Metaprogramming Facilities in Oberon for Windows and Power Macintosh

Christoph Steindl and Hanspeter Mössenböck

Report 8
July 1996

Authors' address

Institut für Informatik (Systemsoftware)
Johannes Kepler Universität Linz
A-4040 Linz / Auhof

e-mail:
steindl@ssw.uni-linz.ac.at
moessenboeck@ssw.uni-linz.ac.at

Abstract

This report describes metaprogramming facilities in the Oberon V4 system for Power Macintosh and Windows. Metaprogramming means that a module can access the structure of other modules (i.e., procedures, types, run-time data) at run time.

The purpose of this report is threefold:
a) It explains the use and the internals of some existing but hardly known modules of the Oberon V4 system (i.e., *Modules* and *Types*).
b) It introduces a new module *Ref* for exploring the structure of types, procedures, stack frames, global data and heap data as well as for accessing the contents of arbitrary variables at run time.
c) It explains the layout of module descriptors, type descriptors, dynamic array descriptors, and stack frames in the Oberon implementations for Windows and Power Macintosh.

We show how metaprogramming can be used in a number of interesting applications such as a post mortem debugger that allows the user to zoom into records, arrays and pointers, a heap inspector, a database interface, and a general output module.

Contents

1. Metaprogramming

In programs we distinguish between the *data level* and the *program level*. Variables are at the data level and can be accessed by the statements of a program. Modules, types and procedures are at the program level. They serve to structure a program but they are usually not viewed as data. Sometimes, however, programs want to inspect the components of other programs at the program level, for example, in order to answer the following questions:

- What are the field names of a record type *T* declared in module *M*?
- Which procedures are currently active (e.g., when a run-time trap occurs)? What are the names, types and values of their variables?
- Does the caller of the currently executing procedure have a variable named "x", and if so, what is its type and value?

Questions like these are considered to be on a *meta level*. They treat modules, types, and procedures as data. They have to know the structure of this "data" in order to access (or even modify) their contents. If a programming system supports questions of this kind we call it a *metaprogramming system*. If programs can ask these question also about themselves we call such a system *reflective*.

Metaprogramming and reflection were pioneered by the languages *Lisp* ([McCar60], [Smi82]) and *Smalltalk* ([GR83]). In Lisp all programs are treated as data. It is possible to inspect their structure and even to dynamically build new programs (higher order functions) that can be executed. In Smalltalk types are represented as classes and procedures as methods of these classes. The structure of a class is described in a metaclass of which the class is an instance. The metaclass information can again be accessed and even modified. Many other languages allow metaprogramming in a similar way (e.g., Self [US87], CLOS [Att89], or Beta [MMN93]).

The original Oberon system [WiGu89] offered only a limited degree of metaprogramming. It provided a module *Modules* which allowed programmers - among other things - to inspect information about all loaded modules. Later a module *Types* was added, which provided basic information about record types. However, *Types* was not documented in the books about Oberon ([Rei91], [WiGu92]). In his dissertation ([Tem94]), J. Templ implemented an experimental version of Oberon for Sun workstations, which treated modules, procedures, and record types as data allowing full access to their components.

This report summarizes what is available in *Modules* and *Types*. It also describes a reimplementation of a subset of Templ's metaprogramming system in the form of a module *Ref* which is available for Windows Oberon and for PowerMac Oberon.

2. Module Modules

The Oberon System [WiGu89] provides dynamic linking and loading of modules. When the execution of a command *M.P* is requested, module *M* is loaded, unless it has already been loaded due to an earlier execution of a command from *M,* or because *M* was imported by another already loaded module. A newly loaded module has to be linked to the already loaded modules. In Oberon, linking is not an extra step but is done by the loader which is therefore called a linking loader.

Module *Modules* is the implementation of this loader. It maintains a list of loaded modules, which can be traversed to obtain information about a particular module. It also provides procedures to load and unload modules on demand.

For historical reasons the interface of *Modules* is not quite the same in Windows Oberon and PowerMac Oberon. Under Windows Oberon some of the following types and variables are defined in module *Kernel*. Nevertheless the interface of *Modules* should conceptionally look like this:

```
DEFINITION Modules;

   CONST (* error codes *)
      done = 0; fileNotFound = 1; invalidObjFile = 2; keyMismatch = 3;
      notEnoughMemory = 4; modNotFound = 5; cmdNotFound = 6;
      refCntNotZero = 7; cyclicImport = 8; corruptedObjFile = 9;

   TYPE
      ModuleName = ARRAY 32 OF CHAR;
      Command = PROCEDURE;
      Module = POINTER TO ModuleDesc;
      ModuleDesc = RECORD  (* descriptor of loaded modules *)
         next-: ModuleName;
         name-: Name;           (* module name *)
         refcnt-: LONGINT;      (* reference count: number of importing modules *)
         ...                    (* further fields (platform dependent) *)
      END ;

   VAR
      modules-: Module;            (* head of module list *)
      res: INTEGER;                (* result code of operations below *)
      importing: ModuleName;  (* name of importing module if res # done *)
      imported: ModuleName;   (* name of imported module if res = keyMismatch *)

   PROCEDURE Free (name: ARRAY OF CHAR; all: BOOLEAN);
   PROCEDURE ThisCommand (m: Module; name: ARRAY OF CHAR): Command;
   PROCEDURE ThisMod (name: ARRAY OF CHAR): Module;

END Modules.
```

State

- *modules* is the anchor of the list of loaded modules.
- *res* is the error code of the last operation and can be one of the values listed in the constant declarations.
- *importing* and *imported* are defined if *res # done*. For *res = keyMismatch*, they denote the importing and the imported modules, respectively. For other error codes, only *importing* is defined and denotes the offending module.

Operations

- *m := ThisMod(name)* returns a pointer to the descriptor of module *name*. If this module is not yet loaded, it will be dynamically loaded.
- *cmd := ThisCommand(m, name)* returns the command *name* of module *m*. Commands are parameterless exported procedures which can be invoked interactively by the user.
- *Free(name, all)* tries to unload module *name*. A module can only be unloaded if its reference count is zero, i.e., if it is not imported by any other loaded module. If *all* is TRUE, all modules which are (transitively) imported by *name* will be unloaded, if their reference count is (or becomes) zero.

Differences between Windows Oberon and PowerMac Oberon

The differences in the command descriptors and module descriptors are summarized in Table 1. A module descriptor stores information about the sections of a module (code, data, run-time information, etc.). Under Windows Oberon these sections are implemented as dynamic arrays. Under PowerMac Oberon the module descriptor stores the addresses and the sizes of the sections. For example, *entries* contains the address of the entries section and *nofentries* contains the number of entries. For every loaded module there are the following sections (see also Fig. 2 in Appendix A.1):

- *entries*: under Windows Oberon, an array with the *addresses* of all exported procedures; under PowerMac Oberon, an array with the *offsets* of all exported procedures relative to the beginning of the module's code section.
- *varEntries*: under Windows Oberon, an array containing the addresses of all exported variables. Under PowerMac Oberon exported variables are not stored as separate entries.
- *cmds* (commands): an array with the names and addresses (Windows) or offsets (PowerMac) of all commands.
- *ptrTab* (pointers): an array with the addresses (Windows) or offsets (PowerMac) of all pointers in the global variable section. These pointers are used as root pointers by the garbage collector.
- *tdescs* (typedescs): an array with the addresses of the type descriptors for all record types declared in this module.
- *imports*: an array of pointers to the module descriptors of imported modules. *imports[0]* refers to the descriptor of the module itself.

Module Kernel under Windows Oberon	**Module Modules under PowerMac Oberon**
Name = ARRAY 32 OF CHAR;	**CommandName** = ARRAY 24 OF CHAR;
	CommandPtr = POINTER TO CommandDesc;
Cmd = RECORD	**CommandDesc** = RECORD
name: Name;	**name**-: CommandName;
adr: LONGINT	**offset**-: INTEGER
END ;	END ;
	ModuleName = ARRAY 32 OF CHAR;
TerminationHandler = PROCEDURE;	**Module** = POINTER TO ModuleDescriptor;
Module = POINTER TO ModuleDesc;	**ModuleDescriptor** = RECORD
ModuleDesc = RECORD	**link**-: Module;
next: Module;	**name**-: ModuleName;
name: Name;	**refcnt**-: INTEGER;
refcnt: LONGINT;	**key**-, **SB-**: LONGINT;
key, sb: LONGINT;	**consize-, codesize-, nofentries-,**
varEntries: POINTER TO ARRAY OF LONGINT;	**nofcmds-, entries-**: LONGINT;
entries: POINTER TO ARRAY OF LONGINT;	**commands-**: LONGINT;
cmds: POINTER TO ARRAY OF Cmd;	**pointers-**: LONGINT;
ptrTab: POINTER TO ARRAY OF LONGINT;	**typedescs-**: LONGINT;
tdescs: POINTER TO ARRAY OF LONGINT;	**imports-**: LONGINT;
imports: POINTER TO ARRAY OF LONGINT;	**block-, PC-**: LONGINT;
data, code: POINTER TO ARRAY OF CHAR;	**refs**-: LONGINT;
refs: POINTER TO ARRAY OF CHAR;	**datasize-, blocksize-, refsize**-: LONGINT;
init: BOOLEAN;	**nofimps-, noftds-, nofptrs-,**
term: TerminationHandler	**noftraps**-: INTEGER;
END ;	**traps-**: LONGINT
	END ;
VAR **modules**: Module;	VAR **modules**: Module;

Table 1. Module descriptors under Windows Oberon and PowerMac Oberon

- *data*: the constants and global variables of the module. Under PowerMac Oberon this section starts at address *block*. *SB* (static base) denotes the separation line between constants and global data, i.e., *SB* is the address of the first global variable of the module.
- *code* (PC): the code of this module.
- *refs*: the reference information of this module (see Section 4).

Under Windows Oberon there are additional types and procedures for object finalization [Tem94]. An object can be registered for finalization, which means that it is notified prior to being deallocated by the garbage collector. This can be used to close files or network connections. Furthermore it is possible to register a procedure that shall be called when the module is unloaded.

