Diss. ETH No 10655

Metaprogramming in Oberon

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Josef Templ, Dipl.-Ing.
Johannes-Kepler-Universität, Linz
born July 24, 1961
citizen of Austria

accepted on the recommendation of
Prof. Dr. H. Mössenböck, examiner
Prof. Dr. N. Wirth, co-examiner

1994

Metaprogramming in Oberon

Josef Templ

Acknowledgements

I want to thank my advisor Prof. H. Mössenböck for a liberal supervision of this project and for his ongoing encouragement and patience. I also wish to thank Prof. N. Wirth for being my co-examiner. His way of thinking and problem solving had a significant impact on this work and is gratefully acknowledged. The Oberon system, developed by Prof. Wirth and Prof. Gutknecht was an excellent working tool and an appropriate base for the work presented in this thesis. My thanks go also to Regis Crelier, Marc Brandis, Jaques Szupcik, Matthias Hausner and Michael Franz who participated in the project of porting Oberon to stock hardware. Regis Crelier provided also the front-end of the portable Oberon compiler. Clemens Szyperski's PhD thesis "On Object-Orientation in Operating Systems" was a rich source of inspiration and provided valuable insights and some starting points for improvements presented in this work. The extensible text editor Write, also developed by Clemens Szyperski and now used as the standard Oberon V4 text editor, has been used to typeset this thesis. Robert Griesemer and Michael Franz proof-read earlier versions of this thesis and provided valuable comments and improvements. I would also like to thank the people from the northern hemisphere of our floor for many stimulating discussions about Oberon System-3. Last but not least, I wish to thank the friends I met in Zurich and my swiss relatives for making my stay in Switzerland an enjoyable experience.

Contents

Abstract

The term *metaprogramming* refers to programming at the level of program interpretation, or in other words, to extending the interpreter of a given programming language in an application-specific way. Traditionally, this concept is available only in dynamically typed and interpreted languages such as *Smalltalk* or *Lisp*. This thesis investigates the possibilities of metaprogramming in a statically typed and efficiently compiled programming language. In the course of a case study, we introduce metaprogramming facilities for the *Oberon* programming language and system.

The result is a variant of the Oberon operating environment which allows a seamless integration of useful meta-level facilities. The key to this integration is a generalized notion of persistent objects and object libraries and its application to components of Oberon programs. Types and procedures are considered to be persistent objects collected in a special kind of library, namely a *module*. We introduce a metaprogramming protocol which allows to manipulate arbitrary data structures based on the notion of *object riders*. An object rider is an iterator which can be used to scan the components of an object and which can be hierarchically refined to structured components down to an arbitrary nesting level. We introduce also facilities for controlling procedure activations based on the notion of *active procedures*. An active procedure is a procedure object which has its own instance specific behavior expressed by a message handler. Active procedures can individually respond to invocation messages and perform any computation as response.

We investigate the implications of this approach with respect to the overall system structure and to the implementation of critical components of the run-time system, such as the library loader and the garbage collector. A new approach to safe library loading and unloading is introduced as well as a simple finalization technique and a way for optimizing libraries with a large number of objects. We show that the integration of metaprogramming facilities does not introduce undue static or dynamic complexity into the Oberon system. A number of realistic applications serve as proof-by-example of the feasibility of the metaprogramming approach.

Kurzfassung

Der Begriff *Metaprogrammierung* bezieht sich auf Programmieren auf der Ebene der Interpretation eines Programmes, oder in anderen Worten, auf das Erweitern des Interpreters einer gegebenen Programmiersprache in einem anwendungsbezogenen Sinn. Traditionellerweise ist dieses Konzept nur in dynamisch typisierten und interpretativ ausgeführten Sprachen, wie zum Beispiel *Smalltalk* oder *Lisp*, vorhanden. Die vorliegende Arbeit untersucht die Möglichkeiten der Metaprogrammierung in einer statisch typisierten und effizient kompilierten Programmiersprache. Im Rahmen einer Fallstudie wird Metaprogrammierung für die Programmiersprache *Oberon* und das *Oberon*-System eingeführt.

