Reflection in Oberon

Christoph Steindl

Department of Computer Science (System Software)
Johannes Kepler University Linz, Austria
steindl@ssw.uni-linz.ac.at

Abstract. We introduce metaprogramming facilities into the Oberon V4 system. Metaprogramming means that a module can access the structure of other modules (i.e., procedures, types, run-time data) at run time. We discuss how type safety can be enforced in an environment with strong typing. Finally we show how metaprogramming can be used to implement an easy-to-use database interface and conclude with a comparison with other metaprogramming systems for statically-typed programming languages.

1 Introduction

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:

1. What are the field names of a record type T declared in module M?
2. Which procedures are currently active (e.g., when a run-time trap occurs)? What are the names, types and values of their variables?
3. 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 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*.

The notions of metaprogramming and reflection are common and widely used in programming languages like 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] is a modular operating system based on the general-purpose programming language Oberon. It 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.

Metaprogramming and reflection are not widely used in statically typed programming languages, although their use might be beneficial, too. One reason is the need for special language constructs to express the access to meta-level information. The code accessing meta-level information must be type-checked at compile time although it will have to work with arbitrary (yet unknown) types and modules.

We introduce support for metaprogramming and reflection into the Oberon system not via new language constructs, but via a new library. Together with the compiler and the dynamic loading facility of the Oberon system it is possible to construct program fragments at run time, to dynamically load and execute them.

In the next section, we provide background about metaprogramming and reflection. In section 3 we introduce our support for metaprogramming and reflection. In section 4 we present applications that make use of the new facilities. In section 5 we compare our approach with other systems, and present our conclusions in section 6.

2 Background

In this section we explain the controversy between a rigid, but safe type system and a system that allows for metaprogramming. We explain what we understand under introspective, invocational, and intercepting capabilities of a metaprogramming system. Furthermore we show to which amount meta-information is produced by standard Oberon compilers ([Cre90]).

2.1 Roles of the Type System

There is a controversy between a rigid type system and a system that allows for metaprogramming. The first sacrifices flexibility for increased safety, the latter increases flexibility.

The type system of a language plays three roles ([Ste+93]):

- it shall facilitate data modelling,
- it shall help detect and avoid errors in programs at compile time, and

- it shall allow efficiency in code generation.

Static type checking verifies type assertions prior to a program's execution. Strong typing is a little weaker in that it only requires that all programming entities be typed before use and that all use be consistent with the type system. Strong typing ensures that a certain class of errors is cleanly detected and static typing improves efficiency by removing type checks from run-time code. The goal of most type systems is to make checking as static and thus as efficient as possible. However, some type checking cannot be performed statically (especially with object-oriented programming languages). Some programs will be rejected by the type checker as being unsafe although they might execute without errors.

Oberon is a general-purpose programming language in the tradition of Pascal and Modula-2. Its most important features are block structure, modularity, separate compilation, static typing with strong type checking (also across module boundaries), and type extension with type-bound procedures.

Metaprogramming introduces much flexibility into a system: problems that have previously been unsolvable or hard to solve can now be easily solved (e.g., automatic persistence of objects). It allows programs that are more interpretative in their nature, programs that act on arbitrary other programs. Compiling such programs with static type-checking is apparently difficult, if not impossible.

Within a strongly-typed environment, code accessing meta-level information must pass type-checking. Nevertheless meta-programs may have to work with arbitrary types and programs. A generic mechanism must be at hand to access all possible kinds of types and programs (even those that will be created some time in the future).

2.2 Introspective, Invocational, and Intercepting Capabilities

The metaprogramming system should provide introspective, invocational, and intercepting capabilities ([Bra95]).

Introspection allows a program to look into itself, to inspect other programs, and to obtain information about the current run-time state. It does not allow the program to perform any changes.

Invocational capabilities allow a program to explicitly call functionality that is normally hidden in the run-time system, e.g. creation of new objects, dynamic loading, linking, and unloading of compiled code.

Intercepting capabilities allow a running program to change the behaviour of language primitives at run time, e.g., object creation and destruction, method dispatch, and access to simple attributes.

2.3 Reference Information

To implement these capabilities, information that is contained in the symbol table during compilation must be accessible at run time. It must be possible to get information about the type of an object, about its fields, and it must be possible to get information about the parameters and local variables of procedures.