- *RegisterObject(obj, finalizer)* registers procedure *finalizer* to be called before the object *obj* is reclaimed by the garbage collector.
- *InstallTermHandler(term)* installs the procedure *term* to be called when the module containing this call of *InstallTermHandler* is unloaded.

Under PowerMac Oberon, there are additional procedures for maintaining shared libraries (dynamic link libraries, DLLs):

```
VAR
  ThisLib-: PROCEDURE (name: ARRAY OF CHAR; ppc: BOOLEAN): LONGINT;
  ThisSym-: PROCEDURE (conn: LONGINT; name: ARRAY OF CHAR): LONGINT;
```

- *id := ThisLib(n, ppc)* loads the shared library with the name *n* and returns its identifier *id*. The parameter *ppc* specifies if the instruction set architecture of the library is the PowerPC architecture or the 680x0 architecture. *ppc* should normally be TRUE.
- *adr := ThisSym(lib, n)* looks up the symbol with the name *n* in the shared library *lib* that was loaded by *ThisLib*. If found, the address of the symbol is returned. *ThisSym* can be used to load the address of an operating system procedure into an Oberon procedure variable:

  ```
  VAR
    NewPtr: PROCEDURE (size: LONGINT): LONGINT;
    lib, adr: LONGINT;
  ...
  lib := Modules.ThisLib("InterfaceLib", TRUE);
  adr := Modules.ThisSym(lib, "NewPtr");
  SYSTEM.GET(adr, NewPtr)
  ```

  The same result can be obtained in a simpler way by calling

  ```
  Sys.Assign("NewPtr", SYSTEM.ADR(NewPtr))
  ```

Under Windows Oberon dynamic link libraries can be loaded using module *Kernel*:

```
PROCEDURE LoadLibrary (file: ARRAY OF CHAR): LONGINT;
PROCEDURE GetAdr (lib: LONGINT; sym: ARRAY OF CHAR; VAR adr: LONGINT);
```

- *lib := LoadLibrary(n)* loads the dynamic link library with the name *n* and returns its identifier in *lib*.
- *GetAdr(lib, n, adr)* looks up the symbol with the name *n* in the dynamic link library *lib* that was loaded by *LoadLibrary*. If found, the address of the symbol is returned in *adr*. *GetAdr* can be used to load the address of a DLL function into an Oberon procedure variable. Note, that for some Windows API functions there exist two versions, one that accepts Unicode parameters and one for ASCII parameters. The former have a "W" appended to their name, the latter an "A" (hence "FindFirstFileA" and not "FindFirstFile" in the following example).

  ```
  VAR
    FindFirstFile: PROCEDURE (filename: LONGINT; data: LONGINT): LONGINT;
    lib: LONGINT;
  ...
  lib := Kernel.LoadLibrary("Kernel32");
  Kernel.GetAdr(mod, "FindFirstFileA", SYSTEM.VAL(LONGINT, FindFirstFile));
  ```

Examples

In order to find the module that contains a certain program counter value, one can traverse the list of modules. As this involves some platform-dependent fields of the module descriptors, the implementation varies between Windows and Power Macintosh. Therefore it is a good idea to use the procedure *Ref.OpenProc(pc, r)* (see Section 4) to find the procedure that contains the program counter *pc*.

```
m := Kernel.modules;   (* code for Windows Oberon *)
WHILE (m # NIL) & ((SYSTEM.ADR(m.code^) > pc)
OR (pc > SYSTEM.ADR(m.code^) + LEN(m.code^))) DO
  m := m.next
END ;

m := Modules.modules;   (* code for PowerMac Oberon *)
WHILE (m # NIL) & ((pc < m.PC) OR (pc > m.PC + m.codesize*4)) DO
  m := m.link
END ;
```

In order to call a command *P* from module *M* one can use the following code:

```
VAR m: Modules.Module; p: Modules.Command;
...
m := Modules.ThisMod("M");
p := Modules.ThisCommand(m, "P");
p   (* call M.P *)
```

3. Module Types

Module *Types* provides information about record types. This information is obtained from the type descriptors of which there is one for every record type (see Appendix A.2). More detailed information about records can be obtained using module *Ref* (see Section 4).

Interface

```
DEFINITION Types;
  IMPORT SYSTEM, Modules;

  TYPE
    Type = POINTER TO TypeDesc;
    TypeDesc = RECORD
      name: ARRAY 32 OF CHAR;   (* name of the (record) type *)
      module: Modules.Module   (* module in which the type is declared *)
    END ;

  PROCEDURE TypeOf (o: SYSTEM.PTR): Type;
  PROCEDURE This (mod: Modules.Module; name: ARRAY OF CHAR): Type;
  PROCEDURE NewObj (VAR o: SYSTEM.PTR; t: Type);
  PROCEDURE LevelOf (t: Type): INTEGER;
  PROCEDURE BaseOf (t: Type; level: INTEGER): Type;
END Types.
```

- *t := TypeOf(o)* returns the dynamic type of the record pointed to by *o*. The pointer *o* must not be NIL.
- *t := This(mod, name)* returns the *Type* of the record type *name* declared in module *mod*, or NIL, if this type does not exist.
- *NewObj(o, t)* allocates a record of type *t* on the heap and returns a pointer to that record in *o*. If the actual parameter corresponding to *o* is declared as POINTER TO *T*, *t* must be equal to *T* or an extension of it.
- *lev := LevelOf(t)* returns the extension level of the specified record type. Record types that do not extend other types have extension level 0. If *T1* directly extends *T0*, the level of *T1* is the level of *T0* plus 1.
- *t0 := BaseOf(t1, lev)* returns the base type of *t1* that has extension level *lev.*

Examples

The most frequently used operations of *Types* are *TypeOf*, *This* and *NewObj*. They can be used to convert a type to a type name and vice versa. This is necessary, for example, when objects of arbitrary types have to be written to a file and read in again. During reading the objects have to be allocated with their correct types. In order to be able to do that, the type name of every object must occur in front of the object's data. This type name can be used to allocate an object of the appropriate type. The following two procedures ([Tem94]) write and read arbitrary objects whose type is derived from a type *Object*. We assume that type *Object* supports a *Load* and a *Store* message to read and write the object's data:

```
PROCEDURE WriteObj (VAR r: Files.Rider; obj: Object);
  VAR t: Types.Type;
BEGIN
  IF obj = NIL THEN WriteString(r, "")
  ELSE t := Types.TypeOf(obj);
    WriteString(r, t.module.name); WriteString(r, t.name); obj.Store(r)
  END
END WriteObj;

PROCEDURE ReadObj (VAR r: Files.Rider; VAR obj: Object);
  VAR name: ARRAY 32 OF CHAR; m: Modules.Module; t: Types.Type;
BEGIN
  ReadString(r, name);
  IF name = "" THEN obj := NIL
  ELSE m := Modules.ThisMod(name);
    ReadString(r, name); t := Types.This(m, name);
    Types.NewObj(t, obj); obj.Load(r)
  END
END ReadObj;
```

Module *Types* can also be used to perform some other interesting operations that are not available in the Oberon language. Let's assume the following declarations:

```
TYPE
  Object = POINTER TO ObjectDesc;
  ObjectDesc = RECORD END;
VAR
  x, y: Object;
  t, t1: Types.Type;
  lev, lev1: INTEGER;
```

Cloning an object x. Assume that we do not know the dynamic type of *x*. We want to have a clone *y* with the same dynamic type and the same contents as *x*. We cannot use NEW(*y*) because that would set the dynamic type of *y* to *Object*. The following code does the job:

```
Types.NewObj(y, Types.TypeOf(x));
x.CopyFieldsTo(y)
```

Does the variable x have the dynamic type T? The type test *x IS T* finds out if the dynamic type of *x* is at least *T*. However, the test also returns TRUE if the dynamic type is an extension of *T*. If we want to know whether the dynamic type is exactly *M.T* we can use the following code:

```
t := Types.TypeOf(x);
IF (t.module.name = "M") & (t.name = "T") THEN ... END
```

Is the dynamic type of x equal to the static type of x? This test may be necessary if an operation wants to treat extended objects in a different way than original objects.

```
IF Types.LevelOf(Types.TypeOf(x)) = 0 THEN ... END
```

Are the dynamic types of x and y derived from the same base type?

```
t := Types.TypeOf(x); lev := Types.LevelOf(t);
t1 := Types.TypeOf(y); lev1 := Types.LevelOf(t1);
IF Types.BaseOf(t, lev - 1) = Types.BaseOf(t1, lev1 - 1) THEN ... END
```

4. Module Ref

Module *Ref* can be used to obtain information about the procedures, record types, and variables of a module. For example, it is possible to access the names, types and components of these items at run time. For variables it is also possible to read and write their values.

4.1 Riders

All information is accessible via *riders*. A rider is a cursor that iterates over sequences of variables, procedures, types, or other items. The general pattern for using a rider *r* is

```
Ref.Open...(..., r);
WHILE r.mode # Ref.End DO
  ...
  r.Next
END
```

At any time the rider contains information about the item on which it is positioned. A rider can be opened on data (global variables, local variables, heap) or on a module's list of procedures or record types.

global variables	OpenVars(module, r)	sets *r* to the first global variable of the module
local variables	OpenStack(info, r)	sets *r* to the topmost stack frame
heap	OpenPtr(p, r)	sets *r* to the first record field or array element to which *p* refers
procedures	OpenProcs(module, r)	sets *r* to the first procedure of the module
record types	OpenTypes(module, r)	sets *r* to the first record type of the module

Program data is organized hierarchically, e.g., the stack is a sequence of stack frames, which are sequences of variables, which may be sequences of record fields and so on. The following grammar shows this structure.

```
Stack         = {Frame}.                  accessible via OpenStack
Frame         = {Variable}.
Variable      = simpleVar | RecordVar | ArrayVar.
RecordVar     = {Field}.
ArrayVar      = {Elem}.
Field         = Variable.
Elem          = Variable.

Globals       = {Variable}.               accessible via OpenVars

PointerBase   = RecordVar | ArrayVar.     accessible via OpenPtr

Procedure     = {Proc}.                   accessible via OpenProcs
Proc          = {Variable}.

Types         = {RecordType}.             accessible via OpenTypes
RecordType    = {Field}.
```

When a rider is positioned on a composite item it is possible to zoom into this item and iterate over its elements. For example, to iterate over the variables of the second frame on the stack (i.e., the variables of the caller of the currently active procedure) one does the following:

```
Ref.OpenStack(NIL, r);  (* r is on the frame of currently active procedure *)
r.Next;                 (* r is on the caller's frame *)
r.Zoom(r)               (* r is on the first variable of the caller's frame *)
```

When a rider was opened on data (using *OpenVars*, *OpenStack*, or *OpenPtr*) and is currently positioned on a simple variable (i.e., not a record or an array), the value of this variable can be read or modified using procedures like *r.ReadInt* or *r.WriteInt*. If a rider was opened on the list of procedures or record types there is no data to be read or written, so the read and write procedures must not be called.

