObjects is not built to handle more than one reference to an object. In addition, named types (and only these!) may be exported and imported over many modules and it must always be guaranteed that a type, whether it was imported via several modules (*indirect import*) or not, is described by one and only one struct node. A unique identification is necessary, which is the type name combined with the description of the module where the type was defined first. An analogous situation occurs when a type gets an alias name: then, the struct node always points to the first type object which defined the type.

```
    VAR
        nofStructs: INTEGER;
(* structure number; initially = 1 *)

    PROCEDURE WriteStruct(typ: Struct);
        VAR name: Name;
    BEGIN
        IF typ = NIL THEN WriteInt(0)
        ELSIF typ.ref > 0 THEN (* already marked *) WriteInt(–typ.ref)
        ELSE (* first occurence *)
            WriteInt(typ.form); typ.ref := nofStructs; INC(nofStructs);
            IF typ.obj # NIL THEN (* named type *)
                name := typ.obj.name;
                IF ~typ.obj.marked THEN (* invisible type *)
                    name[0] := CHR(ORD(name[0]) – ORD("@"))
                END;
                WriteName(name); WriteMod(GMod[–typ.obj.mnolev])
            ELSE Write(0X)
            END;
            CASE typ.form OF
                | Proc: (* write parameter list and result type *)
                | Array: (* write element type and length *)
                ...
            END
        END
    END WriteStruct;
```

Note that for exported named types there is a distinction between those with and those without corresponding *exported type objects*, where the latter are called *invisible types*. Invisible types must not be visible in an importing module, i.e. a programmer is not allowed to use them by name within a declaration. On the other hand they have to be visible to the compiler because it must be ensured that all invisible types with the same name are mapped onto a common struct node. Hence, the same information as for named types is written, but a simple trick inhibits any use of the type name within a program: the first letter of its name (which is always greater or equal than "A") is modified such that the name becomes a syntactically invalid identifier.

For named types the declaring module has to be known. Because several types may be exported by the same module, the corresponding module object may be referenced more than once. A special procedure which handles modules correctly is used:

```
    VAR
        nofLMods: INTEGER;
(* local module number; initially = 0 *)

    PROCEDURE WriteMod(mod: Object);
    BEGIN
        IF mod.ref < 0 THEN (* first occurence *)
```

```
            mod.ref := nofLMods; INC(nofLMods);
            WriteInt(Mod); WriteKey(mod.key); WriteName(mod.name)
        ELSE WriteInt(–mod.ref)
        END
    END WriteMod;
```

Like struct nodes, module descriptors are only written at their very first occurence. Whenever the same module is referenced later, only its reference number is written. Module pointers are never NIL, so numbering starts with zero and a module is marked iff mod.ref >= 0. During import the compiler has to check that only one version of a module is used globally, therefore every module needs a unique key. Remember that modules are never written out accidentally by the WriteObjects procedure, because they are never marked for export.

Writing the complete symbol file requires writing out the module descriptor of the compiled module (by means of WriteMod) followed by writing out all its exported objects (by means of WriteObjects).

2.3 Import

Like the general ReadNode procedure is mirroring the structure of the WriteNode procedure, the import procedures are similar to the export procedures. This fact allows for easy extension, because additionally written data in an export procedure automatically leads to the corresponding read operations in the import procedure. Nevertheless, a few difficulties need to be mastered anyway.

In several situations an object may be imported more than once: the trivial case occurs when the same module is imported twice because of a substitution (or alias) name. One might expect that this special case should be handled by simply ignoring a second import; but actually a multiple import of objects has to be handled anyway, hence double import of entire modules is only a special case of a more general situation. Multiple import of an object normally occurs for type objects, when beeing imported indirectly across different modules. Consider the following situation: a module A exports a type (object) T which is used in a variable V of a module B. Module C which imports A and B consequentially also imports the type T from A and the variable V from B and hence once again the type T as type of V (Fig. 2).

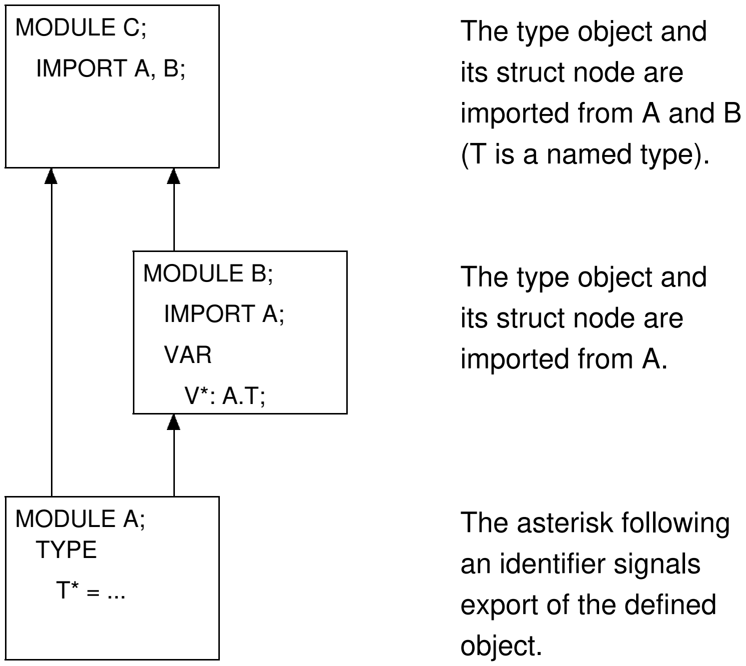


Fig. 2 Double import

However, multiply loading objects or structures, if not detected, could lead to incorrect incompatibilities during type checking. Therefore, each object which is read from the symbol file is discarded if it was already present in the symbol table. Hence, each object is represented by its very first loaded instance which is called *primary instance* [Gu85]. In consequence, the ReadObjects procedure reads an object from the symbol file but the procedure *InsertImport* inserts it in the corresponding module scope only if the object (with the same name) has not already been imported. As an optimization, note that only *alias types* have to be (additionally) inserted with InsertImport because their names differ from their type object names. All other types are handled in the ReadStruct procedure.

```
PROCEDURE InsertImport(VAR obj: Object; scope: Object);
    VAR p, q: Object;
BEGIN
    p := scope; q := scope.next;
    WHILE q # NIL DO
        IF q.name = obj.name THEN obj := q; RETURN END;
        p := q; q := q.next
    END;
    obj.mnolev := scope.mnolev; p.next := obj
END InsertImport;

PROCEDURE ReadObjects(scope: Object);
    VAR mode: LONGINT; obj: Object; typ: Struct; name: Name;
BEGIN
    LOOP (* read all objects *)
        ReadInt(mode);
        IF mode = Undef THEN (* no more objects *) EXIT
```

```
            ELSIF mode = Type THEN
                ReadName(name);
                IF name # "" THEN (* alias type *)
                    NewObject(obj, name, Type); ReadStruct(obj.typ);
                    InsertImport(obj, scope)
                ELSE (* other types *) ReadStruct(typ)
                END
            ELSE
                ReadName(name); NewObject(obj, name, SHORT(mode));
    ReadStruct(obj.typ);
                IF mode = Const THEN (* read const value *)
                ELSIF obj.mode = LProc THEN (* read parameter list *)
                    ...
                END;
                InsertImport(obj, scope)
            END
        END
    END ReadObjects;
```

The procedure ReadStruct decides upon reading the first number *form* whether the structure was already read from the (same) symbol file or not. In the first case, the structure already built is taken out of a *local structure table* LStruct which serves as a translation table for structure references analogous to NodeTab in the ReadNode procedure. In the second case, the structure is created and inserted in the LStruct table, then the specific structure information is read. As described in ReadObjects, named types must also be ignored if occuring a second time: the structure information is read but then discarded (by means of InsertImport) and the primary instance is used (and also inserted in the LStruct table).

As a tricky point, notice further: Because each type (named or not) may refer to itself, it is absolutely necessary that the primary instance is found *before* any other types are read (which potentially refer to the same type and therefore would obtain the wrong structure out of the LStruct table). Hence, no other types must occur between the type tag of a named type and its identification, i.e. its name and its original module description. This rule corresponds to postulate 5 in [Gu85].

```
    VAR
        nofStructs: INTEGER;
 (* structure number; initially = 1 *)
        LStruct: ARRAY MaxNofStructs OF Struct;
 (* local structure table; LStruct[0] = NIL *)

    PROCEDURE ReadStruct(VAR typ: Struct);
        VAR form: LONGINT; htyp: Struct; name: Name; obj, mod: Object;
    BEGIN
        ReadInt(form);
        IF form <= 0 THEN (* struct reference or NIL *) typ := LStruct[–form]
        ELSE (* first occurence *)
            NewStruct(htyp, SHORT(form)); ReadName(name);
            IF name # "" THEN (* named type *)
                NewObject(obj, name, Type); obj.marked := TRUE; obj.typ := htyp; htyp.obj :=
 obj;
```

```
            ReadMod(mod); InsertImport(obj, mod.link);
            typ := obj.typ
        ELSE typ := htyp
        END;
        LStruct[nofStructs] := typ; INC(nofStructs);
        CASE form OF
            | Proc: (* read parameter list and result type *)
            | Array: (* read element type and length *)
            ...
        END
    END
END ReadStruct;
```

At the end, the ReadMod procedure is presented. As expected, a *local module table* LMod is used as translation table for module references. In addition, because modules must always be represented by their primary instances, a *global module table* GMod is necessary, which is accessed in the InsertMod procedure.

```
VAR
    nofGMods*: INTEGER;                   (* global module number; initially = 0 *)
    GMod*: ARRAY MaxNofGMods OF Object;
(* global module table *)

PROCEDURE InsertMod*(VAR mod: Object; VAR name: ARRAY OF CHAR; key:
LONGINT);
    VAR i: INTEGER;
BEGIN i := 0;
    WHILE (i < nofGMods) & (name # GMod[i].name) DO INC(i) END;
    IF i < nofGMods THEN  (* module already imported *)
        mod := GMod[i];
        IF mod.key # key THEN err(150) (* key inconsistency *) END
    ELSE
        NewObject(mod, name, Mod); (* must not be visible in global scope *)
        mod.key := key; mod.mnolev := –nofGMods;
        OpenScope(mod.link, mod.mnolev); CloseScope; (* allocate own scope *)
        GMod[nofGMods] := mod; INC(nofGMods)
    END
END InsertMod;

VAR
    nofLMods: INTEGER;
(* local module number; initially = 0 *)
    LMod: ARRAY MaxNofLMods OF Object; (* local module table *)

PROCEDURE ReadMod(VAR mod: Object);
    VAR ref, key: LONGINT; name: Name;
BEGIN
    ReadInt(ref);
    IF ref > 0 THEN (* first occurence *)
        ReadKey(key); ReadName(name);
        InsertMod(mod, name, key);
        LMod[nofLMods] := mod; INC(nofLMods)
```

```
    ELSE mod := LMod[–ref]
    END
  END ReadMod;
```

Importing a whole module requires reading the module (by means of ReadMod) followed by reading all exported objects of this module (by means of ReadObjects). Together there are only a few additional rules to the general graph algorithm to be observed:

1. For *types* and *modules*, which both may be referenced several times within the *same symbol file*, a marking scheme and a translation table analogous to the general graph read/write algorithms is used.

2. For *named types* and *modules*, which both may occur in *several symbol files*, the *primary instance* always must be used and inserted into the local translation tables. If the primary instance already exists, the remaining data must be read but then discarded. Repeated occurence of the same module is detected using a global module table, repeated occurence of the same named type is detected using its module and the corresponding module scope.

3. As a consequence of the second rule, no other types must occur in the symbol file before name and module of a named type are specified.