Das Ergebnis ist eine Variante des Oberon-Systems, die eine nahtlose Integration von nützlichen Metaprogrammiermöglichkeiten erlaubt. Der Schlüssel für diese Integration liegt in einem verallgemeinerten Begriff von persistenten Objekten und Objektbibliotheken und dessen Anwendung auf Bestandteile von Oberon-Programmen. Typen und Prozeduren werden dabei als persistente Objekte betrachtet, die in einer speziellen Bibliothek, einem *Modul*, gesammelt sind. Auf dieser Grundlage wird ein Metaprogrammierprotokoll eingeführt, das es gestattet, beliebige Datenstrukturen mit Hilfe sogenannter *Objekt-Riders* zu manipulieren. Ein Objekt-Rider ist ein Iterator, mit dem die Komponenten eines Objekts aufgezählt werden können und der für strukturierte Komponenten hierarchisch verfeinert werden kann. Es werden auch Mechanismen zur Kontrolle von Prozeduraufrufen eingeführt, die auf dem Konzept von *aktiven Prozeduren* beruhen. Eine aktive Prozedur ist ein Prozedurobjekt, das sein eigenes Verhalten in Form eines Meldungsinterpreters ausdrückt. Aktive Prozeduren können individuell auf Aufrufmeldungen reagieren und beliebige Berechnungen als Reaktion darauf durchführen.

Die Konsequenzen dieses Ansatzes auf die Systemstruktur im Grossen und auf wichtige Bestandteile des Laufzeitsystems, wie zum Beispiel Bibliotheksverwaltung und automatische Speicherrückgewinnung, werden detailliert untersucht. Für das sichere Laden und Entladen von Bibliotheken, das Finalisieren von Objekten und das Optimieren von Bibliotheken, die aus einer grossen Zahl von Objekten bestehen, werden neue Ansätze vorgestellt. Es wird gezeigt, dass die Integration von Metaprogrammiermöglichkeiten in Oberon keine übermässige Komplexität, weder in Bezug auf die Laufzeit noch auf die Programmierung, nach sich zieht. Einige nicht-triviale Anwendung sollen als Beweis für die Sinnhaftigkeit dieses Ansatzes dienen.

1 Introduction

1.1 Motivation

The motivation for the work presented in this thesis stems from the observation that there are programming tasks which cannot be accomplished in conventional general-purpose programming systems. Even modern languages such as Modula-3 or Oberon with object-oriented facilities and polymorphism fail when applied to a certain class of problems. This class can be characterized as programming tasks that happen at the level of language interpretation, i.e. on the level behind (=meta) the user programs. Consider for example the task of writing a program that linearizes an arbitrary data structure and maps it from main memory to non-volatile storage. This program has to know about the internal representation of the data structure in order to perform the mapping without requiring auxiliary type-specific code. A similar problem is the visualization of data structures – again without type-specific code – which may be useful for documentation or debugging purposes. Another example is to implement a package which allows to call procedures remotely (RPCs) or to dynamically check pre- and postconditions of procedure activations. All these examples involve some code that participates in the program (and data) interpretation, or in other words, that extends the language interpreter. Of course, such extensions should be written at a reasonably high level of abstraction. They should not depend on hardware details or internal system data structures and – in the case of a general-purpose programming system – they should be efficiently executable and they should not affect the effciency of traditional programs.

Such demanding tasks have been a matter of research in the Lisp and Smalltalk communities for about 10 years now. The outcome of these efforts may be identified by the keywords *Metaprogramming* and *Computational Reflection*. The impact of this work on other fields of computer science, however, has been modest. This is probably due to different priorities guiding programming language research. The artificial intelligence community on the one side was primarily interested in expressive and flexible languages. Efficient implementation was of secondary importance. General-purpose programming languages on the other side have to be as efficient as possible. A general-purpose programming system should for example be applicable to implementing itself, i.e. it should be possible to write a compiler and an operating system with it.

1.2 Goals

The purpose of this work is to investigate the possibilities of metaprogramming in a general-purpose programming system that is based on static compilation (such as Pascal or C) rather than on interpretation or dynamic compilation (such as Smalltalk or Lisp). We try to show that it is possible to introduce useful metaprogramming facilities without introducing undue complexity. As outcome of this work, we have created a programming system that allows to write programs which take part in the language interpretation or in other words, we have created an extensible programming system.