4.2 Interface

```
DEFINITION Ref;

IMPORT SYSTEM, Types;

CONST
  (* item forms *)
  None = 0; Byte = 1; Bool = 2; Char = 3; SInt = 4; Int = 5; LInt = 6;
  Real = 7; LReal = 8; Set = 9; String = 10; NilTyp = 11; NoTyp = 12;
  Pointer = 13; Procedure = 14; Array = 15; Record = 16; DynArr = 17;

  (* item modes *)
  End = 0; Var = 1; VarPar = 2; Elem = 3; Fld = 4; Frame = 5; Proc = 6; Type = 7;
```

```
TYPE
  ProcVar = PROCEDURE;

  Rider = RECORD   (* see Table 2 and 3 *)
    name: ARRAY 32 OF CHAR;
    mode, form: SHORTINT;
    idx, off, len: LONGINT;
    mod: ARRAY 32 OF CHAR;
    level: SHORTINT;
    PROCEDURE (VAR r: Rider) Next;
    PROCEDURE (VAR r: Rider) Zoom (VAR sub: Rider);
    PROCEDURE (VAR r: Rider) Adr (): LONGINT;
    PROCEDURE (VAR r: Rider) Type (): Types.Type;
    PROCEDURE (VAR r: Rider) SetTo (idx: LONGINT);

    PROCEDURE (VAR r: Rider) Read (VAR ch: CHAR);
    PROCEDURE (VAR r: Rider) ReadBool (VAR b: BOOLEAN);
    PROCEDURE (VAR r: Rider) ReadInt (VAR i: INTEGER);
    PROCEDURE (VAR r: Rider) ReadLInt (VAR li: LONGINT);
    PROCEDURE (VAR r: Rider) ReadLReal (VAR lr: LONGREAL);
    PROCEDURE (VAR r: Rider) ReadProc (VAR p: ProcVar);
    PROCEDURE (VAR r: Rider) ReadPtr (VAR p: SYSTEM.PTR);
    PROCEDURE (VAR r: Rider) ReadReal (VAR x: REAL);
    PROCEDURE (VAR r: Rider) ReadSInt (VAR si: SHORTINT);
    PROCEDURE (VAR r: Rider) ReadSet (VAR s: SET);
    PROCEDURE (VAR r: Rider) ReadString (VAR str: ARRAY OF CHAR);

    PROCEDURE (VAR r: Rider) Write (ch: CHAR);
    PROCEDURE (VAR r: Rider) WriteBool (b: BOOLEAN);
    PROCEDURE (VAR r: Rider) WriteInt (i: INTEGER);
    PROCEDURE (VAR r: Rider) WriteLInt (li: LONGINT);
    PROCEDURE (VAR r: Rider) WriteLReal (lr: LONGREAL);
    PROCEDURE (VAR r: Rider) WriteProc (p: ProcVar);
    PROCEDURE (VAR r: Rider) WritePtr (p: SYSTEM.PTR);
    PROCEDURE (VAR r: Rider) WriteReal (x: REAL);
    PROCEDURE (VAR r: Rider) WriteSInt (si: SHORTINT);
    PROCEDURE (VAR r: Rider) WriteSet (s: SET);
    PROCEDURE (VAR r: Rider) WriteString (str: ARRAY OF CHAR);
  END ;

  ExceptionInfo = ...;   (* machine state: system dependent *)

  PROCEDURE OpenVars (mod: ARRAY OF CHAR; VAR r: Rider);
  PROCEDURE OpenStack (inf: ExceptionInfo; VAR r: Rider);
  PROCEDURE OpenPtr (p: SYSTEM.PTR; VAR r: Rider);
  PROCEDURE OpenProcs (mod: ARRAY OF CHAR; VAR r: Rider);
  PROCEDURE OpenTypes (mod: ARRAY OF CHAR; VAR r: Rider);
  PROCEDURE PC (mod, name: ARRAY OF CHAR): LONGINT;
  PROCEDURE OpenProc (pc: LONGINT; VAR r: Rider);
END Ref.
```

A *Rider* is a cursor that iterates over a sequence of variables, procedures, record types, and other items. Its fields show the attributes of the item on which it is currently positioned. Field *mode* indicates the item kind (Table 2).

mode	currently positioned on
Var	local variable, global variable, or value parameter
VarPar	variable parameter
Elem	array element
Fld	record field
Frame	stack frame
Proc	procedure
Type	record type

Table 2. Field *mode* of a *Rider*

The meaning of the other fields depends on the rider's *mode* (Table 3).

	Var, VarPar	Elem	Fld	Frame	Proc	Type
name	of variable	of array	of field	of procedure	of procedure	of type
mod	declaring module	---	declaring module	decl. module	decl. module	decl. module
form	of variable	of element	of field	---	---	---
off	in frame	in array	in record	in code	in code	---
idx	---	in array	---	of procedure	of procedure	---
len	---	of array	---	of procedure	of procedure	---
level	---	---	extension lev. of record	---	---	---

Table 3. Fields of a *Rider* (depending on the *mode* of the *Rider*)

If a rider is positioned on an element of an array variable, the field *name* shows the name of the array variable. Information about the module declaring an array type is not available in the reference information (see below), so that the field *mod* is undefined for array elements.

The type *ExceptionInfo* captures the machine state at the time of a trap. It is machine dependent and has a different form for Windows and for the Power Macintosh.

Operations

- *OpenVars(mod, r)* sets the rider *r* to the first global variable of module *mod*. If there are no global variables, *r.mode* = *End*. If module *mod* does not exist, *r.mode* = *End* and *r.mod* = "".
- *OpenStack(inf, r)*. If *inf* = NIL the rider *r* is set to the stack frame of the procedure that called *OpenStack*. If *inf* # NIL, it describes the machine state at the time of a run-time exception (trap). The rider *r* is set to the stack frame of the procedure in which the trap occurred.
- *OpenPtr(p, r)* sets the rider *r* to the first field of the record pointed to by *p*. If *p* = NIL then *r.mode* = *End*.
- *OpenProcs(mod, r)* sets the rider *r* to the first procedure of module *mod*. If there are no procedures, *r.mode* = *End*. If module *mod* does not exist, *r.mode* = *End* and *r.mod* = "".

- *OpenTypes(mod, r)* sets the rider *r* to the first record type of module *mod*. If there are no record types, *r.mode* = *End*. If module *mod* does not exist, *r.mode* = *End* and *r.mod* = "".
- pc := *PC(mod, name)* returns the absolute start address of the procedure *name* declared in module *mod*. If there is no such procedure, *pc* = 0.
- *OpenProc(pc, r)* sets the rider *r* to the procedure that contains the (absolute) program counter value *pc*. If such a procedure could not be found, *r.mode* = *End*, otherwise the fields of *r* are set appropriately.
- *r.Next* advances the rider *r* to the next item (variable, array element, record field, stack frame, procedure, or record type). If *r* was already positioned on the last item, *r.mode* is set to *End*.
- *r.Zoom(sub)*. If *r* is positioned on a composite item, a new rider *sub* is set to the first component of the composite according to the following table:

r.mode	*r.form*	*sub.mode*
Var, VarPar, Elem, Fld	Record, Pointer to Record	Fld
Var, VarPar, Elem, Fld	Array, DynArr, Pointer to Array or DynArr	Elem
Type	---	Fld
Proc, Frame	---	Var or VarPar

- *a* := *r.Adr()* returns the address of the current item (variable, parameter, record field, or array element).
- *t* := r.*Type()* returns the type of the current item if this item is of a record type, otherwise the result is undefined. If *r.mode* = *Type*, *r.Type* returns the description of this record type. If *r.form* = *Record*, *r.Type* returns the description of the item's record type.
- r.*SetTo(i)*. If *r* is positioned on an element of an array (*r.mode* = *Elem*), it is set to the *i*-th element of that array. If it is positioned on the fields of a record type *T* (*r.mode* = *Fld*), it is set to the first field of the *i*-th extension level of *T*.
- *r.ReadX*. If *r.mode* IN {*Var, VarPar, Fld, Elem*} and if *r* (or the rider from which it was zoomed) was opened with *OpenVars*, *OpenStack* or *OpenPtr*, the value of the current item can be read with the *ReadX* procedure that matches the *form* of the item (i.e., *r.ReadInt* if *r.form* = *Int*).
- *r.WriteX*. If *r.mode* IN {*Var, VarPar, Fld, Elem*} and if *r* (or the rider from which it was zoomed) was opened with *OpenVars*, *OpenStack* or *OpenPtr*, the value of the current item can be written with the *WriteX* procedure that matches the *form* of the item.