2.4 Implementation Aspects

Symbol File Representation

As mentioned in the beginning, symbol files need not to be represented by a separate file. Actually, a symbol file independent of the corresponding object file mirrors the fact that in Modula–2 and other languages the *definition* and the *implementation* of a module were compiled separately: the definition module was compiled into a symbol file and the implementation module into an object file. If exported objects are marked in some way within the implementation module (as in Oberon), a definition module and hence a separate symbol file is not necessary. It is more adequate to produce only a single file during compilation, which contains all the necessary information about the module including its interface description. For example, the (conventional) symbol file may be appended to the object file. The advantages are obvious: only a single file has to be distributed, this file is always self–contained and therefore consistent and it is possible to directly generate the interface specification out of an object file with a *Browser* tool. Should it be necessary to inhibit public use of a module (e.g. because it supports low–level features), it is easy to remove the symbol file from the object file and distribute the interface–less object file only.

Canonical Form for Symbol Files

When a module has to be recompiled for some reason without a change in its interface, the old module key should be used again in the symbol file. Otherwise all depending modules would also be invalidated and would have to be recompiled. A simple method to check whether a module's interface has changed or not is to bytewise compare the new symbol file against the old one. If it has not changed, the old key can be reused. This comparison method requires very stable symbol files: e.g. the ordering of the exported objects should have no effect on the symbol file. This is clearly not fulfilled in the implementation described above (which we've chosen for simplicity), because objects are ordered in the linear list according to their occurence in the module. Desired is a kind of *canonical form* for symbol files.

As we mentioned earlier, access to a special object in the symbol table should be as efficient as possible, so one could implement the table using binary trees instead of (unsorted) linear lists. Then, the objects could be written in alphabetical order to the symbol file using an in–order traversal of the tree. Symbol files in this canonical form are invariant to any changes in the ordering of a module's objects. Unfortunately this implies an undesired side effect which destroys the efficiency of binary trees: Importing a symbol file means inserting the imported objects in binary trees. Because the objects are alphabetically sorted, the symbol table tree degenerates to a linear list. Further, most of the objects in typical modules are known by import. In consequence, searching in the symbol table means actually searching in linear lists. Hence, the additional implementation effort for simple binary trees may be no longer justified.

The situation may be improved by modifing the proposed canonical form. Instead of writing all objects in alphabetical order, groups of objects of the same kind (with the same mode) are written alphabetically: first all constants, then the variables, then types, and so on. Within each group the objects are ordered and the group ordering is also specified. This is another canonical form which is also invariant against any permutations of exported objects. When imported, the trees will not completely degenerate but are decomposed into several partial lists. The result is at least a better balanced tree. Note that the addional effort of traversing the tree in several passes (for each object group) is only necessary during export; the more time critical import procedure is not at all affected.

Predefined Types

In several languages there exist predefined types which are known to the compiler. Such types are always exported using a fixed reference number. Before any object is imported, they are inserted into the local structure table by the import procedure (see Appendix B).

Increased Import Speed

As measurements show (see below), when reading is bytewise, the import speed highly depends on the symbol file length. Besides strings, most of the data on a symbol file are integers. Hence, a principal goal should be to reduce the space used to write a single integer number. Integers occur frequently as tag or reference numbers and most of them are very small (with an absolute value less than 64). Nevertheless also bigger integers should be managed easily. The ReadInt and WriteInt procedures described in [Te90] allow integers to be written in a machine–independent format which uses only one byte per number in most cases.

2.5 Measurements

The following measurements show lengths and reading times of symbol files. The basic modules of the Oberon System are choosen as a typical "module mix". The method described here (used in module CCT) is compared to the method used in the existing portable Oberon compilers (module OPT, see [Cre90]) which is essentially the same method as described in [Gu85]. To be fair, the CCT method was adjusted such that about the same information per object was written out as with OPT (e.g. the address of each parameter of a procedure) but using a compactifying WriteInt procedure. The measurements were made on a Ceres–2 computer [He88] with a NS32532 CPU running at 25 Mhz clock speed.

In the left table the lengths of the symbol files in bytes are shown (OPT lengths are 100%). On average, the CCT symbol files are about 25% shorter than the corresponding OPT files. When the same symbol files are written using a non–compactifying WriteInt procedure in CCT (for addresses only), the files are about 8 % shorter (measurements not shown here). The right table shows the reading times in ms for each symbol file (OPT times are 100%). For this, each file was imported 100 times and only the pure file reading time without directory operations was measured and the average taken. The actual time spent in the import procedures is much higher because of directory accesses and may vary even for the same file.

Module	length OPT	length CCT	in %	time OPT	time CCT	in %
Diskette	497	411	83	13.7	11.6	85
Display	1137	915	80	30.3	29.1	96
Files	519	391	75	14.7	12.8	87
Fonts	242	161	64	5.7	4.9	86
Input	113	102	90	3.4	3.3	97
Math	115	103	90	4.0	3.9	98
MenuViewers	450	258	57	11.1	7.7	69
Oberon	2157	1403	65	56.7	46.4	82
Printer	720	466	65	18.9	13.7	72
SCC	265	206	78	6.9	5.6	81
TextFrames	2126	1470	69	57.2	44.6	78
Texts	1747	1237	71	48.9	38.8	79
V24	170	151	89	5.6	5.0	89
Viewers	653	445	68	17.9	13.1	73
in comparison to OPT		75		in comparison to OPT		84

Tab. 1a and 1b

	OPT	CCT	pure file
average reading time per byte (ms)	0.0268	0.0317	0.0090
regression coefficient r	0.9990	0.9980	0.9999
lines of code for import	196	176	
lines of code for export	163	125	

Tab. 2

Further analysis of the measurements shows what was presumed, namely a practically linear dependency between the file lengths and their reading times (Tab. 2); the linear regression coefficient r is nearly 1.0 for both methods. Although the average reading time per byte in CCT is larger than in OPT, the shorter file lengths made up for this difference. For comparison, the values for pure file reading are also shown (the reading time per byte is the average measured time for this file mix and is only valid for short files in general). In every case the file length completely dominates other factors, so to improve importing speed shorter symbol files have to be achieved. But nevertheless, such an effort is only justified if the file system offers some kind of caching strategy for files already open. Otherwise the reading time for such short files is negligible compared to the directory access time. The import and export procedures in CCT are about 20 % shorter in source code size than the OPT procedures (Tab. 2).

3. Summary

As we have seen, the linearization algorithm for general graphs is very simple and easily adjusted to similar problems. In the example of symbol files it leads to a clean and understandable solution. The simplicity of the algorithm and the fact that only local invariants have to be considered, allows symbol files to be easily extended. The main difference between the algorithm described here and the one described in [Gu85] is that here a pre–order traversal instead of a post–order traversal is used. Using a post–order traversal, several postulates have to be guaranteed all the time, which complicate the algorithm. Nevertheless, cyclic references within types are a problem which has to be handled in a special way. Although the post–order algorithm requires no recursion for reading, the presented algorithm is faster in the average because symbol files are shorter and time used for reading in todays computers is determined mostly by the length of the files which are to be processed.

Acknowledgements

I would like to thank Josef Templ, Cuno Pfister, Clemens Szyperski and H. Mössenböck. Josef answered tricky questions about symbol files; he and Cuno worked as proof readers. H. Mössenböck added valuable comments to the modified algorithm in Appendix A. Not to forget, the entire paper was written using the excellent *Write* text editor of Clemens.

References

[Cre90] R. Crelier.
OP2: A Portable Oberon Compiler.
Computersysteme ETH Zürich, Technical Report No. 125, February 1990.

[Gri90] R. Griesemer.
Seneca – A Language for Numerical Computations on Vectorcomputers.
Conpar 90 Proceedings, Volume on special technical contributions, Zürich, September 1990.

[Gu85] J. Gutknecht.
Compilation of Data Structures: A New Approach to Efficient Modula–2 Symbol Files.
Computersysteme ETH Zürich, Technical Report No. 64, July 1985.

[PfiHeTe91] C. Pfister, B. Heeb, J. Templ.
Oberon Technical Notes.
Companion paper.

[ReMö86] P. Rechenberg, H. Mössenböck.
An Algorithm for the Linear Storage of Dynamic Data Structures.
Internal Paper, University of Linz, Austria, 1986.

[Te90] J. Templ.
SPARC–Oberon. User's Guide and Implementation.
Computersysteme ETH Zürich, Technical Report No. 133, June 1990.

[Wi85] N. Wirth.
A Fast and Compact Compiler for Modula–2.
Computersysteme ETH Zürich, Technical Report No. 64, July 1985.

[Wi88] N. Wirth.
The Programming Language Oberon.
Software – Practice and Experience, 18, 7, July 1988 and Computersysteme ETH Zürich, Technical Report No. 143, November 1990.

Appendix A: A modified Linearization Algorithm

When working with rather large graphs consisting of several thousands or even millions of nodes, it might be impractical to use a translation table *NodeTab* of about the same size as the graph for reading. If only multiple referenced nodes had to be stored in the NodeTab array, this translation table could be much smaller. This can be achieved by a split of the write phase into two subphases. As a desirable side effect, after writing, all preconditions for writing are established again, i.e. no unmarking is necessary. The modifications are described shortly:

Writing: In the first pass all nodes are traversed and the mark field is used as reference counter (procedure Mark). For the mark field of every (reachable) node the following holds:

$$((\text{mark} < 0) \wedge (\text{node not visited yet})) \vee ((\text{mark} > 0) \wedge (\text{mark} = \text{no. of node references}))$$

In the second pass, every node that is referenced more than once (i.e. mark > 1) is written using a special tag and its negative node number is stored in its mark field (which now again fulfills the preconditions for writing).

Reading: A negative node tag designates an already read node. A positive node tag specifies a node which occurs only once (tag is even) or a node which will be referenced again (tag is odd) and therefore must be stored in the NodeTab array.

```
    TYPE
        Node = POINTER TO NodeDesc;
        NodeDesc = RECORD
            mark: INTEGER;                          (* used for linearization; initially < 0 *)
            tag: INTEGER;
(* determines node type; tag > 0 *)
            data: ARRAY Mtag OF INTEGER;            (* N depends on the node type (tag) *)
            link: ARRAY Ntag OF Node                (* M depends on the node type (tag) *)
        END;

    VAR
        NofNodes: INTEGER;
(* node number; initially = 1 *)
        NodeTab: ARRAY MaxRefs OF Node;             (* node table; contains each node referenced more than once
*)

    PROCEDURE Mark(q: Node); (* Pass 1 *)           (* precondition: ∀q: q.mark < 0 *)
        VAR i: INTEGER;
    BEGIN
        IF q # NIL THEN
            IF q.mark < 0 THEN (* first occurence *) q.mark := 1;
                i := 0; WHILE i < Ntag DO Mark(q.link[i]); INC(i) END
            ELSE INC(q.mark)
            END
        END
    END Mark;                                       (* postcondition: ∀q: q is marked *)

    PROCEDURE WriteNode(q: Node); (* Pass 2 *)      (* precondition: (∀q: q is marked) ∧ (NofNodes = 1) *)
        VAR i: INTEGER;
    BEGIN
        IF q = NIL THEN WriteInt(0)
```

```
    ELSIF q.mark < 0 THEN (* already marked *) WriteInt(q.mark)
    ELSE (* first occurence *)
        IF q.mark = 1 THEN (* node occurs only once *) WriteInt(q.tag*2); q.mark := –1
        ELSE (* node is referenced several times *) WriteInt(q.tag*2 + 1); q.mark := –NofNode; INC(NofNodes)
        END;
        i := 0; WHILE i < Mtag DO WriteInt(q.data[i]); INC(i) END;
        i := 0; WHILE i < Ntag DO WriteNode(q.link[i]); INC(i) END
    END
END WriteNode;                                   (* postcondition: ∀q: q.mark < 0 *)

PROCEDURE ReadNode(VAR q: Node);                 (* precondition: (NodeTab[0] = NIL) ∧ (NofNodes = 1) *)
    VAR tag, i: INTEGER;
BEGIN
    ReadInt(tag);
    IF tag <= 0 THEN (* node reference *) q := NodeTab[–tag]
    ELSE (* first occurence *)
        NEW(q, tag DIV 2);
        IF ODD(tag) THEN (* node is referenced several times *) NodeTab[NofNodes] := q; INC(NofNodes) END;
        i := 0; WHILE i < Mtag DO ReadInt(q.data[i]); INC(i) END;
        i := 0; WHILE i < Ntag DO ReadNode(q.link[i]); INC(i) END
    END
END ReadNode;
```