The compiler ([Cre90]) used in many Oberon environments generates an object file and a symbol file out of an Oberon source file using the symbol files of imported modules in order to get type information about imported items and to detect changes to the interface of the imported modules. Furthermore a reference file is generated containing information about the types and procedures defined in the module. The reference file does not exist as an own file but is appended to the object file, which reduces the overall number of files. When a module is loaded, its data and code are loaded into memory, the type descriptors for the types defined in the module are built and the reference information contained in the reference section of the object file is loaded into memory as well.

The structure and contents of the reference section can be considered as a simplified or linearized symbol table. Therefore it contains information about the fields of record types, information about local variables and parameters of procedures, and information about global data. An EBNF grammar describing the contents of the reference section can be found in [StM96].

3 Module Ref

We opted not to extend the programming language Oberon to facilitate metaprogramming but to extend the Oberon system by a new module which supports metaprogramming.

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.

3.1 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 its procedures or record types (see Table 1).

Program data is organised 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. Table 2 shows the organisation of data.

Table 1. Opening Riders

global variables	OpenVars(module, r)	sets r to the first global variable of the module *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) OpenProc(pc, r)	sets r to the first procedure of the module *module*, or to the procedure containing *pc*
record types	OpenTypes(module, r)	sets r to the first record type of the module

Table 2. Organisation of Data

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}.	*and via OpenProc*
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 *)
```

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
    name: ARRAY 32 OF CHAR;
    mode: SHORTINT; (* End .. Type *)
    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) ReadInt (VAR i: INTEGER);
    PROCEDURE (VAR r: Rider) ReadProc (VAR p: ProcVar);
    PROCEDURE (VAR r: Rider) ReadPtr (VAR p: SYSTEM.PTR);
    PROCEDURE (VAR r: Rider) ReadString (VAR str: ARRAY OF CHAR);
    ...
    PROCEDURE (VAR r: Rider) Write (ch: CHAR);
    PROCEDURE (VAR r: Rider) WriteInt (i: INTEGER);
    PROCEDURE (VAR r: Rider) WriteProc (p: ProcVar);
    PROCEDURE (VAR r: Rider) WritePtr (p: SYSTEM.PTR);
    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.
```

Operations

- *OpenVars(mod, r)* sets the rider *r* to the first global variable of module *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*.
- *OpenProcs(mod, r)* sets the rider *r* to the first procedure of module *mod*.
- *OpenTypes(mod, r)* sets the rider *r* to the first record type of module *mod*.
- pc := *PC(mod, name)* returns the absolute start address of the procedure *name* declared in module *mod*.
- *OpenProc(pc, r)* sets the rider *r* to the procedure that contains the (absolute) program counter value *pc*.
- *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*.
- *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.
- *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 Table 3.
- *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.

Table 3. Zooming into Riders

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

4 Applications

We have implemented the following tools using module *Ref*:

- a post-mortem debugger that is invoked when another program terminates with a trap. Its responsibility is to show the machine state in a human-readable form. We show all variables in the same window and 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.
– showing the global variables of a module: As in the original Oberon system, the command System.State opens a viewer displaying the global variables of the specified module with the possibility to zoom into structured variables.
– a heap inspector which displays a bitmap that represents the heap. All blocks of a desired type are coloured red. By clicking on a block, the information contained in the block is displayed. Furthermore information about the number and sizes of objects, the memory space occupied by objects of a specified type, etc. is displayed.
– a general output module which can be used to facilitate simple output.
– a database interface.

In the following we will explain the usage of the database interface in more detail (see also [Ste96b]).

4.1 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 *ESQL* [Ste96a] that provides access to ODBC databases [ODBC94].

```
DEFINITION ESQL;

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 ESQL.
```

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.

4.2 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".

We declare the type *Person* that will be used to represent persons. After opening the connection, we create a table for the persons that we will insert later on. 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.

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

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

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

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; Out.String(person.firstName); Out.Char(" ");
    Out.String(person.lastName); 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 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 the database driver. When processing a "CREATE TABLE" statement, the variables' types are used to generate the expanded SQL statement with a column for each scalar variable (with type and name).

A minor restriction is that the variables specified in an SQL statement must exist when the statement is actually executed. Local variables go out of scope when the procedure returns. Therefore it is not possible to use local variables in an SQL statement and execute the statement at a moment where these variables do no longer exist.