Examples

The following procedure *P(s)* prints the values of all variables, whose names appear in the argument string *s* (e.g., s = "x y z"). The variables are assumed to be local to the caller of *P*.

```
PROCEDURE P (s: ARRAY OF CHAR);
  VAR n: ARRAY 32 OF CHAR; r, r1: Ref.Rider;
BEGIN
  Ref.OpenStack(NIL, r); (* r is positioned on the stack frame of P *)
  r.Next;                (* r is positioned on the frame of P's caller *)
  FOR all names n in s DO
    r.Zoom(r1);          (* r1 is positioned on the first local variable of the caller *)
    WHILE (r1.mode # Ref.End) & (r1.name # n) DO r1.Next END;
    IF r1.mode # Ref.End THEN PrintValue(r1) END
  END
END P;

PROCEDURE PrintValue (VAR r: Ref.Rider);
  VAR i: INTEGER; re: REAL; ...
BEGIN
  CASE r.form OF
    Ref.Int: r.ReadInt(i); Out.Int(i, 0)
  |  Ref.Real: r.ReadReal(re); Out.Real(re, 0)
  |  ...
  END
END PrintValue;
```

The following procedure prints the names and values of all fields of a record variable *rec* globally declared in module *M*.

```
PROCEDURE P;
  VAR r: Ref.Rider;
BEGIN
  Ref.OpenVars("M", r);
  WHILE (r.mode # Ref.End) & (r.name # "rec") DO r.Next END;
  IF r.mode # Ref.End THEN r.Zoom(r);
    WHILE r.mode # Ref.End DO
      Out.String(r.name); Out.String(" = "); PrintValue(r);
      r.Next
    END
  END
END P;
```

The following procedure prints the names of all procedures of module *M* together with the names of their local variables:

```
PROCEDURE P;
  VAR r, r1: Ref.Rider;
BEGIN
  Ref.OpenProcs("M", r);
  WHILE r.mode # Ref.End DO
    Out.String(r.name); Out.Ln; (* procedure name *)
    r.Zoom(r1);
    WHILE r1.mode # Ref.End DO
      Out.String(r1.name); Out.Ln; (* variable name *)
      r1.Next
    END;
    r.Next
  END
END P;
```

The following code fragment shows how to print a string variable:

```
VAR r, r1: Ref.Rider; s: ARRAY 64 OF CHAR;
...
ASSERT((r.mode = Ref.Var) & (r.form = Ref.Array));
r.Zoom(r1); (* r1 is used for a lookahead! *)
IF r1.form = Ref.Char THEN r.ReadString(s); Out.String(s) END
```

The following code fragment prints the name of the procedure containing the address *adr*.

```
Ref.OpenProc(adr, r);
IF r.mode # Ref.End THEN Out.String(r.mod); Out.Char("."); Out.String(r.name) END
```

4.3 Reference information

At run time, the memory (global variables, stack, heap) is an unstructured sequence of bytes. In order to interpret it correctly one needs the symbol tables in which the names, types, addresses, and sizes of the variables are stored. The symbol table of a module is a data structure of the compiler and is usually discarded after a compilation. However, the compiler may also write (parts of) the symbol table to a file. This is called the *reference information* of a module. Most Oberon compilers generate reference information as an appendix to the object file of a module.

The original Oberon compiler [WiGu92] as well as the OP2 compiler [Cre90] generate only a partial reference information in which structured variables are not described. We extended OP2 according to the suggestions in [Tem94] so that the reference information contains a complete description of all procedures and record types declared in a module together with their structures. The reference information is described by the following grammar (a name with an asterisk denotes a number in variable length format [GPHT91]; a name ending with "1" denotes a one byte number; names in italic denote constants):

```
RefInfo =        {Procedure} {Record}.
Procedure =      ProcTag offset* PowerMacInfo Name {Variable}.
Record =         RecTag tdadr* Name {Field} 0X.
Variable =       (Var | VarPar) Object.
Object =         Name offset* Type.
Field =          Object.
Type =           Byte | Bool | Char | SInt | Int | LInt | Real | LReal
                 | Set | String | Nil | NoTyp
                 | Pointer Type
                 | Proc key*
                 | Array length* elemsize* Type
                 | DynArray elemsize* Type
                 | Rec mod1 tdadr*.
Name =           char {char} 0X.
PowerMacInfo =   -- only on the Power Macintosh
                 fsize* psize* ralloc* falloc* calloc* isleaf1.
```

The reference information can be regarded as the linearized and simplified symbol table of a module. There are just two different kinds of objects in the reference information: procedures (*Procedure*) and record types (*Record*). Global variables are treated as local variables of the module body, which is regarded as a procedure with the special name "$$".

Record types consist of a reference to their type descriptor, a name, and a list of record fields. Each record field consists of a name, an offset within the record, and a type. The type consists of a form and optional attributes: pointer types have a base type; procedure types have a key (a fingerprint of their parameter types to be used as a version key); arrays have a length, an element size and an element type; open arrays have an element size, and an element type; record types are described by a module number and a reference to their type descriptor. The module number is used locally within the module *m* to which the reference information belongs. A module number of 0 means *m* itself; any module number *n* > 0 means the *n*-th module imported by module *m*. How exactly to convert the pair (*mod tdadr*) into a type descriptor is system dependent. Under Windows Oberon *tdadr* is an index into the table *varEntries* of module *mod*. The referenced entry contains the address of the type descriptor address in the global data section of module *mod*. Under PowerMac Oberon *tdadr* is the offset of the type descriptor address in the global data section of module *mod*.

Procedures consist of their offset in the code section of the module (start offset under Windows; end offset under PowerMac), a name, and a list of local variables and formal parameters. Value parameters are treated like local variables (*mode = Var*), reference parameters have *mode = VarPar*. Under PowerMac Oberon there is additional information about procedures such as the frame sizes for variables and parameters (*fsize*, *psize*), the number of general purpose registers (*ralloc*), floating point registers (*falloc*), and bits in the condition code register (*calloc*) used by the procedure, as well as an indication if the procedure is a leaf procedure (*isleaf*).

When a module is loaded, its reference information is read into memory. A rider is then simply a link between the reference section and the data of a module. When the rider is advanced to the next item in the reference section it is also advanced to the next data item in memory so that the corresponding information is kept synchronized (Figure 1).

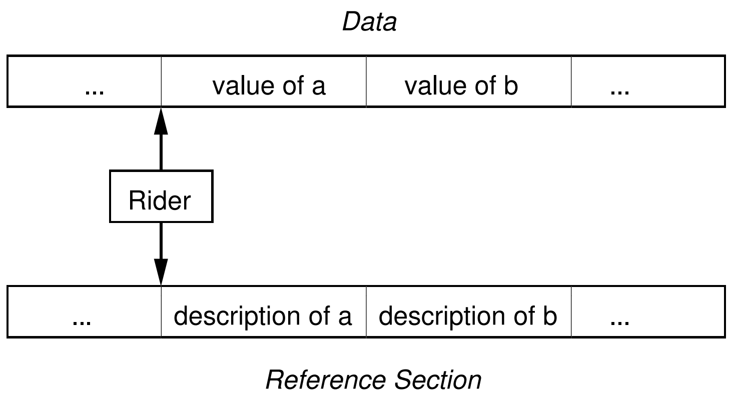


Figure 1. A rider as a link between the reference section and the data of a module

5. Applications

5.1 A Post-Mortem Debugger

A post-mortem debugger is a program that is invoked when another program terminates with a trap. Its responsibility is to show the machine state in a human-readable form, i.e., in terms of the program's source code. At minimum, it has to show the currently active procedures and the values of their local variables. In the original Oberon system this was done by the so-called trap handler. Its major shortcoming was that it only showed the values of unstructured variables whereas the contents of arrays, records and pointers could not be inspected (the reference information for structured variables was missing). Since we now have the full reference information, we re-implemented the trap handler so that it shows the values of *all* variables.

Reference Elements

The major remaining problem was how to present structured variables to the user. The common solution in debuggers is to show them in separate windows. We decided not to use this solution but to show all variables in the same window and to expand structured variables "in place". A mechanism for "zooming" into structures was already available in the Oberon system in the form of fold elements [MöKo96]. We extended the fold elements so that they now include also relevant reference information. These extended elements were called *reference elements*. Like fold elements, reference elements occur in pairs and are shown as small triangles behind the name of a structured variable. In collapsed state (▶◀) they do not enclose text whereas in expanded state (▷ i = 17 ◁) they enclose a list of variables with their names and values. A click on one of the elements changes their state from collapsed to expanded and vice versa. Consider the following example:

```
VAR
  i: INTEGER;
  rec: RECORD x, y: CHAR END;
  arr: ARRAY 3 OF INTEGER;
  ptr: POINTER TO RECORD a, b: INTEGER END;
```

The new trap handler shows the values of these variables in the following form (collapsed on the left-hand side, expanded on the right-hand side):

```
i = 17              i = 17
rec = ▶◀            rec = ▷
                      x = "a"
                      y = "b" ◁
arr = ▶◀            arr = ▷
                      1 = 10
                      2 = 11
                      3 = 12 ◁
ptr = ^ ▶◀          ptr = ^ ▷
                      a = 4
                      b = 5 ◁
```

Reference elements are implemented in module *RefElems* which also provides a procedure *WriteRider* to generate the contents of expanded elements.

```
DEFINITION RefElems;
  IMPORT FoldElems, Texts, Ref;

  TYPE
    Elem = POINTER TO ElemDesc;
    ElemDesc = RECORD (FoldElems.ElemDesc) END;

  PROCEDURE HandleElem (e: Texts.Elem; VAR m: Texts.ElemMsg);
  PROCEDURE WriteRider (VAR w: Texts.Writer; VAR r: Ref.Rider; ind: INTEGER);
END RefElems.
```

ElemDesc extends the standard *FoldElems* and adds some fields pertinent to the reference information for the structured variable to which the element refers. If this variable is a pointer, the element also stores a copy of that pointer in order to make sure that the referenced object is not reclaimed by the garbage collector.

Operations

- *HandleElem* is the message handler of a *RefElem*.
- *WriteRider(w, r, i)* writes the sequence of items (variables, fields, elements) on which the rider *r* is positioned to the writer *w*. It uses an indentation of *i* tabs. The list is written as a sequence of pairs <*name*, *value*> where the value of structured items is again represented by a pair of collapsed *RefElems*.