Appendix B: Import / Export in Detail

In the following an extract of a table handler with the complete import / export procedures is shown. A few points are specific to this implementation and hence commented accordingly.

```
CONST
(* implementation restrictions *)
    MaxNofGMods = 32; MaxNofLMods = 24; MaxNofStructs = 256;

(* object modes *)
    Undef* = 0; Scope = 1; Const* = 2; Type* = 3; Var* = 4; VarPar* = 5; XVar* = 6; IVar* = 7; Field* = 8;
    LProc* = 9; XProc* = 10; IProc* = 11; SProc* = 12; Mod* = 13;

(* type forms *)
    Char* = 2; Bool* = 3; SInt* = 4; Int* = 5; LInt* = 6; Set* = 7; Real* = 8;
    LReal* = 9; Cmplx* = 10; LCmplx* = 11; NilTyp* = 13; NoTyp* = 14;
    Proc* = 15; String* = 16; Array* = 17; DynArr* = 18; Record* = 19; Pointer* = 20;

TYPE
    Name* = ARRAY 32 OF CHAR;
    Object* = POINTER TO ObjectDesc;
    Struct* = POINTER TO StructDesc;

    ObjectDesc* = RECORD
        link*, next*: Object;                   (* objects are chained using next *)
        name*: Name;
        typ*: Struct;
        marked*: BOOLEAN;                       (* marked objects are exported *)
        leave*: BOOLEAN;
        mode*: INTEGER;                         (* identifies object kind *)
        mnolev*: INTEGER;                       (* module numbers are <= 0 *)
        a0*, a1*: LONGINT;                      (* object specific data *)
        b0*, b1*: LONGINT                       (* object specific data *)
    END;

    StructDesc* = RECORD
        ref: INTEGER;
(* used as mark field *)
        form*: INTEGER;                         (* identifies structure kind *)
        obj*: Object;                           (* points to type object if it exists *)
        len*, size*: LONGINT;
        base*: Struct;                          (* result–, element–, base– or pointee type *)
        link*: Object                           (* record scope *)
    END;

VAR
    system, universe: Object;                   (* predefined scopes *)
    topScope*, undefObj*: Object;               (* current topScope, error object *)
    firstStructRef: INTEGER;
    nofGMods*: INTEGER;
    GMod*: ARRAY MaxNofGMods OF Object;         (* global module table *)

(* predefined types *)
    undefTyp*, noTyp*, stringTyp*, boolTyp*, charTyp*, sIntTyp*, intTyp*, lIntTyp*, setTyp*,
    realTyp*, lRealTyp*, cmplxTyp*, lCmplxTyp*: Struct;


PROCEDURE err(no: INTEGER);
```

(* *Displays an error message* *)

(* *general table handling* *) ———

```
PROCEDURE NewObject*(VAR obj: Object; VAR name: ARRAY OF CHAR; mode: INTEGER);
    (* Creates a new object and initializes its fields *)

PROCEDURE NewStruct*(VAR str: Struct; form: INTEGER);
    (* Creates a new structure and initializes its fields *)

PROCEDURE OpenScope*(VAR scope: Object; mnolev: INTEGER);
    (* Opens a new scope if scope is NIL; otherwise the old scope is reopened *)

PROCEDURE CloseScope*;
    (* Closes topScope *)

PROCEDURE Insert*(VAR obj: Object; name: ARRAY OF CHAR; mode: INTEGER);
    VAR p, q: Object;
BEGIN
    p := topScope; q := topScope.next;
    WHILE q # NIL DO
        IF q.name = name THEN err(1) END;
        p := q; q := q.next
    END;
    NewObject(obj, name, mode); obj.mnolev := topScope.mnolev; p.next := obj
END Insert;
```

(* *import table handling* *) ———

```
PROCEDURE InsertMod*(VAR mod: Object; VAR name: ARRAY OF CHAR; key: LONGINT);
    VAR i: INTEGER;
BEGIN i := 0;
    WHILE (i < nofGMods) & (name # GMod[i].name) DO INC(i) END;
    IF i < nofGMods THEN  (* module already imported *)
        mod := GMod[i];
        IF mod.b0 # key THEN err(150) (* key inconsistency *) END
    ELSE
        NewObject(mod, name, Mod); (* must not be visible in global scope *)
        mod.b0 := key; mod.mnolev := –nofGMods;
        OpenScope(mod.link, mod.mnolev); CloseScope; (* allocate own scope *)
        IF nofGMods < MaxNofGMods THEN GMod[nofGMods] := mod; INC(nofGMods)
        ELSE err(227) (* to many imported modules *)
        END
    END
END InsertMod;

PROCEDURE InsertImport(VAR obj: Object; scope: Object);
    VAR p, q: Object;
BEGIN
    p := scope; q := scope.next;
    WHILE q # NIL DO
        IF q.name = obj.name THEN obj := q; RETURN END;
        p := q; q := q.next
    END;
    obj.mnolev := scope.mnolev; p.next := obj
END InsertImport;
```

(* *import* *) ———

```
PROCEDURE OpenRider(VAR R: Files.Rider; name: ARRAY OF CHAR; VAR res: INTEGER);
    (* Sets a rider to the beginning of the symbol file *)

PROCEDURE Import*(VAR substName, impName, selfName: ARRAY OF CHAR);
    VAR
```

```
    R: Files.Rider;
    mod0, mod: Object;
    res, nofLMods, nofStructs: INTEGER;
    LMod: ARRAY MaxNofLMods OF Object;
    LStruct: ARRAY MaxNofStructs OF Struct;

PROCEDURE^ ReadStruct(VAR typ: Struct);

PROCEDURE ReadInt(VAR i: LONGINT); (* Reads integers written in a compacted form [Te90] *)
    VAR n: LONGINT; s: INTEGER; x: CHAR;
BEGIN
    s := 0; n := 0; Files.Read(R, x);
    WHILE ORD(x) >= 128 DO INC(n, ASH(ORD(x) – 128, s)); INC(s, 7); Files.Read(R, x) END;
    i := n + ASH(ORD(x) MOD 64 – ORD(x) DIV 64 * 64, s)
END ReadInt;

PROCEDURE ReadName(VAR name: ARRAY OF CHAR);
    (* Reads a name terminated with 0X *)

PROCEDURE ReadString(VAR pos, len: LONGINT);
    (* Reads a string terminated with 0X into the scanner string buffer *)

PROCEDURE ReadKey(VAR x: LONGINT);
    (* Reads an integer in uncompacted form *)

PROCEDURE ReadMod(VAR mod: Object);
    VAR ref, key: LONGINT; name: Name;
BEGIN
    ReadInt(ref);
    IF ref > 0 THEN (* first occurence *)
        ReadKey(key); ReadName(name);
        IF name = selfName THEN err(49) END;
        InsertMod(mod, name, key);
        IF nofLMods < MaxNofLMods THEN LMod[nofLMods] := mod; INC(nofLMods)
        ELSE err(227) (* to many imported modules *)
        END
    ELSE mod := LMod[–ref]
    END
END ReadMod;

PROCEDURE ReadFields(VAR field: Object);
    VAR obj, last: Object; name: Name;
BEGIN
    field := NIL;
    LOOP
        ReadName(name);
        IF name = "" THEN EXIT END;
        NewObject(obj, name, Field); ReadStruct(obj.typ); ReadInt(obj.a0);
        IF field = NIL THEN field := obj ELSE last.next := obj END;
        last := obj
    END
END ReadFields;

PROCEDURE ReadFP(VAR par: Object; VAR nofArgs: LONGINT);
    VAR obj, last: Object; typ: Struct; n, a0, a1: LONGINT; name: Name;
BEGIN
    par := NIL; ReadInt(nofArgs); n := nofArgs;
    WHILE n > 0 DO
        ReadName(name); ReadStruct(typ); ReadInt(a0); ReadInt(a1);
        IF ODD(a1) THEN NewObject(obj, name, VarPar)
        ELSE NewObject(obj, name, Var)
        END;
        obj.typ := typ; obj.a0 := a0; obj.a1 := a1 DIV 2;
        IF par = NIL THEN par := obj ELSE last.next := obj END;
        last := obj; DEC(n)
```

```
    END
END ReadFP;

PROCEDURE ReadStruct(VAR typ: Struct);
    VAR form: LONGINT; htyp: Struct; name: Name; obj, mod: Object;
BEGIN
    ReadInt(form);
    IF form <= 0 THEN (* struct reference or NIL *) typ := LStruct[–form]
    ELSE (* first occurence *)
        NewStruct(htyp, SHORT(form)); ReadName(name);
        IF name # "" THEN (* named type *)
            NewObject(obj, name, Type); obj.marked := TRUE; obj.typ := htyp; htyp.obj := obj;
            ReadMod(mod); InsertImport(obj, mod.link);
            typ := obj.typ
        ELSE typ := htyp
        END;
        IF nofStructs < MaxNofStructs THEN LStruct[nofStructs] := typ; INC(nofStructs)
        ELSE err(228) (* to many imported types *)
        END;
        ReadStruct(htyp.base);
        CASE form OF
            | Proc: OpenScope(htyp.link, 0); ReadFP(htyp.link.next, htyp.len); CloseScope; htyp.size := 1
            | Array: ReadInt(htyp.len); ReadInt(htyp.size)
            | DynArr: ReadInt(htyp.size)
            | Record:
                OpenScope(htyp.link, 0); ReadFields(htyp.link.next); CloseScope; ReadInt(htyp.size);
                IF htyp.base # NIL THEN htyp.len := htyp.base.len+1; htyp.link.link := htyp.base.link
                ELSE htyp.len := 0
                END
            | Pointer: typ.size := 1
        END
    END
END ReadStruct;

PROCEDURE ReadObjects(scope: Object);
    VAR mode: LONGINT; obj: Object; typ: Struct; name: Name;
BEGIN
    LOOP (* read all objects *)
        ReadInt(mode);
        IF mode = Undef THEN EXIT
        ELSIF mode = Type THEN ReadName(name);
            IF name # "" THEN (* alias type *)
                NewObject(obj, name, Type); ReadStruct(obj.typ);
                InsertImport(obj, scope)
            ELSE (* other types *) ReadStruct(typ)
            END
        ELSE
            ReadName(name); NewObject(obj, name, SHORT(mode)); ReadStruct(obj.typ);
            IF mode = Const THEN
                IF obj.typ.form = String THEN ReadString(obj.b0, obj.b1)
                ELSIF obj.typ.form = LReal THEN ReadInt(obj.b0); ReadInt(obj.b1)
                ELSE ReadInt(obj.b0)
                END
            ELSIF obj.mode = Var THEN ReadInt(obj.a0); obj.mode := XVar
            ELSIF obj.mode = LProc THEN
                OpenScope(obj.link, 0); ReadFP(obj.link.next, obj.b0); CloseScope; obj.mode := XProc
            ELSIF obj.mode = IProc THEN
                OpenScope(obj.link, 0); ReadFP(obj.link.next, obj.b0); CloseScope
            END;
            InsertImport(obj, scope)
        END
    END
END ReadObjects;

PROCEDURE EnterStruct(typ: Struct);
```

```
        (* Enters a predefined type into the LStruct table *)

    BEGIN
        IF impName = "SYSTEM" THEN Insert(mod, impName, Mod); mod.link := system
        ELSE OpenRider(R, impName, res);
            IF res < 0 THEN (* no error occured *)
                LStruct[0] := NIL; EnterStruct(stringTyp);
                EnterStruct(undefTyp); EnterStruct(noTyp); EnterStruct(boolTyp); EnterStruct(charTyp);
                EnterStruct(sIntTyp); EnterStruct(intTyp); EnterStruct(lIntTyp); EnterStruct(setTyp);
                EnterStruct(realTyp); EnterStruct(lRealTyp); EnterStruct(cmplxTyp); EnterStruct(lCmplxTyp);
                nofLMods := 0; nofStructs := firstStructRef;
                ReadMod(mod0); ReadObjects(mod0.link);
                Insert(mod, substName, Mod);
                mod.b0 := mod0.b0; mod.mnolev := mod0.mnolev; mod.link := mod0.link
            ELSE err(res)
            END
        END
    END Import;


(* export *) ——————————————————————————————————————————

    PROCEDURE Export*(mod: Object; VAR buf: ARRAY OF CHAR; VAR len: LONGINT; VAR new: BOOLEAN);
        VAR pos: LONGINT; nofLMods, nofStructs: INTEGER;

        PROCEDURE^ WriteStruct(typ: Struct);

        PROCEDURE Write(ch: CHAR);
        BEGIN buf[pos] := ch; INC(pos)
        END Write;

        PROCEDURE WriteInt(i: LONGINT); (* Writes integers in a compacted form [Te90] *)
        BEGIN
            WHILE (i < –64) OR (i > 63) DO Write(CHR(i MOD 128 + 128)); i := i DIV 128 END;
            Write(CHR(i MOD 128))
        END WriteInt;

        PROCEDURE WriteName(VAR name: ARRAY OF CHAR);
            (* Writes a name terminated with 0X *)

        PROCEDURE WriteString(pos: LONGINT);
            (* Writes the string terminated with 0X at position pos of the scanner string buffer *)

        PROCEDURE WriteKey(x: LONGINT);
            (* Writes an integer in uncompacted form *)

        PROCEDURE WriteMod(mod: Object);
        BEGIN
            IF mod.b1 < 0 THEN (* first occurence *)
                mod.b1 := nofLMods; INC(nofLMods);
                WriteInt(Mod); WriteKey(mod.b0); WriteName(mod.name)
            ELSE WriteInt(–mod.b1)
            END
        END WriteMod;

        PROCEDURE WriteFields(field: Object);
        BEGIN
            WHILE field # NIL DO
                IF field.marked THEN WriteName(field.name); WriteStruct(field.typ); WriteInt(field.a0) END;
                field := field.next
            END;
            Write(0X)
        END WriteFields;

        PROCEDURE WriteFP(par: Object; nofArgs: LONGINT);
```

```
BEGIN
    WriteInt(nofArgs);
    WHILE nofArgs > 0 DO
        WriteName(par.name); WriteStruct(par.typ); WriteInt(par.a0);
        IF par.mode = VarPar THEN WriteInt(par.a1*2 + 1) ELSE WriteInt(par.a1*2) END;
        par := par.next; DEC(nofArgs)
    END
END WriteFP;

PROCEDURE WriteStruct(typ: Struct);
    VAR name: Name;
BEGIN
    IF typ = NIL THEN WriteInt(0)
    ELSIF typ.ref > 0 THEN (* already marked *) WriteInt(–typ.ref)
    ELSE (* first occurence *)
        WriteInt(typ.form); typ.ref := nofStructs; INC(nofStructs);
        IF typ.obj # NIL THEN (* named type *)
            name := typ.obj.name;
            IF ~typ.obj.marked THEN (* invisible type *)
                name[0] := CHR(ORD(name[0]) – ORD("@"))
            END;
            WriteName(name); WriteMod(GMod[–typ.obj.mnolev])
        ELSE Write(0X)
        END;
        WriteStruct(typ.base);
        CASE typ.form OF
            | Proc: WriteFP(typ.link.next, typ.len)
            | Array: WriteInt(typ.len); WriteInt(typ.size)
            | DynArr: WriteInt(typ.size)
            | Record: WriteFields(typ.link.next); WriteInt(typ.size)
            | Pointer:
        END
    END
END WriteStruct;

PROCEDURE WriteObjects(obj: Object);
BEGIN
    WHILE obj # NIL DO
        IF obj.marked THEN
            WriteInt(obj.mode);
            IF (obj.mode # Type) OR (obj.typ.obj # obj) THEN (* no–type or alias type *)
                WriteName(obj.name)
            ELSE (* other type *) Write(0X)
            END;
            WriteStruct(obj.typ);
            IF obj.mode = Const THEN
                IF obj.typ.form = String THEN WriteString(obj.b0)
                ELSIF obj.typ.form = LReal THEN WriteInt(obj.b0); WriteInt(obj.b1)
                ELSE WriteInt(obj.b0)
                END
            ELSIF obj.mode = Var THEN WriteInt(obj.a0)
            ELSIF obj.mode IN {LProc, IProc} THEN WriteFP(obj.link.next, obj.b0)
            END
        END;
        obj := obj.next
    END;
    WriteInt(Undef) (* termination tag *)
END WriteObjects;

PROCEDURE Compare(VAR buf: ARRAY OF CHAR; len: LONGINT; VAR new: BOOLEAN);
    VAR res: INTEGER; i: LONGINT; R: Files.Rider; ch: CHAR; prefix: ARRAY 5 OF CHAR;
BEGIN
    OpenRider(R, mod.name, res); new := TRUE;
    IF res < 0 THEN
        Files.ReadBytes(R, prefix, LEN(prefix)); Files.Read(R, ch); i := LEN(prefix);
```

```
            WHILE (i < len) & (buf[i] = ch) DO Files.Read(R, ch); INC(i) END;
            IF i = len THEN (* same symbol table => use old key *) i := 0;
                WHILE i < LEN(prefix) DO buf[i] := prefix[i]; INC(i) END;
                new := FALSE
            END
        END
    END Compare;

BEGIN
    pos := 0; nofLMods := 0; nofStructs := firstStructRef;
    WriteMod(mod); WriteObjects(mod.link.next);
    Compare(buf, pos, new)
END Export;
```

