Position Paper, Workshop W5: Object-Oriented Reflection and Meta-level Architectures

Reflection in Oberon

J. Templ
Institut für Computersysteme
ETH Zürich

Abstract

On the attempt of making extensible operating systems more powerful, we investigate the benefits and costs that may result from putting more attention to reflection. As a case study, we try to introduce a reflective architecture into the Oberon system. As Oberon is intended to be a general purpose programming system, we strive for efficiency of the resulting system. It is claimed that the benefits of a reflective architecture will only be honored by a larger community if it is efficiently implemented. Thus we focus on those reflective facilities that can be implemented without affecting the efficiency of non-reflective computation.

0. Introduction

The starting point of our research is the extensible operating system Oberon developed by N. Wirth and J. Gutknecht at ETH Zurich. Actually, the name Oberon refers to both, a programming system and an object-oriented programming language developed for implementing it. Oberon has proved to be useful for many kinds of personal computing such as program development, document preparation, and education. Nevertheless, there are several dimensions for future evolution of the system. One dimension is the user interface aspect, another dimension is to improve the networking capabilities, and a third dimension is the evolution into a reflective architecture. Roughly speaking, these dimensions correspond to the different communication partners of the system. The user interface aspect corresponds to communication of the system with a human user, the networking aspect corresponds to communication with other systems, and the reflectional aspect corresponds to communication of the system with itself. All developments are done using Oberon as a programming environment and using the Oberon (alias Oberon-2) language, where the latter is intended to be left unchanged if ever possible.
Oberon, both as a language and as a programming environment, is intended to be *general purpose*, i.e., it should be applicable for as many different purposes as possible. One such purpose is to implement the Oberon system and compiler itself. Therefore, efficiency is a *conditio sine qua non*. On the other hand, Oberon may also be viewed as a prototyping environment because the edit-compile-link-go cycle is close to that of interpreted systems. However, throw-away prototypes are not the intension of Oberon programming. Instead of, evolutionary prototyping is the favorable programming technique. A program that has been implemented and tested should not have to be rewritten to get a production version as this essentially doubles the effort. If we are going to think about reflective facilities in programming systems, we should keep in mind that general purpose systems must be efficient because inefficiency reduces the range of applicability.
Our approach to introducing reflective facilities is therefore to introduce only those facilities that can be implemented using static compilation without affecting the efficiency of non-reflective

computation. This approach is similar to the way object-oriented concepts have been introduced in conventional languages. The restriction to an efficiently implementable subset opened the door for a widespread industrial use of OO-concepts. It is expected that a similar development will happen with meta-level programming.

1. Reflective aspects already existing in Oberon

According to P. Maes reflection is a matter of degree. There are a number of reflective facilities already available in Oberon, however, Oberon has not been designed with reflection as an explicit goal. Existing reflectional aspects are the list of loaded modules, which can be changed at run-time by loading or unloading modules. Thus, the system is able to reflect on its state (the list of loaded modules) and to extend or shrink itself. Another reflective aspect is the concept of commands which are the atomic actions activated by a user. The command interpreter (the system) reflects on the state of a module (the commands exported by a module) and invokes the procedure, i.e. it changes its own future. In Oberon it is also possible to view the state of global data and a readable stack dump is produced in case of an exception. Variables in Oberon not only have a static but also a dynamic type. Instructions like type-tests provide a sort of introspection on the language level. Last but not least, the fact that Oberon has been implemented using Oberon itself is a noteworthy reflective aspect.

2. Reflective aspects to be introduced

When thinking about a reflectional architecture, one expects reflection to be a major design goal. At first sight, as much reflection as possible should be included. However, there is no free lunch, and compromises have to be accepted in order to allow an efficient implementation of the system. Thus, our primary design goal was to introduce only those reflective facilities that allow to leave non-reflective computation untouched. At least at the moment, meta-computation in a system featuring static compilation is expected to be the exceptional case that should not slow down object-computation.

Reflection on data

First of all, it should be possible to reflect on the state of all objects allocated during program execution, in particular, objects allocated on the heap. This is an extension to Oberon’s type-tests and dynamic binding mechanisms, as it allows meta-operations to be performed on objects without the objects knowing about the operation. The implementation of this feature is relatively easy as Oberon’s type descriptors can be extended to contain all the type information and not just the information that is necessary for type-tests and for the garbage collector. Reflection on dynamically allocated objects makes it possible to implement e.g. persistent objects or to write tools for data structure visualization in a machine independent way. Reflection on the state of global data and (possibly) on the state of the activation stack should be possible using the same (or a similar) mechanism.

The work on persistent objects uncovered the problem of loading and storing procedure variables. We try to solve this problem by introducing procedures as persistent objects collected in a library (a module) and by treating procedure variables as references (pointers) to them. Thus, in the reflective Oberon system programs are made up of objects, viz., procedure objects. As a side effect, this allows us to employ the garbage collector for safe unloading of modules and as a side effect of that, it allows the compiler to generate a little bit more efficient type tests. Without going into details it should be mentioned that this is an example where the system’s performance could

have been improved by introducing more knowledge about the system into the system.
Viewing procedures as objects opens the door for a other useful reflective facilities. According to the notion of libraries and persistent objects as proposed by J. Gutknecht, every object has a library, a reference number, and a procedure associated with it, where the latter defines the behavior of the object (the way an object responds to messages). We can think of a procedure object as responding to an *InvocationMsg* by executing its body. Procedure activations in the form P(param1, param2, ...) may be regarded as a shorthand notation for sending the procedure object an invocation message with param1, param2, ... as message components. This provides a natural way of invoking procedures (with arbitrary parameters) from a command interpreter. The indirection introduced by the *handle* instance variable allows us to change the behavior of an object at run time. Thus, it should also be possible to change the behavior of a procedure object by simply exchanging the handler procedure.

```
TYPE
    Object = POINTER TO ObjectDesc;
    ObjectDesc = RECORD
        lib: Library;
        ref: INTEGER;
        handle: PROCEDURE (o: Object; VAR msg: ObjectMsg)
    END ;

    ObjectMsg = RECORD END ;
    InvocationMsg = RECORD (ObjectMsg)
        param1, param2, ...
    END ;
```