As a general rule, we follow the principle *don't use, don't lose*, i.e. we focus on those meta-level facilities that have no or almost no impact on the efficiency of traditional programming tasks. We are also interested in finding a way to integrate metaprogramming facilities seamlessly into the overall structure of a programming system. Furthermore, we want to inquire into the implementation of vital parts of the system such as the module loader and the garbage collector. We try to show that it is possible to combine the introduction of metaprogramming facilities with the elimination of weaknesses of traditional programming systems. Since we notice that previous work in this area has focused on dynamically-typed languages, we are also interested in finding out if and where static typing gets into the way of metaprogramming.

1.3 Outline

Chapter 2 gives an introduction into the subject of metaprogramming. Basic terminology is introduced and history and related work in this area are sketched.

Chapter 3 describes the main concepts of a state-of-the-art general-purpose programming system (Oberon) that has been used as the basis of this work. Existing and lacking metaprogramming facilities in Oberon are discussed.

Chapters 4, 5, and 6 contain the essence of this dissertation. Chapter 4 deals with integration of metaprogramming into a general-purpose programming system. As a case study, a meta-level architecture for Oberon is introduced and the introduced facilities are specified by means of a metaprogramming protocol.

Chapter 5 describes implementation aspects of the introduced meta-level facilities and of vital parts of the Oberon run-time system such as the module loader and the garbage collector. The implementation description is kept target machine independent wherever possible.

Chapter 6 describes several realistic applications of the introduced metaprogramming facilities and compares them to traditional approaches. The emphasis is not put on outstanding sophistication of individual examples but on showing what can be achieved in principle by metaprogramming and what our metaprograms look like. Thus, Chapter 6 serves also to deepen the understanding of the metaprogramming protocol introduced in Chapter 4.

Finally, Chapter 7 summarizes what has been achieved, outlines some areas for future research and concludes the thesis.

2 Meta-level Programming

This chapter presents an introduction into the field of meta-level programming. Besides some historical notes, basic terminology is introduced and previous work in this field is discussed.

2.1 Historical Notes

The very beginning of meta-level programming can be traced back to the design of the EDVAC computer by Mauchly & Eckert, which was described in a famous draft paper by John von Neumann [Neu45]. The new concept in the EDVAC's design was that data and programs were stored in the same memory. This turned programs into data that could be manipulated at run time. First applications of this concept were simple relocation tasks performed at load time. Interestingly, Mauchly & Eckert pointed out that they would have designed earlier computers the same way, but memory was too small to be (ab)used for programs. The idea of using the gained flexibility in dynamic algorithms (self modifying code) was present since these early days of computing and has been investigated by J. v. Neumann and others. It soon turned out that arbitrary self modification should be avoided because it lacks a mathematical foundation and is difficult if not impossible to understand. Nevertheless, the possibility to modify code at run time has become standard practice for special purposes such as dynamic loading, dynamic compilation, or dynamic debugging. The involved code modifications are, however, not really self modifications but modifications initiated outside the modified program by the loader, the compiler, or the debugger. Generally speaking, they take place on a meta-level leading to the term metaprogramming.

Another influencing idea was the concept of frame languages proposed by M. Minsky for knowledge representation [Min74]. The novel point in this class of languages was that data was represented in units called frames that not only contained information about the problem domain but also knowledge about the data being stored. By providing access to this meta-information, programs could be written that not only delt with knowledge but also with knowledge about knowledge.

Probably the most influencial (or at least the most often cited) work in the field of metaprogramming and computational reflection has been carried out by B. Smith [Smi82]. His work not only introduced a reflective variant of Lisp (3-Lisp) but also many of the terms which are now in common use (c.f. 2.2). Subsequently, [Maes 87] combined the work done by the knowledge representation people and the Lisp community and pointed out the close relationship between reflective architectures and object-oriented programming. Maes' design of a reflective language (3-KRS) introduced the notion of metaobjects and is a predecessor of the Common Lisp Object system (CLOS) [KRB91].