A sketch of the procedure *WriteRider* should help to understand how it works. The rider *r* is used to iterate over all items of the sequence on which it is positioned and to write them to the writer *w*. If a structured item is encountered (record, array, pointer) a pair of *RefElems* is inserted, which hide the item's structure from the user. By clicking on these elements they get expanded and reveal the item's structure.

```
PROCEDURE WriteRider* (VAR w: Texts.Writer; VAR r: Ref.Rider; ind: INTEGER);
  VAR si: SHORTINT; i, j: INTEGER; li: LONGINT; re: REAL; lr: LONGREAL;
    ch: CHAR; b: BOOLEAN; s: SET; proc: Ref.ProcVar; r1: Ref.Rider; p: S.PTR;
    str: ARRAY 1024 OF CHAR;
  ...

BEGIN
  WHILE r.mode # Ref.End DO
    WriteLn;
    FOR i := 1 TO indent DO WriteChar(09X) END;
    IF r.mode = Ref.Elem THEN WriteInt(r.idx) ELSE WriteString(r.name) END;
    WriteString(" = ");
    CASE r.form OF
      ...
    |  Ref.Int: r.ReadInt(i); WriteInt(i)
    |  Ref.LInt: r.ReadLInt(li); WriteInt(li)
```

```
        |  Ref.Real: r.ReadReal(re); WriteReal(re)
          ...
        |  Ref.Pointer:
             r.ReadPtr(p);
             IF p = NIL THEN WriteString("NIL")
             ELSE WriteString("^ "); ... (* insert RefElems containing the record fields *)
             END
          ...
        |  Ref.Array, Ref.DynArr:
             r.Zoom(r1);
             IF r1.form = Ref.Char THEN
                r.ReadString(str); WriteChar('"'); WriteString(str); WriteChar('"')
             ELSE ... (* insert RefElems containing the array elements *)
             END
        END;
        r.Next
     END
  END WriteRider;
```

Showing the Global Variables of a Module (System.State)

As in the original Oberon system, the command

```
  System.State (name | "^")
```

opens a viewer displaying the global variables of the specified module. The implementation sets a rider to the global variables and uses *RefElems.WriteRider* to write them to a viewer.

```
  PROCEDURE State;
     VAR modName: ARRAY 32 OF CHAR; r: Ref.Rider; t: Texts.Text;
  BEGIN
     In.Open; In.Name(modName);
     ... Open a viewer v with an empty text t ...
     Texts.WriteString(w, "MODULE "); Texts.WriteString(w, modName);
     Ref.OpenVars(modName, r);
     RefElems.WriteRider(w, r, 1);
     Texts.Append(t, w.buf)
  END State;
```

Showing the Local Variables of all Active Procedures (Trap viewer)

When a trap occurs, a viewer is opened showing information about the active procedures, their variables and parameters. Again, *RefElems* are used for representing the values of record, array, and pointer variables. A minor problem are records and arrays on the stack: As the stack is unrolled after a trap, they usually do not exist any more when the user clicks on the *RefElem*. Therefore the expanded view of their *RefElems* has to be generated before the stack is unrolled. For the same reason, pointers on the stack have to be copied to *RefElems* so that the garbage collector cannot remove the referenced objects after the stack was unrolled.

The contents of the trap viewer can be generated in the following way (*exceptionInfo* is a system-dependent data structure containing the register values at the time of a trap. It also specifies the stack that contains the relevant local variables):

```
Ref.OpenStack(exceptionInfo, r);
WHILE r.mode # Ref.End DO
  Texts.WriteLn(w); Texts.WriteString(w, r.mod);
  Texts.Write(w, "."); Texts.WriteString(w, r.name);
  ...
  r.Zoom(r1); RefElems.WriteRider(w, r1, 1);
  Texts.Append(t, w.buf);
  r.Next
END
```

5.2 A Database Interface

Databases allow users to perform queries on the stored data. Some databases even allow queries to be executed from within a program. That means that the programming language has to be extended so that query statements can be expressed or that a preprocessor must be used to specify the query in a preprocessor language.

Using module *Ref*, one can specify such queries as strings and pass them to a procedure that analyses the strings and executes the statements described by them. For example, one can write

```
conn.Prepare("CREATE TABLE Persons FOR Person")
```

without needing a language extension nor a preprocessor. We implemented a module *EmbeddedSQL* [Ste96] that provides access to ODBC databases [ODBC94] for Windows Oberon.

```
DEFINITION EmbeddedSQL;

  CONST
    (* return codes *)
    InvHandle = -2; Error = -1; Success = 0; SuccessWithInfo = 1;
    NoDataFound = 100;
  TYPE
    Connection = POINTER TO ConnectionD;
    ConnectionD = RECORD
      ret: INTEGER;   (* return code of last operation *)
      PROCEDURE (c: Connection) Prepare (sqlStr: ARRAY OF CHAR): Statement;
    END ;
    Statement = POINTER TO StatementD;
    StatementD = RECORD
      ret: INTEGER;   (* return code of last operation *)
      conn-: Connection;   (* the connection on which the statement is executed *)
      PROCEDURE (s: Statement) Execute;
      PROCEDURE (s: Statement) Fetch (): BOOLEAN;
      PROCEDURE (s: Statement) IsNull (name: ARRAY OF CHAR): BOOLEAN;
      PROCEDURE (s: Statement) SetNull (name: ARRAY OF CHAR);
    END ;

  PROCEDURE Open (source, user, passwd: ARRAY OF CHAR): Connection;

END EmbeddedSQL.
```

Types

Connection represents a communication channel between the application and the database. Requests are issued and responses are returned via this connection. *ret* indicates the success of the last operation.

Statement represents an SQL statement that has been prepared for execution via connection *conn*. *ret* indicates the success of the last operation.

Operations

- *conn := Open(source, user, password)* opens a connection to the database with the given user identification and password.
- *stat := conn.Prepare(s)* prepares an SQL statement (specified by the string *s*) for execution.
- *stat.Execute* executes the previously prepared SQL statement.
- *done := stat.Fetch*. If the execution of an SQL statement results in a table (i.e., a sequence of records), *Fetch* retrieves one row of the table (i.e., one record of this sequence) at a time and stores it in the variable(s) specified in the statement. If there are no more rows to retrieve, *done* becomes FALSE.
- *b := stat.IsNull(n)* returns TRUE if the variable specified by the name *n* contains a null value. Null values are special values which indicate that the value is not valid or present. As this cannot be expressed by a legal value in programming languages (e.g., 0 for integer variables, or "" for string variables), *IsNull* is necessary to check for the validity of a value.
- *stat.SetNull(n)* makes the variable specified by the name *n* contain a null value.

Embedded SQL and Oberon

For data transfer between the database and the application SQL statements use ordinary Oberon variables. In order to distinguish these variables from names that are used within the database (e.g. names of tables and columns), they are preceded by a colon. In the SQL statement

```
"SELECT firstName FROM Persons WHERE age > :minAge INTO :name"
```

minAge and *name* are Oberon variables. *minAge* is an input variable, and *name* is an output variable.

Variables can be either scalar or of a record type. When record variables are specified, they are implicitly expanded to their fields. The statement

```
"SELECT * FROM Persons INTO :person"
```

is therefore equivalent to

```
"SELECT * FROM Persons INTO :person.firstName, :person.lastName, :person.age"
```

Examples

We declare the type *Person* that will be used to represent persons.

```
TYPE
  Person = RECORD
    firstName, lastName: ARRAY 32 OF CHAR;
    age: INTEGER
  END ;

VAR
  conn: EmbeddedSQL.Connection;
  stat: EmbeddedSQL.Statement;

BEGIN
  conn := EmbeddedSQL.Open(source, user, password);
  stat := conn.Prepare("CREATE TABLE Persons FOR Person");
  stat.Execute();
END
```

After opening the connection, we create a table for the persons that we will later insert. The table will consist of as many columns (with appropriate types) as there are fields in the record type *Person*. The record type can be qualified with the module in which the type is declared.

In order to insert data into the table, we prepare an INSERT statement in which we specify the variables containing the values to be inserted (*firstName, lastName, age*). These variables are preceded by a colon (which distinguishes them from database identifiers for tables and columns). Then we assign values to the variables and consider null values (i.e., values that should remain undefined). When we finally execute the statement the values from the variables are taken and transferred into the database. Note that the statement - once it has been prepared - can be executed several times with different values.

```
PROCEDURE Insert;
  VAR firstName, lastName: ARRAY 32 OF CHAR; age: INTEGER;
BEGIN
  In.Open;
  stat :=
    conn.Prepare("INSERT INTO Persons VALUES (:firstName, :lastName, :age)");
  REPEAT
    In.Name(firstName); In.Name(lastName); In.Int(age);
    IF firstName = "NULL" THEN stat.SetNull("firstName") END ;
    IF lastName = "NULL" THEN stat.SetNull("lastName") END ;
    IF In.Done THEN stat.Execute() END
  UNTIL ~In.Done
END Insert;
```

In order to retrieve all persons older than *minAge* we can use the following procedure *Select*. After preparing the SELECT statement and assigning values to the input variables (in this case *minAge*), we execute the statement and fetch the resulting data row by row. As the table is defined for the type *Person*, every row is a record of type *Person*. If we were only interested in the columns *firstName* and *lastName*, we could use a SELECT statement like "SELECT firstName, lastName FROM Persons WHERE age >= :minAge INTO :person.firstName, :person.lastName".

```
PROCEDURE Select;
  VAR person: Person; minAge: INTEGER;
BEGIN
  stat := conn.Prepare
    ("SELECT * FROM Persons WHERE age >= :minAge INTO :person");
  In.Open; In.Int(minAge);
  stat.Execute();
  WHILE stat.Fetch() DO
    Out.Ln;
    IF stat.IsNull("person.firstName") THEN Out.String("NULL")
    ELSE Out.String(person.firstName)
    END ;
    Out.String(", ");
    IF stat.IsNull("person.lastName") THEN Out.String("NULL")
    ELSE Out.String(person.lastName)
    END ;
    Out.String(", ");
    IF stat.IsNull("person.age") THEN Out.String("NULL")
    ELSE Out.Int(person.age, 0)
    END
  END ;
END SelectAll;
```

Implementation

The analysis of the SQL commands in the strings is implemented using module *Ref*. Any variable preceded by a colon is looked up in the local scope of the procedure that issued the SQL statement (the local scope contains the local variables, as well as the parameters of the procedure). The addresses of such variables are then passed to low-level modules of the ODBC database.