Oberon Technical Notes

Cuno Pfister (ed.)

The purpose of the Oberon technical notes is to provide the implementor of an Oberon system with the experience gained during the implementation efforts undertaken at the *Institut für Computersysteme* at ETH. Furthermore they give an overview over work already done or under way. This report contains the first five technical notes.

Table of Contents

1. Oberon Implementations

Cuno Pfister

The original Oberon implementation [1] has been realized by N. Wirth and J. Gutknecht for the Ceres workstation [2]. Several ports of the system to other machines have been completed since then. We will give a short description of each of those projects. Differences to the original implementations are described.

Ceres (National Semiconductor NS32x32)

The original implementation. Oberon is the basic operating system. Module Display is written in assembly language. The garbage collector is a mark–and–sweep garbage collector which runs only between commands, i.e. it does not need to handle pointers on the stack. An access to a freed module results in a trap on the Ceres–1 and Ceres–2, but goes undetected on the Ceres–3. The heap allocator/garbage collector is written in assembly language and linked together with the inner core modules.

Sun SPARCstation (Sun SPARC)

This implementation [3] runs on top of SunOS as a Unix process. Oberon takes over the whole screen. The display operations are based on the SUN Pixrect routines. Oberon files are mapped to Unix files. A variant of the buddy system strategy is used for heap allocation. The garbage collector is a mark–and–sweep garbage collector which usually runs between commands, but when NEW fails due to a memory shortage, the garbage collector is started also. This collector also takes pointers on the stack into account. This is done by treating each memory location on the stack as a possible pointer value. To find out whether a memory word is a pointer, it is subjected to some plausibility tests, such as testing whether the value points to a possible block address in the heap and whether a possibly valid type tag is found there. If the plausibility tests succeed, the heap is traversed sequentially to find the corresponding block. If the block is found, the tested memory location is treated as a root for the garbage collector. Modules are never freed but merely removed from the module list, i.e. an access to a freed module cannot be detected and aborted. The loader and heap allocator/garbage collector are written in Modula–2 and linked as a Unix application.

Apple Macintosh II (Motorola MC68020)

This implementation [4, 5] runs on top of the MacOS as a (MultiFinder friendly) application. Oberon runs in one Macintosh window. The display operations are largely based on the Apple QuickDraw routines. Oberon files are mapped to Macintosh files. An integrated allocator/collector is used (see technical note # 2). Modules are never freed but merely removed from the module list, i.e. an access to a freed module cannot be detected and aborted. The type descriptors contain information about procedure variables in records, such that a checked version of the System.Free command could be implemented in the future. The loader, heap allocator/garbage collector and some raster operations are written in assembly language and linked as a Macintosh application.

DEC DECstation (MIPS R2000)

This implementation runs on top of Ultrix as a process. Oberon runs in an X–window. The display operations are based on X–windows. Oberon files are mapped to Ultrix files. An integrated allocator/collector is used (see technical note # 2). Modules are never freed but merely removed from the module list, i.e. an access to a freed module cannot be detected and aborted. The loader is written in C and linked as a Unix application.

IBM S/6000 (IBM S/6000)

This project has recently been started.

IBM PS/2 (Intel 80386)

This project has recently been started.

Others

Other Oberon compiler back–ends have been written by students, but the system was not ported. The compiler back–ends available produce code for the following processors:

– a virtual stack machine (similar to P–Code or M–Code)
– Intel 8086
– Intel 80386
– INMOS T800 Transputer
– C language (Oberon subset to C translator, for bootstrapping a compiler)

References

1. Wirth N, Gutknecht J (1989), The Oberon System,
   Software–Practice and Experience, 19 (9), 857–893
2. Eberle H (1987), Development and Analysis of a Workstation Computer,
   Ph. D. thesis no. 8431, ETH Zürich
3. Templ J (1990), SPARC–Oberon, User’s Guide and Implementation,
   Report 133, ETH Zürich
4. Franz M (1990), The Implementation of MacOberon,

Report 141, ETH Zürich
5. Franz M (1990), MacOberon Reference Manual, Report 142, ETH Zürich

2. An Integrated Heap Allocator/Garbage Collector

Beat Heeb, Cuno Pfister

Abstract

Heap Allocation and Garbage Collection are fundamental services of an Oberon implementation. It is shown how a simple and efficient implementation of these services can be attained.

Introduction

Programs for personal computers become increasingly loaded with features, most of them rarely needed. The reason for that is, apart from the marketing pressure to advertise more features than the competition, the desire to provide all the features that anyone might ever need or want. The result is that programs become ever larger, more complex, buggier, more expensive and sometimes delivered years after their announcements. Even then they often fail to provide features useful for a particular task.

A solution to this problem lies in the development of extensible programs. Extensibility here means that a program can provide the customer with only the features needed most of the time and with some means to extend it. If the customer needs some special service, he (or usually a third party) can implement this service himself, without having access to the original program's source code.

For example, imagine an extensible page layout program which supports text boxes, draw boxes and bitmap boxes as standard box types, and commands to operate upon them. A user may have special needs concerning the available commands, like e.g. a command which aligns the selected objects in a document in a special way. It should be possible for him to write such a command, which operates on the exported document data structure. This poses a subtle problem, though. The command implementor may generate references, i.e. pointers, to an exported data structure, and these references are not known to the basic layout program. This means in particular that when the layout program disposes of the storage used by a document, there may still be pointers around which reference this storage. Such pointers are called *dangling pointers*. Dangling pointers are one of the most frequent and most dangerous sources of program malfunctions. Their use often results in the destruction of data not belonging to the erroneous module, and the destruction may not be detected for a long time. Such an error is difficult to track down, consequently it is difficult to determine who is responsible for an accident caused by the error. So extensibility leads to a loss of control over references and thereby to an increased probability for dangling pointers. We will come back to that shortly.

The user of a program like the one described above should also be able to write an extension that supports special table boxes, for instance. Such an extension consists of a module which implements the data type Table, together with the particular behaviour of an instance of this type, like displaying, storing and reading itself. A document might then contain text boxes together with table boxes at the same time. This example is typical in that a new data type, which is similar to existing types, is introduced, and that variables of similar data types are integrated in the same data structure (the document).