Reflection on procedures

Having the behavior of a procedure object expressed explicitly by the handle instance variable, it is possible to experiment with non-standard behavior, i.e., with behavior that differs from that of the interpreter generated by the compiler. A particularly interesting behavior introducing the notion of meta-procedures is sketched by procedure MetaHandle below. To be a meta-procedure is a dynamic property of an Oberon procedure M with respect to another Oberon procedure P. Meta(P, M) means that M takes the control over the execution of P, or in other words, M is an interpreter for P. There may be more than one meta-procedure for a given object-procedure and a meta-procedure itself may be the object-procedure of other meta-procedures. Initially, each procedure has a default interpreter generated by the compiler (Eval). The meta-procedures are maintained as a cyclic stack by two auxiliary procedures viz., PushMeta (also performs some compatibility checks) and PopMeta. MetaHandle activates the top-most interpreter (if any) and rotates the meta-procedure stack by one position. Thus, the meta-procedures act as a cyclic stack of filter procedures in case of self-invocation. In case no meta-procedures are present, the procedure has the standard behavior.

```
TYPE
    Procedure = POINTER TO ProcedureDesc;
    ProcedureDesc = RECORD (Objects.ObjectDesc)
        meta: Meta
    END ;

    Meta = POINTER TO MetaDesc;
```

```
MetaDesc = RECORD
    link: Meta;
    p: Procedure
END ;

PROCEDURE MetaHandle* (p: Proc; VAR msg: ObjectMsg);
    VAR m: Meta;
BEGIN
    m := p.meta;
    IF (m = NIL) OR (m.p = p) THEN Eval(p, msg)
    ELSE p.meta := m.link; m.p.handle(m.p, msg)
    END ;
    p.meta := m
END MetaHandle;
```

```
PushMeta(P, M); {Meta(P, M)} m := PopMeta(P)
```

The key to an efficient implementation is to optimize the cases where a procedure is invoked from an Oberon program using the shorthand notation P(param1, param2, ...). If P has the standard behavior (P.handle = MetaHandle & P.meta = NIL), static binding can be used and no conversion of the arguments into a message record is required. Since having the standard behavior is a dynamic property, a limited form of runtime compilation has to be used (the first couple of instructions of the procedure's entry code have to be generated at runtime). Without going into details, procedures that have a non-empty meta-procedure stack may still be optimized by avoiding to convert the arguments into a message record and to reconvert them in Eval. Only in case a procedure has a non-standard handler installed, conversion of the arguments into a message and activation of the handler is necessary. Thus, we have three levels of efficiency that inversely correspond to three levels of flexibility.

3. Benefits and Applications of Meta-Procedures

Customization

Existing applications can be customized e.g. by forwarding commands to other commands.
This means that meta-procedures can be used as a sort of alias-mechanism.
Similarly, behaviour may be customized by forwarding methods to other methods.
Note that this customization can be achieved without the customized application or class beeing prepared for it (e.g by resource files or other explicit levels of indirection).

Pre- and Post-conditions

A meta procedure can be used to check the pre- and post-conditions of its object procedure.
There is no need for special language constructs supporting pre- and post-conditions any longer.
Note that this kind of checks can be turned on and off without any recompilations. Thus, the production version of a program is identical to the instrumented version.

Monitoring

A meta-procedure can be used for monitoring its object-procedure.
This removes in many cases the need for dynamic debugging or profiling tools.

Notification

A meta-procedure can be used for notification purposes. As an example a procedure Register that enters a file name into a directory may be customized by a meta-procedure to notify the viewer system about the registration.

Modularization
The availability of meta-procedures may lead to a cleaner modularization of systems by allowing to separate object-level and meta-level computation.

Programmable Software
By means of meta-procedures we get a new generation of software that may be called *programmable software*. Customizations for standard software packages may become a new branch in the software business.

4. The Costs of Meta-Procedures

Runtime efficiency
There is no performance penalty for non-reflective procedures. Using meta-procedures involves a small constant per-level overhead (about 40 cycles).

Storage requirements
At least one additional pointer (meta) is required for all procedures.
A small constant storage is required for each installed meta procedure.
Our current implementation uses 20 Byte for each procedure
in order to treat procedures like other persistent objects.

Comprehension
As long as meta-procedures are not used, the user does not notice any difference.
If meta-procedures are heavily used, they bear the danger of confusion. It seems to be necessary that at least some naming conventions should be observed or that monitoring the meta-state of a system is possible.

Semantics change
Meta-procedures actually change the semantics of the language by means
of library functions. This situation is similar to coroutines, which are also often introduced as library functions.

4. State of the Project

An existing Oberon system running on SPARC-based computers has been extended by full type information (for dynamically allocated objects including procedures) and meta-procedures. The changes in the internal architecture that have been necessary did not increase the overall complexity of the system. Applications that make use of the reflective facilities are currently beeing developed. Among those are a tool for visualization of data structures, a command interpreter that allows activation of arbitrary procedures (not just parameterless ones as in pure Oberon), and a package that supports persistent objects. All these tools are supposed to be fully machine independent, i.e. they are implemented in Oberon without module SYSTEM.

Write.Print lp WS.OORflx.Text\p n