5.3 A Heap Inspector

Inspector is a tiny module that can be used to browse data on the heap. It was implemented using module *Ref* and is similar to the post-mortem debugger described in Section 5.1.

```
DEFINITION Inspector;
  IMPORT SYSTEM;

  PROCEDURE InspectPointer (p: SYSTEM.PTR);
  PROCEDURE InspectElem;
  PROCEDURE InspectViewer;
  PROCEDURE InspectDirectory;
  PROCEDURE InspectSyntaxTree;
END Inspector.
```

Operations

- *InspectPointer(p)* opens a viewer with the fields of the record pointed to by *p*.
- *InspectElem* opens a viewer with the record fields of the selected text element.
- *InspectViewer* opens a viewer with the record fields of the star-marked viewer.
- *InspectDirectory dirName* opens a viewer displaying the Directories.Directory structure of the directory *dirName*.
- *InspectSyntaxTree modName* compiles the module *modName* and opens a viewer with the syntax tree of the module.

5.4 A General Output Module

In Oberon there is no general output procedure like in Pascal. Instead there are individual output procedures for every basic type, such as *Out.Int* for integers or *Out.Char* for characters. Writing a message to the screen can therefore be quite tedious and unreadable, for example:

```
Out.String("Frame: X = "); Out.Int(f.X, 0); Out.String(", Y = "); Out.Int(f.Y, 0)
Out.String(", W = "); Out.Int(f.W, 0); Out.String(", H = "); Out.Int(f.W, 0);
Out.Ln
```

Module *Put* allows programmers to write such output statements in a more concise and readable way, for example:

```
Put.S("Frame: X = #f.X, Y = #f.Y, W = #f.W, H = #f.H$")
```

The argument string of *Put.S* is written to the standard output viewer. Every designator that is preceded by a "#" is replaced by its value. A "$" denotes the beginning of a new line.

The interface of module *Put* is simple:

```
DEFINITION Put;
  PROCEDURE S (s: ARRAY OF CHAR);
END Put.
```

Procedure *S* uses module *Out* to write the string *s* to the standard output viewer. The string has the following form:

```
string = '"' { char               -- write this character
             | "#" designator     -- write the value of this designator
             | "$"                -- skip to a new line
             }
         '"'.
```

The designator must denote a variable of a basic type (in which case the value of this variable is written), an array of character (in which case the array is written as a string), or a pointer variable (in which case the numeric pointer value is written). As an implementation restriction the designator must not denote a variable of an intermediate procedure level and must not contain type guards.

Put.S is implemented using module *Ref*. The designators in the argument string are looked up piecewise in various scopes. For example, the designator *rec.a[i]* is looked up as follows:

- *rec* is first searched in the local scope (*OpenStack*) and - if not found - in the global scope (*OpenVars*). If it is still not found, *Put.S* checks if *rec* is a module name (because it is followed by a "."). In this case, the next name is assumed to be a global variable of this module.
- Assume that *rec* is found and turns out to be a record variable. A rider *r* is positioned on *rec*. *Put.S* zooms into *rec* (*r.Zoom(r)*) and looks for the field *a*. If it is found, the rider *r* is positioned on it.
- Since *a* turns out to be an array, *Put.S* zooms into this array and starts a new search for the designator *i* (first using *OpenStack*, then *OpenVars*, etc.). The value of *i* is read, the rider *r* is positioned to the *i*-th element of *a*, and the value of this element is read. This is the value of *rec.a[i]*. It is written to the output viewer in readable form.

Acknowledgement

We would like to thank J. Templ for discussing with us the interface and the implementation of module *Ref* as well as for proofreading an earlier draft of this report.

Appendix A: Run-Time Data Structures

This appendix shows the layout of several run-time data structures of the Oberon system for Windows and Power Macintosh. Most of them were designed by members of the OP2 project (R. Crelier, J. Templ, M. Brandis, M. Hausner, M. Franz) but they were undocumented so that a systems programmer had to reverse-engineer them from the source code again and again. To avoid this task we include their description in this report.

A.1 Module descriptors

Module descriptors contain information about loaded modules. This information is used by the loader, the debugger, and other metaprogramming tools. In Windows Oberon module descriptors were designed so that they are easy to use by a programmer, in PowerMac Oberon they were designed so that they are easy to initialize by the loader.

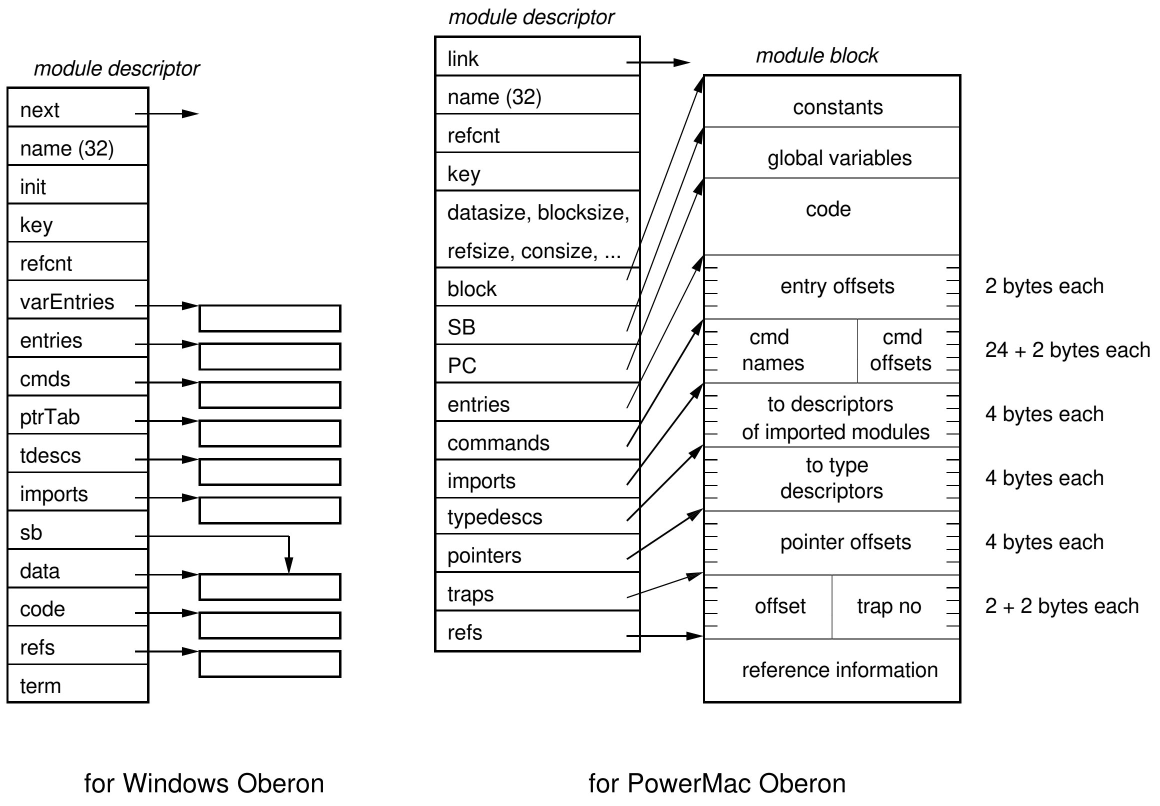


Figure 2. Module descriptor layout

A.2 Type descriptors

For every record type there is a unique type descriptor, which stores information such as the type name, a pointer to the descriptor of the module that declares the type, the table of methods, the table of base types, the extension level, the size of the record, and the offsets of pointers within the record type (necessary for garbage collection). Every record that is allocated on the heap has a hidden pointer to its type descriptor (*tag*). Module *Types* implements portable access to the implementation-dependent layout of type descriptors.

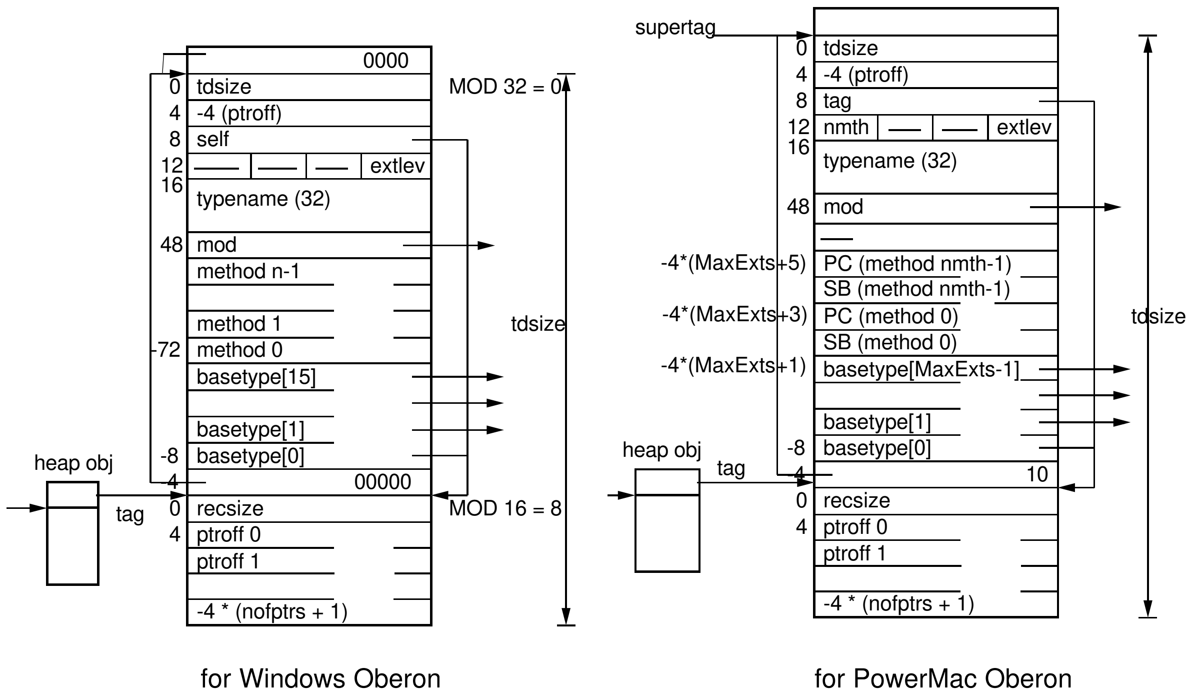


Figure 3. Type descriptor layout

A.3 Dynamic array descriptors

Arrays that are allocated on the heap need a descriptor for the following reasons:

a) The length of dynamic arrays is only determined at run time. Thus it has to be stored in a descriptor.
b) If the array contains pointers, the garbage collector must be able to find them. Therefore the pointer offsets have to be stored in a descriptor.

