Report of Ulm's Oberon Compiler

Andreas Borchert, University of Ulm, SAI, Helmholtzstr. 18, D-89069 Ulm

Introduction

This report is to be seen as supplement of the revised Oberon report in respect to Ulm's implementation of Oberon.

Specifications of data representation, alignment and space usage are implementation specific. Thus, modules relying on assumptions about internal representations are nonportable. However, because implementations of low level modules sometimes need some knowledge about internals, they have been included.

The enclosed numbers of the section headings are references to the paragraphs of the revised Oberon report.

The command line invocation of the Oberon compiler is described in *oc*.

Vocabulary and representation (3.)

White space consists of blanks (20X), tabs (09X), newlines (0AX), and form feeds (0CX). Source file tabs (outside of string constants) are interpreted (for listing generation) in conformance to the UNIX standard tab setting, i.e. tab positions at 9, 17, 25, ... The line length (after tab expansion and including the terminating 0AX) is limited to 32,768 characters. The last source line does not need to be terminated by a newline.

User defined identifiers are significant in their whole length. The maximal identifier length is limited by the maximal line length.

Comments may be nested (yet undocumented revision of Oberon).

Compile options may be specified at the beginning of a comment:

```
$ ControllingComment = "(*" { CompileOption } AnyText "*)" .
$ CompileOption = WhiteSpace "$" OptionLetter ("+" | "-" | "=") .
$ OptionLetter = UpperCaseLetter .
```

Options are boolean values with an initial value of **TRUE** (if not overridden by *oc*). Option settings may be nested: "+" sets the option value to **TRUE**, "-" to **FALSE**, and "=" restores the old value.

Following option letters are currently implemented:

A
: causes checks for assertions to be generated.

C
: requests checks to be generated for conversions from **INTEGER** to **SHORTINT**.

I
: enables some Oberon revisions: ":=" instead of ":" for aliasing in import lists and nested comments. Note that this option is nowadays obsolete and will not supported by future versions.

M
: this option causes the two least significant bits of the type tag to be masked out on each access. This may be useful for some garbage collecting algorithms and requires the option to be applied for all modules.

O
: requests pointer types to point to tagged data structures. This options is to be switched off for pointer types which are intended to point to areas which have not been created by **NEW** or **SYSTEM.NEW**. These pointer types will not be traced on garbage collections. **IS** operators or **WITH** statements are rejected by the compiler for untagged pointers.

P
: requests pointer types to be traced on garbage collections. This option must be switched off in cases where pointers refer to untagged locations. Only types which contain pointers are affected by this option. In case of exported types only the setting in the definition is considered by the compiler.
Note that usually *O* and *P* are switched off simultaneously but *P* may be switched off alone. In the latter case the compiler still expects type tags but does not generate tracing information for them.

R
: enables parameter types to be given as required by revised Oberon. Note that this option is nowadays obsolete and will not supported by future versions.

S
: If switched on, not only strings of length 1 are treated as possible strings but also all other character constants, e.g. 4X, **MAX**(**CHAR**), **CAP**("A"), **CHR**(4). While the report doesn't encourage this interpretation, the Zürich compilers treat all these character constants as possible strings of the length 1.

T
: causes range checks for array indices to be generated.

Constants

Integer constants (3.2)

Integer constants range from -2147483648 (-2^31) to 2147483647 (2^31-1). The type of small integer constants which fits into **SHORTINT** depends on their usage. Usually they are of the smallest integer type possible but in expressions they adapt the appropriate data type; e.g. 12*12 yields 144, 12 fits into **SHORTINT** but not 144.

Floating point constants (3.2)

Floating point constants of **REAL** type range from -1.797693134862314e+308 to 1.797693134862314e+308. The least positive value of type **REAL** is 2.225073858507201e-308. **LONGREAL** floating point constants are not yet implemented. Scale factors with leading "D" are accepted and cause the constant to be of type **LONGREAL** but the constant is represented as **REAL**. Thus, the extended precision is lost and constant values outside of the **REAL** range are not accepted.