The object–oriented programming style is well suited for the implementation of such a program, since it allows the definition of similar, i.e. compatible types (e.g. Table is a subtype of Box) and since control can be delegated to the subtype by means of the overriding facility (e.g. Table implements its own Draw method). For our discussion the terms record and object can be used interchangeably.

Our example has shown the integration of different objects types, potentially implemented by different programmers, in the same data structure. Close integration is obviously useful. On the other hand, it magnifies the problems associated with malfunctioning objects by making the integrity of a data structure dependant on a potentially large number of object implementations. Especially errors which have non–local effects are dangerous. This leads us to the conclusion that while extensibility makes dangling pointers more probable, integration of extensions makes the effects of dangling pointers graver.

It is possible to prevent this type of errors going undetected by using a type–safe language. To prevent dangling pointers, the implementation of a type–safe language must guarantee that every pointer variable is initialized correctly and that a heap record is only released when there are no pointers referencing it anymore. The latter task is performed by a garbage collector. When a garbage collector can prove that a heap record is not referenced anymore, it reclaims the corresponding heap area for the storage allocator.

Automatic garbage collection is known for a long time, but has not found its way into production languages like Pascal or C, not even into their object–oriented descendants. A garbage collector may reduce the response time and the performance of a program dramatically, or may require memory sizes several times as large as would be the case without a garbage collector. Additional reasons why garbage collectors are not popular are the problems caused by the lack of type–safety in the mentioned languages, e.g. posed by untagged variant records, and the complexity involved in handling arbitrary record types.

We want to show how a simple and efficient garbage collector for Oberon [1] can be written. Oberon is a type–safe language derived from Modula–2 which allows for an object–oriented programming style. Oberon is also the name of an operating system and window system [2]. The kernel of an Oberon system provides, among other things, a storage allocator and a garbage collector. Oberon has first been implemented on the Ceres workstation [3]. Our approach is not a radical departure from the Ceres implementation, but rather a refinement which improves upon memory utilization and implementation complexity.

Heap Allocation

A simple storage allocation algorithm is presented.

Small Blocks

Storage is allocated in blocks. Block sizes are multiples of a minimal size *B*. The size *s* of an allocated block is at least the size of variables of the record type bound to the pointer type of *p*. This size is known during compilation. For the allocation it is rounded up to the next multiple of *B*.

There is a free list for all supported block sizes, i.e. free blocks of the same size are linked by a simple linear list. The free lists are anchored in a global array *A* (which could be declared as *ARRAY 1..N OF ADDRESS*) with element number *i* corresponding to the free list for size *i * B*.

Allocation of a block of size s consists in removing a block from free list *A[k]* where

$$(k \geq s / B) \wedge (\forall i: s / B \leq i < k: A[i] = NIL) \wedge (A[k] \neq NIL).$$

If $k = s / B$ then the address of the block is returned. If $k > s / B$ then the block is split into two blocks, the first of size *s* and the second of size $r = k * B - s$. The second block is inserted in the free list *A[r / B]* and the address of the first block is returned.

The Ceres implementation is different in that it uses blocks with sizes restricted to powers of two. This leads to an increased internal fragmentation of the memory. In our scheme, each heap variable wastes less than half of the *minimal* block size. This in contrast to a waste of less than half of the *particular* block size. The most notable other difference is that our scheme is simpler to implement.

Large Blocks

It is reasonable to restrict the size of array *A* such that it supports only small blocks (e.g. less than about 100 bytes). Almost all allocations are done with small block sizes, thus a less efficient allocation strategy can be used for large blocks. A very simple solution is to use *A[N]* as a free list for blocks of variable size (i.e. size‡*(N + j) * B*) and performing a first fit allocation [4] whenever this list must be used. Note that the support for this special case fits naturally into the allocation procedure, it merely adds one line of code.

The following pseudo–code listing shows the complete allocation routine:

```
procedure Allocate(var a: address; size: longint);
   var i: integer; r, l: address;
begin
   i := min(size / B, N);    (* calculate index and restrict it to a maximal value *)
   while (i < N) & (A[i] = NIL) do INC(i) end;    (* search smallest non–empty free list *)
   l := adr(A[i]); a := l^;    (* address and value of pointer to first free block *)
   while (a # NIL) & (a^.size < size) do l := ADR(a^.next); a := l^ end;    (* first fit if i = N *)
   if a # nil then
      l^ := a^.next;    (* remove block from free list *)
      if a^.size > size then    (* block must be split *)
         i := min((a^.size – size) / B, N); r := a + size;
         r^.size := a^.size – size;    (* adjust size of residual block *)
         r^.next := A[i]; A[i] := r    (* insert residual block in free list *)
```

```
      end
    end
  end Allocate;
```

This algorithm doesn't support fast arbitrary block deallocation, because a freed block cannot efficiently be merged with its lower–address neighbour and because only simple linear lists are used for the free lists, which prohibits fast removal of a block from its free list. In the next chapter we will show how a garbage collector circumvents the need for fast arbitrary block deallocation.

Garbage Collection

The Ceres implementation of Oberon uses a mark–and–sweep garbage collector [5] for heap storage reclamation. In Oberon, most temporary variables are local and therefore allocated and deallocated on the stack. Thus relatively little garbage is produced compared to typical Lisp or Smalltalk systems. This explains why the Ceres garbage collector proved adequate in practice, contradicting the statement that "Mark–and–sweep automatic storage reclamation does not seem to be practical on contemporary (1988) computers" [6].

A mark–and–sweep collector works in two phases. In the mark phase, all objects which still can be referenced are marked. In the sweep phase, all heap blocks are traversed sequentially. The ones which have not been marked are reclaimed.

Mark Phase

Let us first consider a simple recursive procedure, which marks all objects reachable from a given pointer:

```
  procedure Mark(q: address);
    var off: address;
  begin
    if (q # NIL) & (Unmarked(q)) then
      SetMark(q); off := FirstPointerOffset(q);
      while off >= 0 do
        Mark(mem[q + off]); off := NextPointerOffset(q, off)
      end
    end
  end Mark;
```

This pseudo–code procedure uses four auxiliary procedures. The procedure *FirstPointerOffset* determines the record field which contains the first pointer. The procedure *NextPointerOffset* repeatedly yields the next record field containing a pointer. A negative offset is used as a terminating sentinel.

To implement *FirstPointerOffset* and *NextPointerOffset* it must be possible to efficiently find out the offsets of a record's pointer fields. These offsets are the same for all variables of this record's type. Thus it is reasonable to provide a so–called *type* *descriptor* containing a table with all these offset values. Every heap record now needs a pointer to this type descriptor. This pointer is a hidden record field called a *type tag* and is usually located at offset *–PtrSize* in the record. The allocation procedure is extended such that it also initializes this tag. (For every record type, the compiler reserves a global variable anchoring the type descriptor. The contents of the appropriate variable is passed to the allocation routine as an additional parameter.)

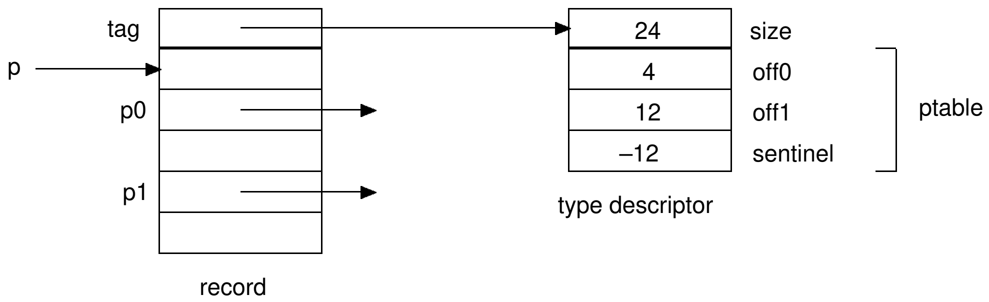


Figure 1: Example of a record variable and its type descriptor

Figure 1 shows the descriptor of a type *T*. It contains the fields *size* and *ptable*. *ptable* is a table of pointer offsets describing where in a variable of type *T* a pointer can be found. This table, which varies from type to type, is terminated by a negative valued sentinel.

The procedure *Unmarked* tests whether an object has already been marked, the procedure *SetMark* marks an object under the assumption that it is unmarked (i.e. *Unmarked(q)* is a precondition of *SetMark(q)*). We won't go into details about how marking is realized. It should be sufficient to say that one bit of the type tag can be used for marking, usually either the sign bit or the least significant bit.

The following version of the above procedure replaces *FirstPointerOffset* and *NextPointerOffset* by an increment of the type tag by the size of a pointer:

```
procedure Mark(q: address);
begin
  if (q # NIL) & (Unmarked(q)) then
    SetMark(q); Increment(Tag(q));
    while mem[Tag(q)] >= 0 do
      Mark(mem[q + mem[Tag(q)]]); Increment(Tag(q))
    end;
    RestoreTag(q)
  end
end Mark;
```

This means that the type tag of a record changes during the mark phase such that it always points to the offset to be processed and after the offsets already processed (see Figure 2).

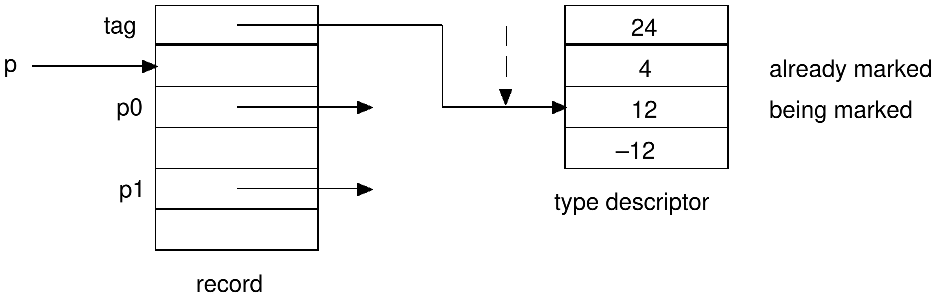


Figure 2. Type Tag during a Mark Phase

The loop over the offsets terminates when a negative offset is found. The value of this offset can be initialized such that *RestoreTag* simply becomes

```
Tag(q) := Tag(q) + mem[Tag(q)].
```

We now transform this procedure such that the guard (q # NIL) & (Unmarked(q)) is moved outside of the mark procedure.

```
procedure Mark(q: address);
   var r: address;
begin   (* (q # NIL) & JustMarked(q) *)
   loop
      Increment(Tag(q));
      if mem[Tag(q)] >= 0 then
         r := mem[q + mem[Tag(q)]];
         if (r # NIL) & (Unmarked(r) then SetMark(r); Mark(r) end
      else RestoreTag(q); return
      end
   end
end Mark;
```

JustMarked means that the object is marked, but none of its descendants. To eliminate recursion, we introduce an explicit stack. The recursive call is replaced by *Push(q); q := r* and the RETURN is replaced by *Pop(q)*:

```
procedure Mark(q: address);
   var r: address;
begin
   Stack := Empty;
   loop
      Increment(Tag(q));
      if mem[Tag(q)] >= 0 then
         r := mem[q + mem[Tag(q)]];
         if (r # NIL) & Unmarked(r) then SetMark(r); Push(q); q := r end
      else
```

```
      RestoreTag(q);
      if Stack = Empty then exit else Pop(q) end
    end
  end
end Mark;
```

The drawback of this procedure is the use of an additional stack, i.e. of additional memory. In the algorithm of Deutsch/Schorr/Waite [7], the stack is distributed to the individual pointer locations which are being traversed. We use a variable *p* as stack pointer, i.e. as pointer to the object containing the predecessor. The predecessor is contained in one of the pointer fields, namely the one currently being processed. The old value of this pointer field is either held in the auxiliary variable *r* or on the stack also. This leads to the following replacements:

```
Stack = Empty   –>  p = NIL
Push(q)         –>  mem[q + mem[Tag(q)]] := p; p := q
Pop(q)          –>  a := p + mem[Tag[p]]; r := mem[a]; mem[a] := q; q := p; p := r
```

```
procedure Mark(q: address);
  var p, r, a: address;
begin
  p := NIL;
  loop
    Increment(Tag(q));
    if mem[Tag(q)] >= 0 then
      r := mem[q + mem[Tag(q)]];
      if (r # NIL) & Unmarked(r) then
        SetMark(r); mem[q + mem[Tag(q)]] := p; p := q; q := r
      end
    else
      RestoreTag(q);
      if p = NIL then exit
      else a := p + mem[Tag[p]]; r := mem[a]; mem[a] := q; q := p; p := r
      end
    end
  end
end Mark;
```

In the appendix there is a listing of an actual implementation of the mark procedure written in MC68000 assembly language. This implementation also takes into consideration that additional data (which is not important for our discussion) must be stored in the type descriptor, between the *size* field and *ptable*.