5 Comparison

We support introspective capabilities to a high degree. Via the standard modules *Modules* and *Types*, information about loaded modules and their type descriptors is available. For a description of these modules see [StM96]. We also provide access to global variables of modules, to variables and parameters of procedures, to the currently active procedures, and to the fields of record types. When a rider is positioned and the value over which the rider is positioned shall be read, run-time checks are performed. Therefore we do not lose type safety, but ensure type safety at run time. Up to now, we do not restrict access to the exported interface of the modules but we provide access to all available items, e.g. we allow access to non-exported global variables, likewise we allow access to local variables of procedures. The motivation behind this was to extend the scope of applications (e.g., also to post-mortem debuggers, etc.) which need access to more than only exported variables.

Invocational capabilities are supported by the standard Oberon system to a large degree. New objects can be created via module *Types*, modules can be loaded on demand via module *Modules* (they are automatically linked to the already loaded modules), and modules that are not needed any more can be freed explicitly. A running program can also invoke the compiler to generate new object code, which can then be loaded. The linking loader checks whether the version of the module that shall be loaded is consistent with the versions of the modules that are already loaded. We do not provide a generic mechanism for the invocation of procedures, i.e. it is not possible to call arbitrary procedures. Nevertheless it is possible to read procedure variables via a rider (e.g., *r.ReadProc(p)*). If the parameter list of the procedure is known at compile time, this procedure variable *p* can be type-cast to a procedure variable of the proper type (e.g., *handler := SYSTEM.VAL(Display.Handler, p)*) and the procedure can then be invoked via this procedure variable where the compiler handles the parameter passing (e.g., *handler(f, msg)*). We admit that this is a major restriction, but accept it for the sake of simplicity.

Intercepting capabilities are not fully supported up to now. It is possible to position a rider over data and update the values of the data. Scalar variables, as well as procedure variables and pointer variables can be updated. Run-time checks enforce type compatibility of the value and the memory location that shall be updated. It is not possible to install callback procedures that are called if an object of a given type is created. Up to now patching the method table is also not supported, as this is considered dangerous. It is also not possible, to be notified of access to simple attributes of a record.

There is no run-time overhead if metaprogramming and reflection are not used. Only when meta-information is needed, it is looked up. But even this lookup is very efficient, since the reference information is kept in memory and there is no disk access.

The main source of inspiration for our support for metaprogramming was the dissertation of J. Templ [Tem94]. He implemented a version of the Oberon System with better support for metaprogramming than our extension to the Oberon System. In his metaprogramming protocol he provides generic access to objects in a way similar to ours (actually we provide access to objects in a way similar to his approach). His work inspired us to use iterators to access arbitrary data structures. We unified the iterators to one type (*Rider*) which serves as a *Rider*, *ArrayRider*, *RecordRider*, or *ActivationRider* in his terminology. We also unified the way how the iterators are opened by the *Zoom* operation. Therefore we think to provide a more orthogonal approach but with some restrictions that did not exist in his system. In particular he provides a mechanism to control procedure activations. Arbitrary procedure objects can be evaluated with arbitrary parameters (*PROCEDURE Eval (proc: Procedure; VAR par: Parameters)*). Therefore it is necessary to tag procedures (like ordinary objects in Oberon are tagged, i.e., each object has a reference to its type descriptor). Every procedure has its own parameter record with the procedure specific parameters. Access to these parameters is available via *GetParams(p, params)*. *OpenParams(paramRider, params)* can then be used to iterate over the parameters and set the input parameters accordingly. Finally *Call(proc, params)* calls the procedure *proc* with the parameters *params*. He also introduces the notion of *active procedures*, i.e. procedures which have a message handler installed in order to react to messages sent to them. A generic message handler would simply evaluate the procedure object with the parameters of the message, but more specific message handlers can filter messages, can access and modify parameters before and after the evaluation.

The metaprogramming support in Oberon/F [Pou95] apparently also stems from the dissertation of J. Templ. In contrast to our approach, Oberon/F restricts access to public information, i.e. it does not allow access to non-exported items of a module. They guarantee safety because their metaprogramming support does not allow to change data which is not exported as modifiable. It only allows to do with a module what could also be done by a normal client module - but in a more dynamic way. It allows inspection and modification of data depending on run-time decisions, without static import of the inspected or modified module. In Oberon/F it is not possible to implement a post-mortem debugger building on the metaprogramming support as it is not possible to access the procedure activation stack. Furthermore it is not possible to inspect the variables of procedures or the types defined by a module. As far as we know, Oberon/F only provides access to global data with the possibility to zoom into the data structures at run time.