Character constants (3.3)

Character constants in hexadecimal notation range from 0X to 0FFX.

String constants (3.4)

The length of strings is not limited. Strings must not contain newlines (0AX). "" denotes the empty string and consists of the terminating 0X character.

Declarations and scope rules (4.)

Export marks are not yet implemented. Instead of marking items which are to be exported a textually separated definition is necessary as defined in the first Oberon report.

```
$ identdef = ident .
```

Constant declarations (5.)

Standard functions including some functions exported from **SYSTEM** are permitted in constant expressions. Exceptions are **SYSTEM.ADR**, **SYSTEM.BIT**, **SYSTEM.TAS**, **SYSTEM.UNIXCALL**, **SYSTEM.UNIXFORK**, and **SYSTEM.UNIXSIGNAL**.

Floating point operations in constant expressions are currently implemented without secure evaluation. Thus, overflow or division by zero are not detected.

Representation of data types

Basic types (6.1)

type	size	representation
BOOLEAN	1	**TRUE** is represented as 1 (least significant bit), **FALSE** as 0.
CHAR	1	characters ranges from 0X to 0FFX.
SHORTINT	1	is signed and ranges from -128 to 127.
INTEGER	4	is signed and ranges from -2147483648 (-2^31) to 2147483647 (2^31-1).
LONGINT	4	is represented like **INTEGER**.
REAL	8	is represented in double precision (MC68881). Values range from -1.797693134862314e+308 to 1.797693134862314e+308. Floating point exceptions are disabled on default and can be enabled by usage of the *MC68881* system module.
LONGREAL	12	is represented in extended precision (internal format of the MC68881).
SET	4	**MAX(SET)** equals 31. {0} is represented as a 4-byte unsigned integer with the most significant bit set. {31} is represented with the least significant bit set.
BYTE	1	represents one byte (8 bits).

Aggregate types (6.2 and 6.3)

Records and arrays occupy multiples of 4 bytes. Array and record components are represented without extra packing. Array elements and record components are aligned on 4-byte-boundaries if the size of elements or components is greater or equal to 4 bytes.

Arrays are implemented in row major order. Dynamic arrays consist of a pointer to the array and a dope vector. The dope vector consists of 4-byte integers which equal the values returned by the **LEN** standard function.

Empty records have zero length. Type tag pointers are added at run time. They precede the record value (offset -4) in case of allocation by **NEW** or follow (in stack direction) the pointer to the record value in case of variable parameters. Type tags are pointers to following structure:

```
TypeTag = POINTER TO TypeTagRec;
TypeTagRec =
   RECORD
      size: LONGINT;              (* size of the type *)
      module: SysModules.Module;
      typeno: INTEGER;            (* of corresponding reference file *)
      hierlen: LONGINT;           (* number of base types *)
      basetype: ARRAY hierlen OF TypeTag;
         (* the 2nd basetype extends the 1st etc. *)
      (* pointer list *)
      sentinel: LONGINT;
   END;
```

size is the size of the type as returned by **SIZE** and not the size of allocated area for objects of this type. *module* and *typeno* allow debuggers to identify the type. Detailed informations are given in *SysModules* and *reffile(5)*. *hierlen* gives the number of base types (not counting the type itself) and *basetype* is an array of type tags of the base types. These informations are followed by a pointer list which is terminated by *sentinel* (-2147483648 = **MIN(INTEGER)**).

Pointer lists conform to following grammar:

```
PointerList =     { PtrDescriptor } .
PtrDescriptor =   SimplePointer | PointerArray | RecordArray | Other .
SimplePointer =   offset .
```

```
PointerArray =    offset ElementCount .
RecordArray =     offset ElementCount typeTag .
ElementCount =    [ numberOfDynDimensions ] count .
Other =           offset ( TmpPointer | SimpleAddress | AddressArray ) .
TmpPointer =      TmpPointerSY StartAddr EndAddr .
SimpleAddress =   SimpleAddressSY
AddressArray =    AddressArraySY ElementCount .
StartAddr =       addr .
EndAddr =         addr .
```

Different pointer descriptors are recognized by their two least significant bits:

```
    |             bit1
bit0|       0             1
____|_______________|_____________
0   |  SimplePointer|  Other
1   |  PointerArray |  RecordArray
```

Other types of pointers are determined by a selector (4 bytes):

```
TmpPointerSY =      0
SimpleAddressSY =   1
AddressArraySY =    2
```

Type tags are not only generated for record types and arrays but also for procedures and global variables which can be found by the tables of *SysModules*.