Concerning the type descriptors we should add that they may be treated just as any other records. Thus they need their own type descriptors. These meta type descriptors differ only in their *size* fields. They in turn can share a common meta meta type descriptor, whose tag points back to itself, i.e. it is its own type descriptor. It may be more practical though to mark type descriptors in some way and to treat them as special cases.

Sweep Phase

In the sweep phase the heap is traversed sequentially, block by block. To do that, the size of all blocks must be known. The sum of a block's address and its size yields the address of the next block. Since there is a type tag at the beginning of each allocated block, such a block's size can be found by inspecting the type descriptor to which the tag points. A free block can be treated the same way, with the difference that it is its own "type descriptor". How this can be done is shown in Figure 3.

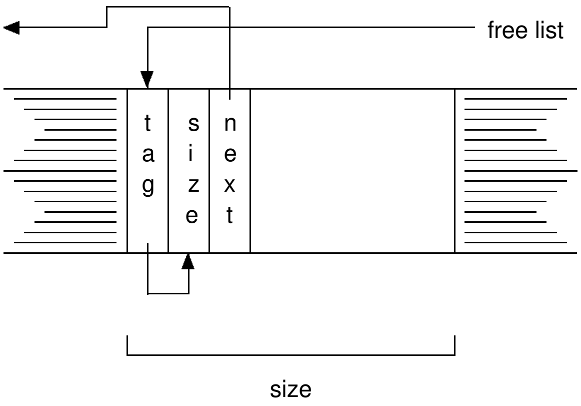


Figure 3: Structure of a free block

At the beginning of the sweep phase, the free lists are all cleared. The sweep constructs completely new free lists by treating consecutive unmarked blocks as single large blocks and inserting them in the appropriate free lists. All marks are cleared.

```
procedure Scan;
   var p: address;
begin
   A[1..N] := NIL;
   q := HeapStart;
   repeat
      while (q # MemSize) & Marked(q) do
         ResetMark(q); q := q + mem[mem[q]]
      end;
      if q # MemSize then
         p := q;
         repeat
            q := q + mem[mem[q]]
         until (q = MemSize) or Marked(q);
         Insert(p, q – p)
      end
   until q = MemSize
end Scan;
```

The procedure *Insert(p, s)* inserts a free block at address *p* in the free list for size *s*. The invariant over this "merge–sweep" is that all free blocks that have been traversed already are of maximal size, i.e. merged. Only the block visited most recently might have to be merged with the next one. This invariant is a principal difference between the merge–sweep deallocation and the deallocation of arbitrary blocks.

Acknowledgements

We would like to thank H. Mössenböck, N. Wirth, R. Griesemer and W. Weck.

References

1. Wirth N (1988) The Programming Language Oberon. Software–Practice and Experience, 18 (7), 661–670
2. Wirth N, Gutknecht J (1989) The Oberon System. Software–Practice and Experience, 19 (9), 857–893
3. Eberle H (1987) Development and Analysis of a Workstation Computer, Ph. D. thesis no. 8431, ETH Zürich
4. Knuth D (1973) The Art of Computer Programming, Addison–Wesley
5. McCarthy J (1960) Recursive Functions of Symbolic Expressions and Their Computation by Machine, I, Comm. ACM, 3, 184–195
6. Ungar D, Jackson F (1988), Tenuring Policies for Generation–Based Storage Reclamation, OOPSLA '88 Proceedings, 107–118
7. Schorr H, and Waite W (1967), An efficient machine–independent procedure for garbage collection in various list structures, Comm. ACM, 10 (8), 501–505

Appendix

The following listing shows an implementation of the mark phase for one root pointer in MC68000 assembly language. A complete implementation would additionally have to iterate over all global (and possibly all local) pointer variables as roots.

Note that the mark bit in the type tag is set during the whole traversal of a record, thus it can be statically compensated for by using an offset of –1 when addressing relative to the type tag.

```
* 68000 mark phase for garbage collector
* A0:     pointer to father
* A1:     pointer to node
* A2:     temporary, for pointer rotation
* A3:     tag or pointer to current pointer offset
*         pointer offsets are usually accessed via A3 with an offset of Offset(ptable) – 4 – 1.
*         The pointer is incremented before it is accessed, thus the subtraction of PtrSize.
*         The subtraction of 1 comes from the set mark bit (bit # 0).
* D0:     offset
* D1:     temporary
```

```
PtrSize  EQU      4                     size of pointers and offsets
Tag      EQU      –4                    offset of type tag
TagL     EQU      Tag+3                 low byte of type tag
Mark     EQU      0                     mark bit (in TagL)
PTab     EQU      36                    ptable offset
Offset   EQU      PTab–PtrSize–1

Start    MOVE.L   A1,D1                 NIL test
         BEQ      End                   NIL
         BSET.B   #Mark,TagL(A1)        test and set mark bit
         BNE      End                   marked
         MOVE.L   #0,A0                 father := NIL
         MOVE.L   Tag(A1),A3            load first tag
         BRA      Loop

Up       ADD.L    D0,A3                 adjust tag
         MOVE.L   A3,Tag(A1)            save tag
         MOVE.L   A0,D1                 NIL test
         BEQ      End                   father = NIL (sentinel)
         MOVE.L   Tag(A0),A3            load father.tag
         MOVE.L   Offset(A3),D0         load offset
         MOVE.L   (A0,D0),A2            rotate pointers, step 1
         MOVE.L   A1,(A0,D0)            rotate pointers, step 2
         MOVE.L   A0,A1                 rotate pointers, step 3
         MOVE.L   A2,A0                 rotate pointers, step 4

Loop     ADDQ.L   #PtrSize,A3           address of next offset
         MOVE.L   Offset(A3),D0         load next offset
         BMI      Up                    negative sentinel reached, i.e. end of list
         MOVE.L   (A1,D0),A2            load son (and rotate pointers, step 1)
         MOVE.L   A2,D1                 NIL test
         BEQ      Loop                  NIL
         BSET.B   #Mark,TagL(A2)        test and set mark bit
         BNE      Loop                  marked

Down     MOVE.L   A3,Tag(A1)            save tag
         MOVE.L   A0,(A1,D0)            rotate pointers, step 2
         MOVE.L   A1,A0                 rotate pointers, step 3
         MOVE.L   A2,A1                 rotate pointers, step 4
         MOVE.L   Tag(A1),A3            load new tag
         BRA      Loop

End
```

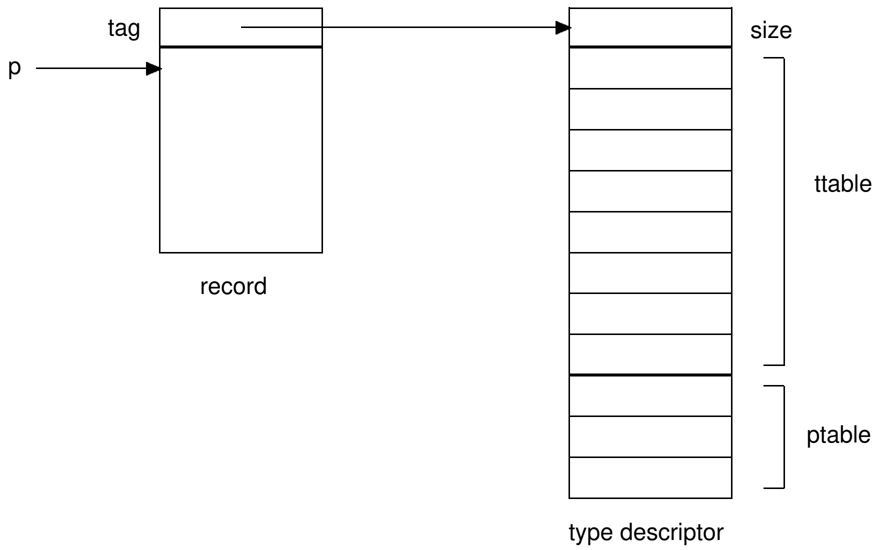
3. Type Guards and Type Tests

Cuno Pfister

In Oberon, indirectly referenced record variables (i.e. referenced by pointer or passed as VAR parameter) may have a different type at run–time than the one which is declared statically. An Oberon implementation must guarantee that the actual type of a record is an extension of the record's declared type. This is done with the aid of type guards [1]: A type guard tests whether the type of an indirectly referenced record variable is an extension of some statically declared type. If not, the program is aborted. A type test is similar to a type guard, but instead of aborting a program it returns the value FALSE, otherwise TRUE. We present a scheme which allows a very efficient implementation of type guards and type tests.

A record type is represented at run–time by a type descriptor (see Figure). In this type descriptor there is a table of pointers (*ttable*), at a fixed offset and with a fixed size. The pointer in entry 0 points to the type descriptor of the original base type of the extension hierarchy (level 0 type). Entry 1 points to the first extension (level 1 type), entry 2 to the extension of the first extension, and so on. If the type descriptor denotes a level n type, the first n + 1 entries are used. The entries for higher level types are set to NIL. The level 0 entry may even be omitted, since a type guard on a base type is never executed at run–time. Nevertheless it is recommended to include it, for an example of where it can be useful see technical note # 4.

Obviously the depth of the extension hierarchy is limited by such an arrangement. We recommend a table size of about 8 entries. This is thought to be large enough for all practical purposes.



The generated code can be described as follows:

the type guard *v(T)* becomes

```
p := Tag(v);
IF p^.ttable[L] # Tadr THEN HALT(18) END
```

and the type test *v IS T* becomes

```
p := Tag(v);
RETURN p^.ttable[L] = Tadr
```

where *L* is the extension's level, and *Tadr* the address of *T*'s type descriptor. *T* is a hidden global variable in the module which declares type *T*.

Tag(v) is different for a record referenced via pointer and for a VAR parameter. The former contains the tag in the record itself, while the latter's tag is passed as an implicit parameter, together with the address of the record.

A guard applied to a NIL valued pointer aborts the program.

The WITH statement produces the same code as a type guard, the difference is only relevant for the compiler.

A similar scheme has been presented in [1].

Hidden type guards are generated for the assignment of one record to another, indirectly referenced record variable. In this case the implementation must enforce strict equality of the record types on both sides of the assignment. This leads to a simpler type guard, namely:

```
p := Tag(v);
IF p # Tadr THEN HALT(19) ENDß
```

References

1. Cohen N H (1989),
   Type–Extension Type Tests Can Be Performed in Constant Time,
   IBM Research Report
2. Wirth N (1988),
   Type Extensions,
   ACM Trans. on Programming Languages and Systems, Vol. 10, No. 2, 204–214

4. A Symmetric Solution to the Load/Store Problem

J. Templ 1.3.91

The problem of loading and storing polymorphic data structures from or to files using load and store messages is usually considered to be asymmetric because an object existing in memory can receive a store message but an object existing on a file cannot receive a load message. Nevertheless a symmetric solution for the problem is proposed. It is argued that a symmetric solution is both, more beautiful and more flexible.