The meta-level architecture for the BETA language ([Bra95], [MMN93]) uses language extensions so that the compiler can type-check code that exploits the meta-information. But in order to perform the really interesting tasks like automatic persistence of objects, unconstrained attribute references are necessary which cannot be type-checked at compile time. The introspective capabilities seem to be quite powerful and comparable to the capabilities provided by our system. The invocational capabilities include replacement of code objects. We did not support this (which could be done by patching the method table of type descriptors), as we consider this to be dangerous. All the other aspects of invocational capabilities of the BETA meta-level architecture are equally covered by our system. The intercepting capabilities are more powerful than in our system. It is possible to register a callback procedure to trace instantiation of objects of a given type. Tracing of garbage collected objects is also possible. This is supported in our system by the mechanism of finalization (see [Tem94] for more details), where an object can register a procedure that will be called if the object is about to being garbage collected. Furthermore method dispatch and simple attribute access can be intercepted. We did not include this into our metaprogramming system as we consider patching the method table dangerous and because intercepting access to simple attributes can only be done at a high cost. Nevertheless work is going on in the area of safe and transparent remote method invocation (see [Hof96] for details).

CLOS [Kiz+91] includes a comprehensive meta-level interface, which allows the meta-level programmer to inspect and change several primitives of the basic programming language, including redefinition of slot access, multiple inheritance semantics, and replacement of meta-classes. We do not reach the same expressive power as CLOS, but we found many applications where our support for metaprogramming was sufficient.

C++ [Str94] has the concept of pointers to members which are basically offsets of attributes within objects. These pointers to members can be applied to objects to get the value of the member they are declared to point to. However, access to run-time type information is only rudimentary supported via the dynamic_cast and typeid; garbage collection is only rarely used in C++ environments. Therefore we do not consider C++ to encourage reflection.

6 Conclusions

We designed and implemented support for metaprogramming and reflection for the interactive environment of the Oberon system. We made use of dynamic linking and loading, of run-time invocation of the compiler and of the reference information that is generated by the compiler. Our system does not incur any run-time overhead, but still provides for powerful metaprogramming. As there are already tools for profiling, the need for additional intercepting capabilities (interception of method invocation) is not as big as in other systems. Other problems like type-orthogonal persistence ([Kna96]) and distributed object systems ([Hof96]) are currently solved at the department, partly based on the metaprogramming system.

As metaprogramming systems are still rare for strongly-typed systems, we believe that we can contribute that the concepts of metaprogramming are spread among the undergraduate students that have to use our version of the Oberon system in programming courses. We hope that some of them will recognise and exploit the possibilities of metaprogramming and reflection.

Acknowledgements

We wish to thank J. Templ and H. Mössenböck and all other members of the department for fruitful discussions and hints. Further thanks go to the anonymous referees that supplied interesting comments.

References

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

[Bra95] S. Brandt, R.W. Schmidt: The Design of a Meta-Level Architecture for the BETA Language.

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

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

[Hof96] M. Hof: Connecting Oberon. Johannes Kepler University Linz, System Software, Technical Report 7, April 1996.

[Kiz+91] G. Kiczales, J. Rivieres, D. Bobrow: The Art of the Metaobject Protocol. MIT Press, 1991.

[Kna96] M. Knasmüller: Adding Persistence to the Oberon System. Johannes Kepler University Linz, System Software, Technical Report 6, January 1996.

[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.

[Pou95] D. Pountain. *The Oberon/F System*, Byte, January 1995.

[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.

[Ste96a] C. Steindl: Entwurf und Implementierung einer Stücklistenverwaltung mittels einer Client/Server-Datenbank. Diploma thesis, University Linz, 1996.

[Ste96b] C. Steindl: Accessing ODBC Databases from Oberon Programs. Johannes Kepler University Linz, System Software, Technical Report 9, Dezember 1996.

[StM96] C. Steindl, H. Mössenböck: Metaprogramming Facilities in Oberon for Windows and Power Macintosh. Johannes Kepler University Linz, System Software, Technical Report 8, July 1996.

[Ste+93] D. Stemple, R. Morrison, G.N.C. Kirby, R.C.H. Connor: Integrating Reflection, Strong Typing and Static Checking Proc. 16th Australian Computer Science Conference, Brisbane, Australia (1993), pp. 83-92.

[Str94] M.A. Ellis, B. Stroustrup: The Annotated C++ Reference Manual. AT&T Bell Laboratories, Murray Hill, New Jersey, 1994.

[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. Addison-Wesley, 1992.

This article was processed using the LaTeX macro package with LLNCS style