Some additional notes: Temporary pointers are only to be considered if the associated program counter ranges inside [*StartAddr*..*EndAddr*). While pointers are guaranteed to point to the beginning of a structure, addresses are free to point inside a structure and they may reference other areas (e.g. global or local data). Base types are encoded as *RecordArray* with a length of 1 if a record contains private parts. Dynamic arrays are encoded with *numberOfDynDimensions* = - *n* for *n*-dimensional arrays.

Pointer types (6.4)

Pointer types are represented as 4-byte unsigned integers. **NIL** is represented as 0. Pointers to records as variable parameters are followed (in stack direction) by a type tag. The Oberon runtime system guarantees pointer values to be initialized to **NIL**. While the compiler does not generate checks against references through **NIL**, they are catched by the memory management and lead to segmentation violations.

Procedure types (6.5)

are represented by a pointer to the starting address of the procedure.

Procedure declarations (10.)

An asterisk following the symbol **PROCEDURE** is defined to be a hint for the compiler that the procedure is to be usable as parameter and assignable to variables of a compatible procedure type. This implementation accepts these hints but does not require them.

Even exported procedures need a forward declaration when their usage precedes the actual declaration. The compiler requires identical parameter identifiers for forward declarations and exports.

The result type of a procedure cannot be a record or an array.

Formal parameters (10.1)

The relaxed parameter compatibility rule for the formal type **ARRAY OF BYTE** applies for variable *and* value parameters.

The actual parameter type must be identical to the type of the corresponding formal parameter in case of variable parameters. Exceptions are records and pointers, where the formal parameter type must be a base type of the actual parameter type. This conforms to unrevised Oberon but revised Oberon restricts the exception to record types.

Standard procedures and functions (10.2)

ASH(*x*, *n*) (arithmetic shift) shifts from left to right in case of negative *n* (*x* **DIV** 2^(-*n*)).

LEN requires a variable identifier as first argument. Thus designators with index expressions are not permitted.

COPY(*x*, *v*) guarantees 0X-termination of *v*.

HALT(*x*) causes *Process.Exit*(*x*) to be called.

ASSERT(*assertion*: **BOOLEAN**) has been added in conformance to newer Oberon compilers from ETH Zürich. There exist an optional second parameter (of type **INTEGER**) which is ignored by this implementation. Assertions, if switched on by the corresponding option, test *assertion* for being **TRUE** and generate a run time error otherwise (see *RTErrors*).

Compilation units (11.)

Compilation units are either definitions or modules. The definition is an excerpt of the module and specifies the items visible to all clients importing the module.

```
$ definition = DEFINITION ident ";"
$    [ImportList] Definitions END ident "." .
$ Definitions = { CONST { ConstantDeclaration ";" } |
$    TYPE { TypeDeclaration ";" } | VAR { VariableDeclaration ";" } }
$    { ProcedureHeading ";" } .
```

All declarations of the definition are to be repeated in the module.

The syntax of the import list has been changed to:

```
$ ImportList = IMPORT import { "," import } ";"
$ import = identdef [ ":=" ident ] .
```

The initialization of the modules depends on *SysMain*. The default implementation initializes all modules in linkage order. Modules can be initialized (and in later revisions possibly loaded) by *Loader* if their name is known.

Parameterless procedures exported by modules compiled with **-C** are inserted into the structure exported by *SysCommands* during the early startup phase. The lists of *SysCommands* and *SysModules* are completed before any modules will be initialized.