The key to a symmetric solution is in the separation of the information associated with an object into a header and a contents part. The header contains the type information and the contents part contains the data associated with the object. As the type information is not maintained by the object itself but by the class the object belongs to, it is straight forward to use class methods (ordinary procedures) to handle the type information and to use instance methods (type bound procedures or message handlers) to handle the contents of an object. In the proposed solution all classes handle the type information the same way, therefore one can think of the type handling methods as meta class methods. When dealing with extended types, the problem of loading and storing inherited state (possibly invisible to the extended type) arises. Using inherited load and store methods (super–calls) solves the problem but there is a subtle point to observe. When the type information is handled by the object itself instead of by the (meta) class, each overriding method is forced to use super calls. Also super calls must be done before any other data is stored onto the file. The more flexible symmetric solution follows postulate 1:

"An object never stores its own type information as response to a store message"

Loading an object o using a Rider R may be done by a procedure ReadObj(R, o) that generates an object according to the header information. Then the object's data can be loaded by sending a load message, e.g. o.Load(R).

$ object = header contents.

Storing an object o using a Rider R may be done by a procedure WriteObj(R, o). Storing the contents of the object may be done by sending a store message o.Store(R).

Let T be a subtype of Object and T1 a subtype of T. Let Load and Store be procedures bound to T and Load' and Store' procedures bound to T1 overriding and invoking the inherited procedures. Storing of an object v with dynamic type T1 to a file is then done by the following steps:

1. WriteObj(R, v);
2. v.Store(R) invokes Store'
   2.1. v.Store^(R) in Store' invokes Store
      2.1.1. data associated with type T is stored
   2.2. additional data associated with type T1 is stored

Loading of an object of type T1 from a file into a variable v is done in symmetric steps:

1. ReadObj(R, v);
2. v.Load(R) invokes Load'
  2.1. v.Load^(R) in Load' invokes Load
    2.1.1. data associated with type T is loaded
  2.2. additional data associated with type T1 is loaded

ReadObj and WriteObj are responsible for internalizing and externalizing the object's type information. For internalized objects this information consists of a type tag, i.e. a pointer to a type descriptor node unique for a type. For externalized objects the type tags are mapped to reference numbers that refer to the type of the object. The first occurence of a type reference is followed by the externalized type descriptor consisting of the name of the module defining the type and the type's name.

$ header = ref [module type].
$ ref = integer.
$ module = char {char} 0X.
$ type = char {char} 0X.

For easy maintenance of the reference numbers of types, a ref field in the type descriptor is assumed. To avoid resetting this field before each "store session", a global virtual clock (store counter) is introduced and instead of the reference number in the type descriptors a time stamp is actually used. This time stamp contains the clock value of the type's last externalization. The reference number written to the file is the difference between the time stamp and the start time (clock0) of the stores which is defined by calling a Reset procedure. For "load sessions" a type table and a type counter has to be maintained which are also initialized by the same Reset procedure. For more details see the prototype implementation below. The use of the Reset procedure is restricted by postulate 2:

"The initialization of the generic load/store mechanism must be symmetric"

More accuratly, each Reset call preceding a store sequence must correspond to a Reset call preceding a load sequence and vice versa. Normally, the resets are done on the level of user activated commands.

A module Files1 is assumed to support persistent data portable across different Oberon implementations. Files1 contains procedures for loading and storing basic types (integers, reals, ...) in a portable way and it contains procedures to load and store the empty object, i.e. it handles the dynamic type information associated with an object derived from Files1.Object.

Properties of the proposed solution:

– no install mechanism required
– efficient externalization and internalization

– no additional storage in internalized objects
– compact external representation
– easy to implement
– low space overhead in type descriptors (time stamp plus type name)
– flexible in the use of super–calls
– pure Oberon (language)

A prototype of module Files1 has been implemented under SPARC–Oberon:

```
MODULE Files1;  (* J.Templ, 24.2.91 *)
  IMPORT SYSTEM, Files, Kernel, Modules;

  TYPE
    Object* = POINTER TO ObjectDesc;
    ObjectDesc* = RECORD END ;

    TDesc = POINTER TO RECORD
      m: Kernel.Module;
      name: ARRAY 24 OF CHAR;
      time: LONGINT   (* < clock *)
    END ;

  VAR
    module*, type*: ARRAY 24 OF CHAR;   (* most recent internalized type *)
    clock, noftypes: LONGINT;       (* clock0 = clock – noftypes *)
    typTab: ARRAY 256 OF LONGINT;
  ...

  PROCEDURE New(typetag: LONGINT): Object;
  ...
  END New;

  PROCEDURE ThisType(m: Kernel.Module; VAR type: ARRAY OF CHAR):
LONGINT;
  ...
  END ThisType;

  PROCEDURE Reset*;
  BEGIN noftypes := 0
  END Reset;

  PROCEDURE ReadObj* (VAR R: Files.Rider; VAR o: Object);
    VAR ref, tag: LONGINT; m: Kernel.Module;
  BEGIN
    Read(R, ref);
    IF ref = noftypes THEN
      ReadString(R, module);
```

```
      ReadString(R, type);
      m := Modules.ThisMod(module);
      IF m # NIL THEN tag := ThisType(m, type);
        IF tag # 0 THEN typTab[ref] := tag; INC(noftypes);
          o := New(typTab[ref])
        ELSE R.res := 1
        END
      ELSE R.res := 2
      END
    ELSIF ref # –1 THEN o := New(typTab[ref])
    ELSE o := NIL
    END
  END ReadObj;

  PROCEDURE WriteObj* (VAR R: Files.Rider; o: Object);
    VAR tag: TDesc; t: LONGINT;
  BEGIN
    IF o # NIL THEN
      SYSTEM.GET(SYSTEM.VAL(LONGINT, o)–4, t);
      tag := SYSTEM.VAL(TDesc, t – 36);
      IF tag.time < clock – noftypes THEN
        Write(R, noftypes);
        Files1.Write(R, noftypes);
        tag.time := clock;
        INC(noftypes); INC(clock);
        Files1.WriteString(R, tag.m.name);
        Files1.WriteString(R, tag.name)
        ELSE Write(R, tag.ref)
      ELSE Files1.Write(R, tag.time – (clock – noftypes))
      END
    ELSE Write(R, –1)
    END
  END WriteObj;

BEGIN clock := 1; noftypes := 0
END Files1.
```

Example: loading and storing a binary tree using type bound procedures.

```
TYPE
  Tree = POINTER TO TreeDesc;
  TreeDesc = RECORD
    (Files1.ObjectDesc)
    left, right: Tree
  END

PROCEDURE (t: Tree) Load (VAR R: Files.Rider);
```

```
BEGIN
   Files1.ReadObj(R, t.left);
   IF t.left # NIL THEN t.left.Load(R) END ;
   Files1.ReadObj(R, t.right);
   IF t.right # NIL THEN t.right.Load(R) END ;
END Load;

PROCEDURE (t: Tree) Store (VAR R: Files.Rider);
BEGIN
   Files1.WriteObj(R, t.left);
   IF t.left # NIL THEN t.left.Store(R) END ;
   Files1.WriteObj(R, t.right);
   IF t.right # NIL THEN t.right.Store(R) END ;
END Store;

PROCEDURE StoreCmd*;
   ...
   Files1.Reset;
   Files1.WriteObj(R, t);
   IF t # NIL THEN t.Store(R) END
END StoreCmd;

PROCEDURE LoadCmd*;
   ...
   Files1.Reset;
   Files1.ReadObj(R, t);
   IF t # NIL THEN t.Load(R) END
END LoadCmd;
```

Note that an asymmetric solution which stores the type information of an object as response to the store message is not significantly shorter because NIL pointers have to be handled explicitly. It also has the disadvantage that the external representation of NIL has to be known (unless a special procedure that stores the value NIL has been introduced).

```
PROCEDURE StoreCmd*;    (* the asymetric solution *)
   ...
   Files1.Reset;
   IF t # NIL THEN t.Store(R)
   ELSE Files1.Write(R, 0)
   END
END StoreCmd;
```

The following presents the complete interface of module Files1 together with a short description of the external data representation.

```
DEFINITION Files1;    (*J.Templ 31.1.91*)
```

```
   (* module to support portable persistent data.
   ReadInt, WriteInt: 2 Byte integers, little endian byte ordering
   ReadLInt, WriteLInt: 4 Byte integers, little endian byte ordering
   ReadSet, WriteSet: 4 byte sets, little endian byte ordering, ORD({0}) = 1
   ReadReal, WriteReal: 4 byte IEEE reals, little endian byte ordering
   ReadLReal, WriteLReal: 8 byte IEEE reals, little endian byte ordering
   ReadString, WriteString: arbitrary length, null terminated
   Read, Write: compact integers, 1 to 5 byte, cf. ETH Report 133, 1990 *)

   IMPORT Files;

   TYPE
      Object = POINTER TO ObjectDesc;
      ObjectDesc = RECORD END ;

   VAR module, type: ARRAY 24 OF CHAR;

   PROCEDURE Read (VAR R: Files.Rider; VAR i: LONGINT);
   PROCEDURE ReadInt (VAR R: Files.Rider; VAR i: INTEGER);
   PROCEDURE ReadLInt (VAR R: Files.Rider; VAR i: LONGINT);
   PROCEDURE ReadLReal (VAR R: Files.Rider; VAR r: LONGREAL);
   PROCEDURE ReadReal (VAR R: Files.Rider; VAR r: REAL);
   PROCEDURE ReadSet (VAR R: Files.Rider; VAR s: SET);
   PROCEDURE ReadString (VAR R: Files.Rider; VAR s: ARRAY OF CHAR);
   PROCEDURE ReadObj (VAR R: Files.Rider; VAR o: Object);

   PROCEDURE Write (VAR R: Files.Rider; i: LONGINT);
   PROCEDURE WriteInt (VAR R: Files.Rider; i: INTEGER);
   PROCEDURE WriteLInt (VAR R: Files.Rider; i: LONGINT);
   PROCEDURE WriteLReal (VAR R: Files.Rider; r: LONGREAL);
   PROCEDURE WriteReal (VAR R: Files.Rider; r: REAL);
   PROCEDURE WriteSet (VAR R: Files.Rider; s: SET);
   PROCEDURE WriteString (VAR R: Files.Rider; VAR s: ARRAY OF CHAR);
   PROCEDURE WriteObj (VAR R: Files.Rider; o: Object);

   PROCEDURE Reset;

END Files1.
```

5. Garbage Collection on Open Arrays

J. Templ 3.3.91

Traditional garbage collectors for Oberon ignore the problem of traversing array structures on the heap by assuming that the compiler forbids such constructs (implementation restriction). However, there are good reasons for supporting pointers to arrays (fixed size and open arrays) with arbitrary element types and at the same time eliminating the implementation restriction. This paper proposes a solution for traversing array data structures that affects the inner loop of the garbage collector's mark phase in the common case of traditional record nodes only by two simple assignments when following a pointer (two register moves on most processors). The outer loop needs two additional bit tests (within registers). Although the mark phase can be formulated without nested loops, one can think of the highly time critical operations performed on each pointer of a node (skipping nil pointers, skipping pointers that point to marked blocks, and following a pointer) as the "inner loop". The "outer loop" contains all operations performed on each block, i.e. the inner loop and some operations when leaving the node. In other words, the inner loop is executed O(N) times, where N is the number of reachable pointers, and the outer loop is executed O(M) times, where M is the number of reachable blocks. It is obvious that N >= M and in some cases N >> M.