The array descriptor is allocated in a single block together with the array elements (see the figures on the next two pages). The whole block is treated as a record and therefore has itself a hidden pointer to a type descriptor. Three cases have to be distinguished:

a) *The array elements are records containing pointers*: The type descriptor of the array block is the type descriptor of the element record. This type descriptor contains the pointer offset within *one* record. The garbage collector replicates this type descriptor over the whole array so that all pointers can be found. The fact that the type descriptor must be replicated is indicated to the garbage collector by the "array bit" in the array block's tag.
b) *The array elements are (arrays of) pointers*: The array block gets a dummy type descriptor as if the elements of the array were records with a single pointer field. This record description is again replicated over the array. The fact that the type descriptor must be replicated is indicated to the garbage collector by the "array bit" in the array block's tag.
c) *The array elements are of a type that does not contain pointers*: The array block gets a dummy type descriptor as if it were a record without pointers.

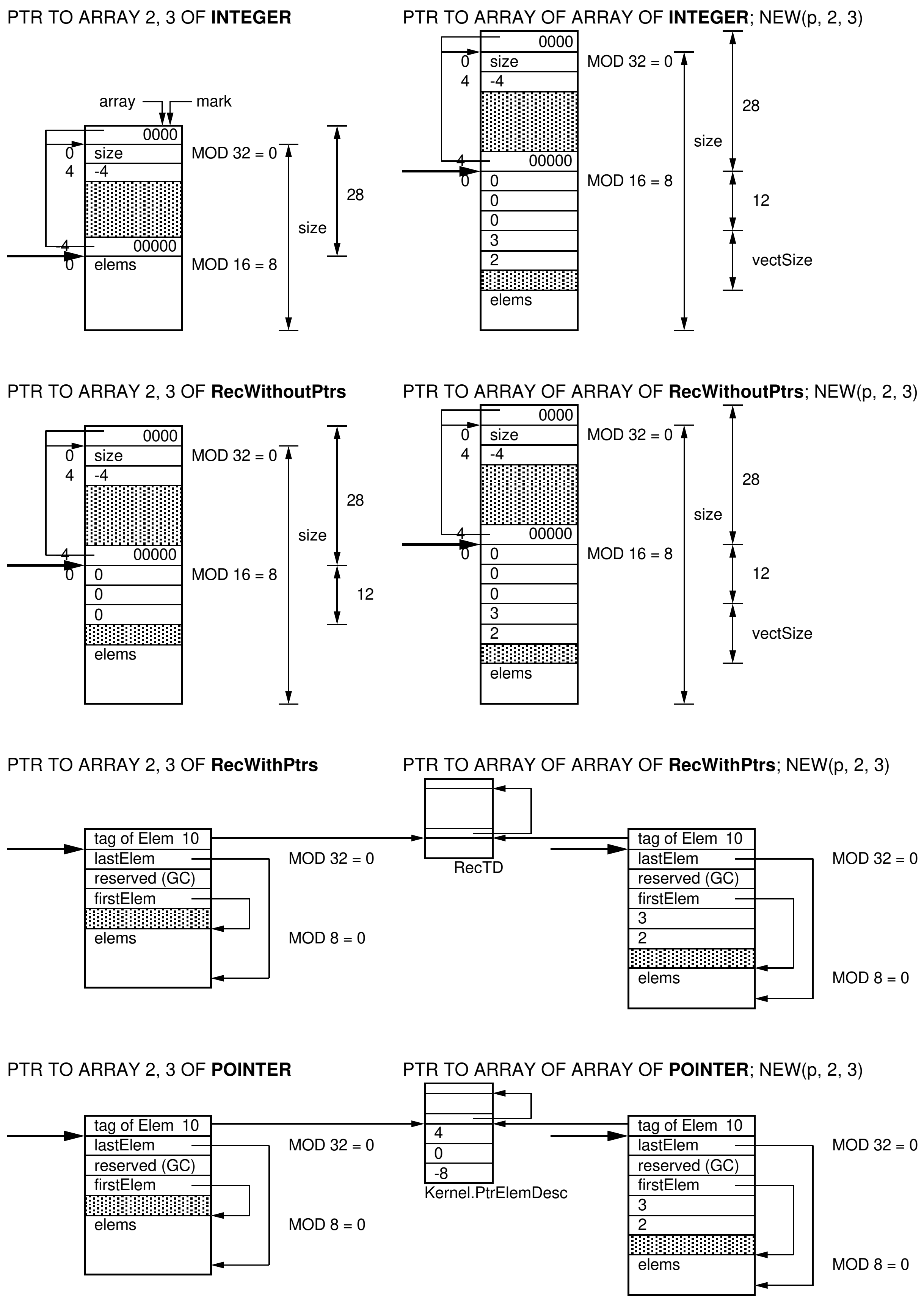


Figure 4. Dynamic array descriptors under Windows Oberon

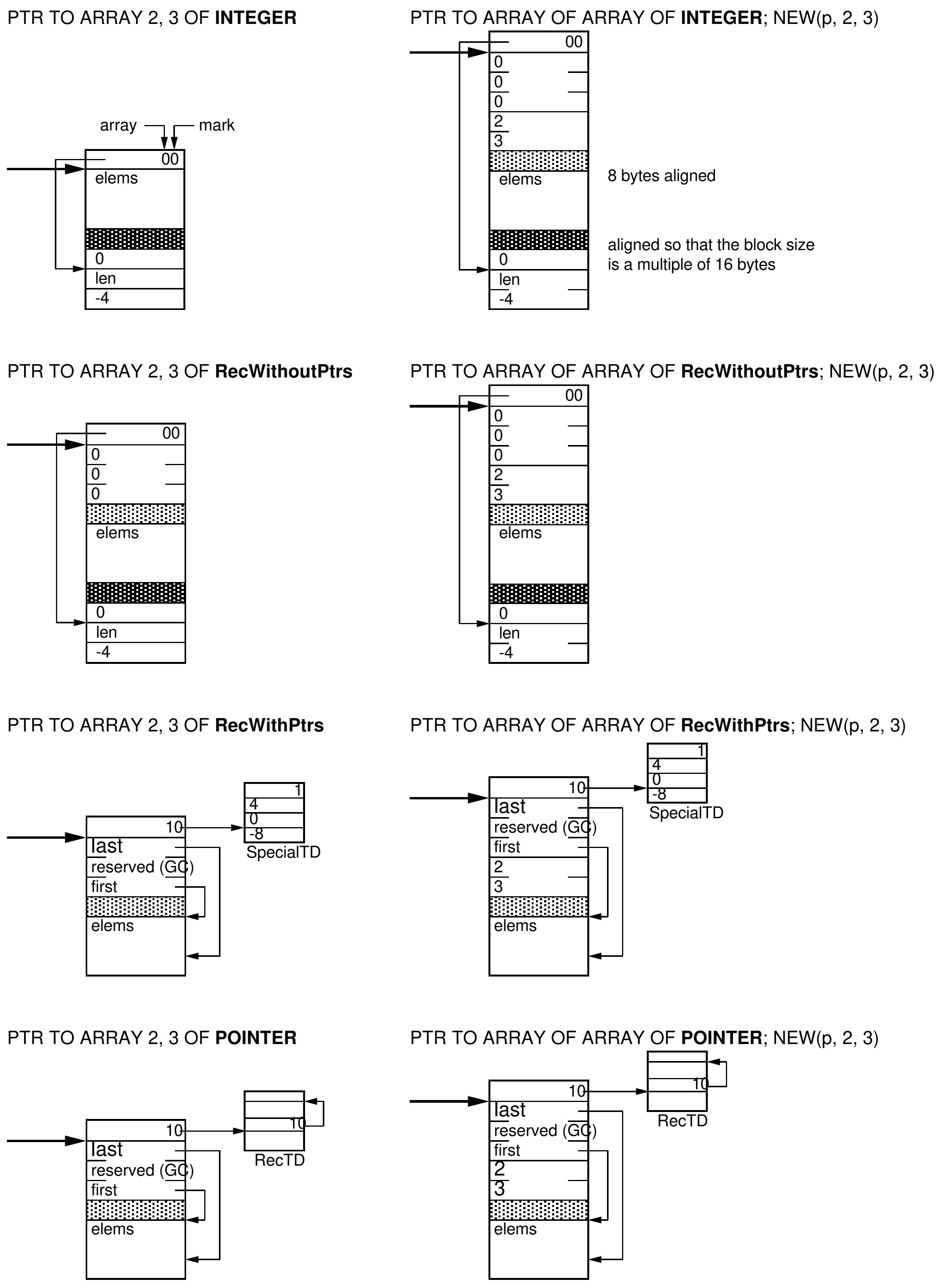

PTR TO ARRAY 2, 3 OF **INTEGER**
array
mark
00
elems
0
len
-4
PTR TO ARRAY OF ARRAY OF **INTEGER**; NEW(p, 2, 3)
00
0
0
0
2
3
elems
0
len
-4
8 bytes aligned
aligned so that the block size is a multiple of 16 bytes
PTR TO ARRAY 2, 3 OF **RecWithoutPtrs**
00
0
0
0
elems
0
len
-4
PTR TO ARRAY OF ARRAY OF **RecWithoutPtrs**; NEW(p, 2, 3)
00
0
0
0
2
3
elems
0
len
-4
PTR TO ARRAY 2, 3 OF **RecWithPtrs**
10
last
reserved (GC)
first
elems
1
4
0
-8
SpecialTD
PTR TO ARRAY OF ARRAY OF **RecWithPtrs**; NEW(p, 2, 3)
10
last
reserved (GC)
first
2
3
elems
1
4
0
-8
SpecialTD
PTR TO ARRAY 2, 3 OF **POINTER**
10
last
reserved (GC)
first
elems
10
RecTD
PTR TO ARRAY OF ARRAY OF **POINTER**; NEW(p, 2, 3)
10
last
reserved (GC)
first
2
3
elems
10
RecTD

Figure 5. Dynamic array descriptors under PowerMac Oberon

The first array element is aligned on an 8 byte boundary so that there is no problem when accessing the elements of a POINTER TO ARRAY OF LONGREAL (some processors require LONGREAL values to be aligned on an 8 byte boundary). The fields inserted for alignment are shown in grey.