The module SYSTEM

All standard procedures and functions of the **SYSTEM** module as defined in the report (for NS32000) are implemented with the exception of **CC**. **CC** cannot be implemented on a MC68020. Note that the return type of **SYSTEM.ADR** has been changed to **SYSTEM.ADDRESS** (see below). Following procedures and functions have been added:

SYSTEM standard functions

TAS(*v*) is of boolean type and corresponds to the equally named machine instruction (test and set). The operation is undividable and returns the value of the boolean variable *v* before setting it to **TRUE**.

UNIXCALL(*syscall*, *d0*, *d1*, *arg*, ...) returns a **BOOLEAN** value and allows to code all UNIX system calls (except *fork* and *signal*). *syscall* is a small integer constant which determines the sort of call. Valid system call numbers are exported by *Sys*. *d0* and *d1* are variable parameters. Their values are stored into the corresponding data registers of the MC68020. After execution of the system call the register values are copied back to the variable parameters. Typically *d0* contains the return value of the system call or the error code (if **UNIXCALL** returns **FALSE**). The arguments following should be **INTEGER** sized and are pushed in reversed order (in C convention) onto the stack. Character strings must be

0X-terminated and given by their address.

UNIXFORK(*pid*) returns a **BOOLEAN** value and realizes the *fork* system call. *pid* is a variable parameter and gets the process id of the child process (parent process) or 0 (child process). In error case (return of **FALSE**) the error number is stored into *pid*.

UNIXSIGNAL(*signo*, *p*, *old*, *error*) returns a **BOOLEAN** value and interfaces the signal handling runtime routines. *signo* must be a valid signal number (see *signal(2)* for details), *p* a signal handling Oberon procedure (or 0 for **SIG_DFL** and 1 for **SIG_IGN**). Signal handling Oberon procedures are of following type:

```
SignalHandler = PROCEDURE(signo: INTEGER; sigcode: ARRAY OF BYTE);
   (* signo:   signal number
      sigcode: additional information
               (e.g. faulting address in case of segmentation violation)
   *)
```

An **INTEGER** argument type is also accepted for convenience of **SIG_DFL** and **SIG_IGN**.

old and *error* are output parameters. *old* returns the previous reaction on *signo* and *error* returns the error number in case of an error (return of **FALSE**).

These system procedures are not intended to be used outside of the Oberon system library. *Events*, *SysProcess*, *SysSignals*, *SysIO* and other modules offer more portable interfaces to the UNIX system.

SYSTEM standard procedures

HALT(*exitcode*) causes an immediate exit of the calling process without any cleanup procedures (i.e. trap to the operating system).

WMOVE(*from*, *to*, *n*) copies *n* 4-byte words from address *from* to *to*. **WMOVE** is much faster than **MOVE**.

WCLEAR(*ptr*, *n*) zeroes *n* 4-byte words beginning from the address pointed to by *ptr*.

A coroutine scheme is offered by the two coroutine primitives **CRSPAWN** and **CRSWITCH**. The main difference to the conventional scheme of Modula-2 (**NEWPROCESS** and **TRANSFER**) is that coroutines declare themselves to be coroutines. This allows parametrized coroutines.

CRSPAWN(*newcr*) performs following steps:

- Creation of a new stack. The stack is allocated by *SysStorage*, the initial stack size depends on the size of the activation record of the calling procedure and *Coroutines.defaultsize*. An optional second parameter of **CRSPAWN** allows to override the default of *Coroutines.defaultsize*. Coroutines stacks are allowed to grow as long there is enough memory and address space for them. See *SysStorage* and *Memory* for details.
- The coroutine structure is allocated at the beginning of the new stack and initialized. The coroutine structure contains information about the stack management registers (pointers to the activation record) and the program counter of the coroutine. The program counter of the coroutine is set to the instruction after the call of **CRSPAWN**.
- The activation record (parameters and local variables) of the procedure calling **CRSPAWN** is copied to the new stack.
- Immediate return to the calling procedure. Thus the coroutine procedure must not be a function.

The *newcr* parameter is of type **COROUTINE** and points to the coroutine structure. The reference to the coroutine structure remains valid as long the coroutine exists. **CRSPAWN** returns like **CRSWITCH** if the newly created coroutine resumes execution.