Let T, Elem and P be types as defined below and v be a variable of type P.

```
  TYPE
    T = ARRAY n0, n1 .. ni–1 OF Elem;
    Elem = RECORD ... END ;
    P = POINTER TO T;
```

The proposed algorithm assumes a storage block pointed to by v to look like this:

v–4		tag	points to Elem type descriptor
v	—>	data	points to the first array element
		size	$n_0 * n_1 * .. * n_{i-1}$ * SIZE(Elem)
		arrpos	reseved for the garbage collector

The type descriptor of an array block is the type descriptor of the element type which must be a record. Multi–dimensional arrays are "flattened", i.e. they are treated like big one–dimensional arrays. Arrays within records are not affected, i.e. they are still expanded in the type descriptor of the record. In addition to the existing block kinds SysBlk (a block allocated with SYSTEM.NEW) and RecordBlk (a block allocated with NEW), a new kind ArrayBlk is defined, i.e. now there are three different kinds of heap blocks. The mark phase of a garbage collector supporting only SysBlk and RecordBlk looks like the following procedure Mark that traverses all nodes reachable from the node pointed to by q. Mark expects the parameter q to point to a marked record block.

```
PROCEDURE Mark(q: Pointer);
  VAR n: Pointer;
BEGIN
```

```
  q.cnt := 0;
  LOOP
    IF Traversed(q) THEN Reset(q);
      IF StackEmpty THEN EXIT END ;
      Pop(q)
    ELSE
      Pointer(q, n);
      IF (n # NIL) & Unmarked(n) THEN SetMark(n);
        IF RecordBlk(n) THEN Push(q); q := n; q.cnt := –1 END
      END
    END ;
    INC(q.cnt)
  END
END Mark;
```

The meaning of the macros is as follows (some of them will be used later):

RecordBlk(q), ArrayBlk(q), SysBlk(q), Unmarked(q)
  check a particular bit combination usually encoded in the type tag of q
SetMark(q)
  set a bit combination in the type tag of q to signal that q is reachable
Traversed(q)
  true iff all nodes reachable from q are marked.
  offset := Offset(q, q.cnt);
  RETURN offset < 0
Reset(q)
  resets some information temporarily encoded in the type tag of q
StackEmpty
  true if stack of partially traversed nodes is empty
Pointer(q, n)
  set n to the next son of q to be traversed.
  offset := q.tag.PtrTab[q.cnt];
  n := mem[q+offset]
Push(q)
  Push q onto the stack of partially traversed nodes.
  offset := Offset(q, q.cnt);
  mem[q+offset] := tos; tos := q
Pop(q)
  Pop q from the stack of partially traversed nodes.
  offset := Offset(tos, tos.cnt);
  n := mem[tos+offset]; mem[tos+offset] := q; q := tos; tos := n
Offset(q, n)
  the offset of the n–th pointer in block q
Elemsize(q)
  the size of one array element.
  Elemsize(q) should be expandable to q.tag.size, i.e. the size of the element type
  should be available in the type descriptor. Most Oberon implementations

currently round the record size available in the type descriptor to the next power of two or to the next number divisible by 16 or the like. In this case, the element size must be included in the array block needing some additional space.

q.cnt
the number of sons of q that are already traversed

To include array blocks, we apply the mark algorithm iteratively to all array elements in the same way as it is done for records. For array blocks q.arrpos is used to hold the offset of the current array element, i.e. q.arrpos DIV Elemsize(q) holds the number of fully traversed array elements. q.cnt gives the number of sons of the element pointed to by q.arrpos that are already traversed. Mark now expects the parameter q to point to a marked record or array block.

```
PROCEDURE Mark(q: Pointer);
  VAR n: Pointer;
BEGIN
  IF ArrayBlk(q) THEN q.arrpos := 0 END ;
  q.cnt := 0;
  LOOP
    IF Traversed(q) THEN Reset(q);
      IF ArrayBlk(q) & (q.arrpos + Elemsize(q) # q.size) THEN
        INC(q.arrpos, Elemsize(q)); q.cnt := –1
      ELSIF StackEmpty THEN EXIT
      ELSE Pop(q)
      END
    ELSE
      Pointer(q, n);
      IF (n# NIL) & Unmarked(n) THEN SetMark(n);
        IF RecordBlk(n) OR ~SysBlk(n) THEN Push(q); q := n; q.cnt := –1 END
      END
    END ;
    INC(q.cnt)
  END
END Mark;
```

Unfortunately every macro that accesses a pointer in q needs to distinguish between Record and Array blocks now. For the Pointer macro the situation is like this:

```
Pointer(q, n) =
  offset := Offset(q, q.cnt);
  IF ArrayBlk(q) THEN n := mem[q.data + q.arrpos + offset]
  ELSE n := mem[q + offset]
  END
```

To avoid this distinction, we introduce an auxiliary variable t and an auxiliary invariant:
H(t, q): <=> (RecordBlk(q) & t = q) OR (ArrayBlk(q) & t = q.data + q.arrpos)

```
Pointer(q, t, n) =
```

```
   offset := Offset(q, q.cnt);
   n := mem[t + offset]
Push(q, t) =
   offset := Offset(q, q.cnt);
   mem[t + offset] := tos; tos := q
Pop(q, t) =
   offset := Offset(tos, tos.cnt);
   IF ArrayBlk(tos) THEN t := tos.data + tos.arrpos ELSE t := tos END ;
   n := mem[t + offset]; mem[t + offset] := q; q := tos; tos := n
```

t must be set appropriately when entering a node, but it remains unchanged while looping over nil pointers, marked pointers, or sysblks within a node. The overhead when pushing a record node is only a single assignment (shown in italics), the overhead for pushing an array node is one additional test and two assignments. The overhead for returning from a record node is two bit–tests and one assignment (shown in italics), for finishing array elements, another test for detecting the end of the array is needed.

```
PROCEDURE Mark(q: Pointer);
   VAR n, t: Pointer;
BEGIN
   IF ArrayBlk(q) THEN q.arrpos := 0; t := q.data ELSE t := q END ;
   q.cnt := 0;
   LOOP {H}
      IF Traversed(q) THEN Reset(q);
         IF ArrayBlk(q) & (q.arrpos + Elemsize(q) # q.size) THEN
            INC(q.arrpos, Elemsize(q)); q.cnt := –1; INC(t, Elemsize(q))
         ELSIF StackEmpty THEN EXIT
         ELSE Pop(q, t)
         END
      ELSE
         Pointer(q, t, n);
         IF (n # NIL) & Unmarked(n) THEN SetMark(n);
            IF RecordBlk(n) THEN Push(q, t); q := n; t := q; q.cnt := –1
            ELSIF ~SysBlk(n) THEN Push(q, t); q := n; q.arrpos := 0; t := q.data; q.cnt := –1
            END
         END
      END ;
      INC(q.cnt)
   END
END Mark;
```

The modest additional complexity of the solution and the small runtime overhead for record blocks (shown in italics) seem to be justified by the additional functionality.

In practice it turns out that a sophisticated encoding of the predicates ArrayBlk(q), SysBlk(q), RecordBlk(q), and Unmarked(q) has to be used to get a fast collector. The encoding used in a prototype implementation in SPARC–Oberon is explained below using an Oberon–like notation with relaxed typing rules, e.g. q.tag is sometimes used as a set, sometimes as a pointer, and sometimes as an integer. To avoid confusion, set operators are indexed with s. It is assumed that sets, pointers and integers have 4 bytes, and that bit i corresponds to value 2^i.

Predicate	Encoding	Invariant
Unmarked(q)	~(0 IN q.tag)	
RecordBlk(q)	~(1 IN q.tag)	RecordBlk(q) => ~SysBlk(q)
ArrayBlk(q)	1 IN q.tag	
SysBlk(q)	2 IN q.tag	SysBlk(q) => ArrayBlk(q)
free(q)	3 IN q.tag	

These encodings follow the rule that an allocated unmarked record block should not have any auxiliary bits set in the type tag, i.e. it should have a valid type tag without masking some bits first. This is important for fast type guards and type tests during program execution. For counting the number of sons already visited, the technique proposed by B. Heeb is used. Therefore the type tag is used as a pointer into the offset table, too. The pointer offsets are 4 byte integers, i.e. only the low order two bits of the tag can be used for Unmarked and RecordBlk. Fortunately, SysBlk is never used when traversing a node, as system blocks are supposed to have no pointers at all. free(q) is used by the scan phase of the collector. Using 4 bits in the type tag forces type descriptors to a 16 byte alignment. Note that the type tag can only be used as a pointer after masking the low order bits. In the following mem[p] means the 4 byte word at memory location p.

```
PROCEDURE Mark(q: Pointer);
  VAR n, t, tos: Pointer; offset: LONGINT;
BEGIN
  IF ArrayBlk(q) THEN q.arrpos := 0; t := q.data ELSE t := q END ;
  INC(q.tag, PtrTabOffset); tos := NIL;
  LOOP {H}
    offset := mem[q.tag –s {0, 1}];
    IF offset < 0 THEN INC(q.tag, offset);
      IF ArrayBlk(q) & (q.arrpos + Elemsize(q) # q.size) THEN
        INC(q.arrpos, Elemsize(q)); INC(q.tag, PtrTabOffset – 4); INC(t, Elemsize(q))
      ELSIF tos = NIL THEN EXIT
      ELSE Pop(q, t)
      END
    ELSE
      n := mem[t + offset];
      IF (n # NIL) & Unmarked(n) THEN INCL(n.tag, 0);
        IF RecordBlk(n) THEN Push(q, t); q := n; t := q; INC(q.tag, PtrTabOffset – 4)
        ELSIF ~SysBlk(n) THEN Push(q, t); q := n; q.arrpos := 0;
          t := q.data; INC(q.tag, PtrTabOffset – 4)
        END
      END
```

```
    END ;
    INC(q.tag, 4)
  END
END Mark;
```

The macros Push and Pop are now defined as:

```
Push(q, t) =
  mem[t + offset] := tos; tos := q
Pop(q, t) =
  offset := mem[tos.tag –s {0, 1}];
  IF ArrayBlk(tos) THEN t := tos.data + tos.arrpos ELSE t := tos END ;
  r := mem[t + offset]; mem[t + offset] := q; q := tos; tos := r
```

This procedure may be implemented using assembly language or "pseudo Oberon" (Oberon + module SYSTEM). However, there are still some improvements possible. The repeated access to mem[q.tag – {0, 1}] in the inner loop can be accelerated by introducing an auxiliary variable tag initialized with q.tag – {0, 1}. In this case, q.tag has to be updated whenever a node is left. This update operation and the bit tests before Pop may be optimized by introducing another variable qtag with qtag = q.tag * {0,1}. There are also some common subexpressions that could be eliminated (by a compiler).

```
PROCEDURE Mark(q: Pointer);
  VAR n, t, tos: Pointer; offset, tag: LONGINT; qmask, ntag: SET;
BEGIN
  IF 1 IN q.tag THEN q.arrpos := 0; t := q.data; qmask := {0, 1} ELSE t := q; qmask := {0}
  tag := q.tag  –s {0, 1} + PtrTabOffset; tos := NIL;
  LOOP {H}
    offset := mem[tag];
    IF offset < 0 THEN q.tag := tag + offset +s qmask;
      IF 1 IN qmask & (q.arrpos + Elemsize(q) # q.size) THEN
        INC(q.arrpos, Elemsize(q)); INC(tag, offset + PtrTabOffset – 4); INC(t, Elemsize(q
      ELSIF tos = NIL THEN EXIT
      ELSE qmask := tos.tag; tag := qmask –s {0, 1}; qmask := qmask *s {0, 1};
        IF 1 IN qmask THEN t := tos.data + tos.arrpos ELSE t := tos END ;
        offset := mem[tag]; n := mem[t + offset]; mem[t + offset] := q; q := tos; tos := n
      END
    ELSE
      n := mem[t + offset];
      IF (n # NIL) THEN ntag := n.tag;
        IF ~(0 IN ntag) THEN q.tag := tag +s qmask; n.tag := ntag + {0};
          IF ~(1 IN ntag) THEN mem[t + offset] := tos; tos := q;
            q := n; t := q; tag := ntag + PtrTabOffset – 4; qmask := {0}
          ELSIF ~(2 IN ntag) THEN mem[t + offset] := tos; tos := q; q := n;
             q.arrpos := 0; t := q.data; tag := ntag –s {1} + PtrTabOffset – 4; qmask := {0,
          END
        END
```

```
      END
    END ;
    INC(tag, 4)
  END
END Mark;
```

The overhead for record nodes is shown in italics. On modern processors it consists of about two machine cycles when following a pointer to a record node and six cycles when leaving a record node. Switching from one array element to the next needs one additional compare, two storage reads, one storage write, and three additions (15 – 20 cycles).