A.4 Stack Frames

Stack Frame under Windows Oberon

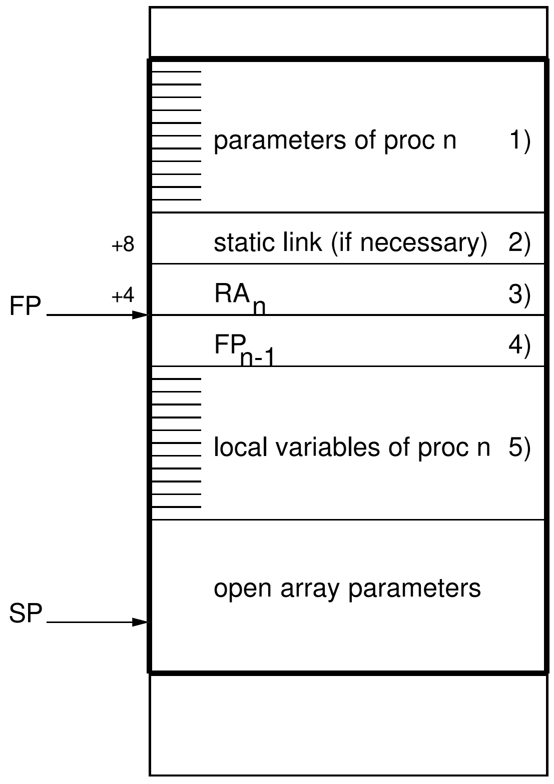
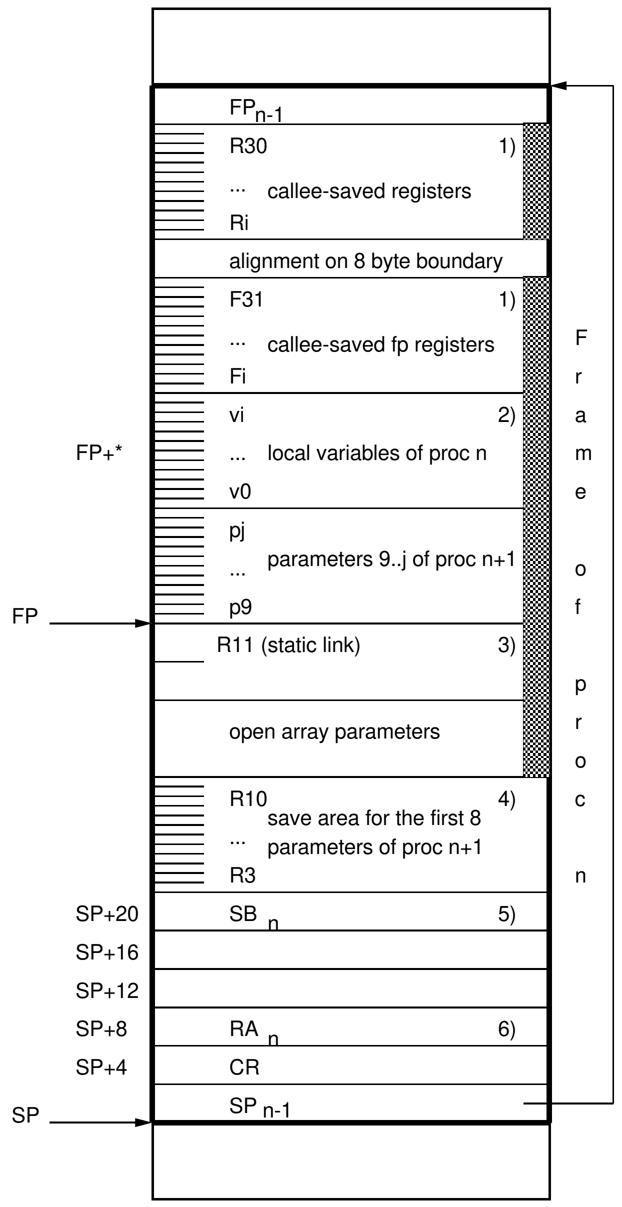


Figure 6. Stack frame under Windows Oberon

1) Parameters of proc $_n$ have positive offsets relative to the frame pointer. The first parameter of the procedure is 12[FP], if there is a static link, otherwise 8[FP]. Further parameters have higher offsets.
2) If the static link of the procedure is necessary (to access intermediate level variables), the static link is located at 8[FP].
3) RA $_n$ is the return address, i.e. the next instruction executed after the return from proc $_n$.
4) FP $_{n-1}$ is the frame pointer of proc $_{n-1}$. This forms the dynamic call chain (used for unrolling the stack when a run-time trap occurred).
5) Local variables of proc $_n$ have negative offsets relative to the frame pointer. The first local variable is 4[FP]

Stack Frame under PowerMac Oberon


FP$_{n-1}$
R30 1)
... callee-saved registers
Ri
alignment on 8 byte boundary
F31 1)
... callee-saved fp registers
Fi
vi 2)
FP+*
... local variables of proc n
v0
pj
... parameters 9..j of proc n+1
p9
FP
R11 (static link) 3)
open array parameters
R10 4)
... save area for the first 8 parameters of proc n+1
R3
SP+20
SB $_n$ 5)
SP+16
SP+12
SP+8
RA $_n$ 6)
SP+4
CR
SP
SP $_{n-1}$
Frame of proc n

SP	R1
FP	R31
RA	R0
SB	R2
scratch	R3..R12
local variables	R13..R30

Areas with a grey bar on their right-hand side are only allocated if necessary

Figure 7. Stack frame under PowerMac Oberon

1) All registers that are used for local variables and formal parameters in proc $_n$ are saved here upon procedure entry. At most 18 general purpose registers (R13..R30) and 18 floating point registers (F14..F31) can be used for local variables.
2) For every local variable of proc $_n$ there is a storage location here, regardless if the variable is in a register or in memory, if it is a simple or a structured variable. Storage locations for formal parameters of proc $_n$ (regardless if integer or floating point) are in the frame of proc $_{n-1}$.
3) Only for local procedures; otherwise this field is missing.
4) Proc $_{n+1}$ may save its parameters here if they were passed in registers but then used as VAR parameters. On entry of proc $_n$ the actual parameters which are passed in R3..Ri are transfered to Rj..R30 (i.e., R3 -> R30, R4 -> R29, etc.)
5) Only saved if proc $_n$ is not a leaf procedure.
6) Saved by the callee (proc $_{n+1}$) but only if the callee is not a leaf procedure

A.5 How to get the modules

All modules described in this report (*Modules*, *Types*, *Ref*, *RefElems*) are part of the standard distribution of the Linz version of Oberon V4 for Windows and Power Macintosh and can be obtained via anonymous internet file transfer (ftp).

Hostname:	ftp.ssw.uni-linz.ac.at
Directories:	/pub/Oberon/Windows
	/pub/Oberon/PowerMac

References

[Att89] G. Attardi et al.: Metalevel Programming in CLOS. Proceedings of the ECOOP'89 conference. Cambridge University Press, 1989.

[Cre90] R. Crelier: OP2 - A portable Oberon compiler. Computer Science Report 125, ETH Zurich, 1990.

[GPHT91] R. Griesemer, C. Pfister, B. Heeb, J. Templ: Oberon technical notes. Computer Science Report 156, ETH Zurich, 1991.

[GR83] A. Goldberg, D. Robson: Smalltalk-80, the language and its implementation. Addison-Wesley, 1983.

[McCar60] J. McCarthy: Recursive functions of symbolic expressions and their computation by a machine. Communications of the ACM 3 (4), 1960, 184-195

[MMN93] O. Lehrmann-Madsen, B. Moller-Pedersen, K. Nygaard: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, 1993.

[MöKo96] H. Mössenböck, K. Koskimies: Active Text for Structuring and Understanding Source Code. To appear in Software - Practice and Experience, 1996.

[ODBC94] Microsoft Open Database Connectivity Software Development Kit Version 2.0, Microsoft Press, 1994

[Rei91] M. Reiser: The Oberon System. User Guide and Programmer's Manual. Addison-Wesley, 1991.

[Smi82] B. C. Smith: Reflection and Semantics in a Procedural Language. PhD thesis, M.I.T., 1982.

[Ste96] C. Steindl: Entwurf und Implementierung einer Stücklistenverwaltung mittels einer Client/Server-Datenbank. Diploma thesis, University Linz, 1996.

[Tem94] J. Templ: Metaprogramming in Oberon. Dissertation, ETH Zurich, 1994.

[US87] D. Ungar, R. B. Smith: SELF: The Power of Simplicity. Proceedings of the OOPSLA'87 conference, Orlando, SIGPLAN Notices 22 (12), 1987.

[WiGu89] N. Wirth, J. Gutknecht: The Oberon System. Software-Practice and Experience, 19(9), 1989, 857-893.

[WiGu92] N. Wirth, J. Gutknecht: Project Oberon - The design of an operating system and compiler for NS32000. Addison-Wesley, 1992.

Technical Reports

The following reports can be obtained as postscript files from
ftp.ssw.uni-linz.ac.at/pub/Reports/