CRSWITCH(*dest*) switches the currently active coroutine. The parameter *dest* must be either the result of a previous **CRSPAWN** call or it must equal *Coroutines.main*. After switching, *Coroutines.current* equals *dest* and *Coroutines.source* references the previously active coroutine.

Coroutines point to a structure which contains following components:

```
TYPE
   Coroutine = POINTER TO CoroutineRec;
   CoroutineRec =
      RECORD
         base,                         (* current activation record *)
         top,                          (* top of stack *)
         pc: SYSTEM.UNTRACEDADDRESS; (* current program counter *)
         interrupts: INTEGER;          (* # of saved contexts due to interrupts *)
      END;
```

Coroutines have a type tag which points always to *Coroutines.tag*. The stack of a coroutine may be examined by following the chain of dynamic links. Relative to *base*, the next base pointer is at offset 0, and the return address at 4. The chain is terminated by a dynamic link which equals **NIL**. Static links (in case of nested procedures) are given as implicit parameter at offset 8.

Additional informations are given in *Coroutines* and *Ulm's Coroutine Scheme*.

Type definitions exported by SYSTEM

SYSTEM exports **INT16**, an integer type which ranges between **SHORTINT** and **INTEGER** and occupies two bytes in storage. **INT16** may be used like other integer types; **SHORT** converts **INT16** values to **SHORTINT** and **LONG** converts to **INTEGER**. **INT16** variables or components will not be aligned by the compiler.

To distinguish usual integer values from addresses and for better support by the garbage collector, two address types have been introduced in **SYSTEM**: **ADDRESS** and **UNTRACEDADDRESS**. Both are compatible to each other and to **LONGINT** but the compiler issues a warning when an address is assigned to **LONGINT**. Addresses of type **ADDRESS** (but not of type **UNTRACEDADDRESS**) are part of pointer lists and recognized by the garbage collector. Copying garbage collectors are required to check the validity of addresses (of type **ADDRESS**) and to update them if they point to living objects even if they do not point to the beginning of an object. Garbage collectors are not required to decide whether an object lives or not on base of addresses only, i.e. normal pointers are needed to keep an object alive.

Interface to command line arguments

UNIX stores command line arguments at the beginning of the main stack and lets the stack pointer point to the number of arguments and the **NIL**-terminated lists of arguments and environment parameters. The module *SysArgs* serves as interface and is initialized during the runtime start. *SysArgs.argc* and *SysArgs.envc* are set to the number of arguments (including the command name) and the number of environment parameters. *SysArgs.argv* and *SysArgs.environ* point to the list of arguments and environment parameters, respectively. These pointers may be modified but not the objects they point to. Oberon has no appropriate data type for pointers to character arrays with unknown length. The problem is fixed by using pointers to array types with sufficient length (to avoid range check faults). But this restricts access to the basic elements (i.e. characters). Access of these arrays in their entirety could cause memory faults.

SysArgs is very system dependent. More portable interfaces are provided by *Args*, *UnixArguments* and *UnixEnvironment*.

Dynamic storage allocation

The compiler uses the interface of *Storage* for **NEW** and **SYSTEM.NEW**.

Runtime errors

Some errors are detected at runtime:

error number	called procedure of module *RTErrors*	description
case	CaseError	attempt to find case failed
return	NoReturn	function does not return any value
range	RangeError	array index out of range
typeguard	TypeGuardFailure	

`conversion`	`ConversionError`	`**SHORT**(*val*) failed`
`assertion`	`FailedAssertion`	`**ASSERT**(*assertion*) failed`

Runtime errors cause the associated procedures of *RTErrors* to be called. These procedures are expected to raise events which allow to customize runtime error handling (default reaction is termination with core dump: *Process.Abort*). These procedures must not return.

Author: borchert, last change: 1994/02/15, revision: 1.9, converted to HTML: 1997/05/05

Oberon || Compiler & Tools || Library || Module Index || Search Engine