2.2 Terminology

This section introduces the basic terminology used throughout this work. It starts with rather simple terms (e.g. computational system) which are nevertheless necessary to introduce more advanced concepts such as metaprogramming and reflective computation. Most of the terminology has been introduced by [Smi82] and [Maes87] and is now generally agreed upon in the metaprogramming community.

Computational Systems

A *computational system* (also called a *system* for short) is something that reasons about or acts upon some part of the world, called the *domain* of the system. A computational system represents its domain in form of *data*. Its *program* prescribes how these data should be manipulated. It dictates the actions that can or must be taken in order to reason about or act upon the domain in a way that complies with the semantics of the domain, i.e. with the relations and properties of entities which hold in the domain.

Causal Connection

A computational system is said to be *causally connected* to its domain if the system and its domain are linked in such a way that if one changes, the other changes too. A causal connection has to be set up only once and is maintained by the system without any further actions. An example of causal connection is a system controlling a robot arm, where the position of the robot arm is known to the system and may also be changed by the system. An example of a system that is not causally connected is a data base of music records, where the insertion or deletion of a record does not change the physical collection of records.

Reason about / act upon

A computational system is said to *act upon* its domain if it is causally connected to it. Otherwise it is said to only *reason about* it. Consequently, to act upon implies to reason about.

Meta-system

A computational system is called a *meta-system* if it has as its domain another computational system, called its *object-system* or *base-system*. Thus a meta-system is a system that reasons about and/or acts upon another computational system. A meta-system has a representation of its object-system in its data. Its program specifies meta-computation about the object-system and is therefore called a metaprogram. The meta-computation returns new information about the object-system, i.e. reasons about the object-system or actually acts upon the object-system.

Metaprogramming

The activity of designing and implementing a meta-system is called *meta-level programming* or *metaprogramming* for short. It is the same as object-level programming except for the fact that the domain of the system is another computational system.

Reflective Systems

A computational system is called a reflective system if it is a causally connected meta-system that has as object-system itself.

Reification

Reification is the process of making something explicit and thus available. A *reifier* is an object which is used to reify something, i.e. to make something explicit. An example of reification is to make the run-time data structures of an interpreter (such as type descriptors or the run-time stack) available for the meta-level programmer.

Meta-level Architecture

A programming environment has a *meta-level architecture* if it has an architecture which supports meta-level programming. This definition deviates from the literature [Maes87], which excludes reflective computation from a meta-level architecture. We felt that excluding reflection is not consistent with regarding a reflective system as a special case of a meta-system.

The activity of implementing a meta-level architecture should not be mixed up with metaprogramming. The latter essentially means to make use of the former. In this thesis, we shall describe the design and implementation of a meta-level architecture in Chapter 4 and 5 and make use of it in Chapter 6.

Two-level Architecture

A meta-level architecture which presents different languages for object-computation and meta-computation is called a two-level architecture.

Reflective Architecture

A programming language is said to have a *reflective architecture* if it recognizes reflection as an important structuring concept and provides building blocks for reflection. Thus, all computational systems implemented in such a language have access to a causally connected representation of (parts of) themselves and can make use of reflection. As reflection can also be applied to meta-level programs, there is no statically fixed number of meta-levels. This phenomenon is commonly called *reflective tower*.

Metalinguistic Abstractions

Given a language that is executable on a computer, one can construct in that language an interpreter for a second language, which is called an *embedded* language. Constructing embedded languages is a powerfull abstraction tool, which is called *metalinguistic abstraction* [AS85]. A simple form of embedded languages occurs whenever one designs a module interface which constitutes a special-purpose language embedded in the implementation language. Meta-linguistic abstractions also occur when a special language or notation is invented in order to cope with the complexity of a given problem. As an example, T-diagrams [Bra61], which are used for describing the bootstrapping steps of a compiler, were introduced because a textual description of the boot-strapping is not the most effective form of communication. The T-diagram language, which is based on a graphical notation, is described by an english text.

Meta-circularity

An interpreter which is written in the same language it evaluates is called *meta-circular*.

Fig. 2.1 outlines a very general programming system and visualizes some of the introduced terms. Note that the notion of compilation does not exist in this model. In this context, compilation is considered an implementation technique (compile and go) rather than a concept on its own.

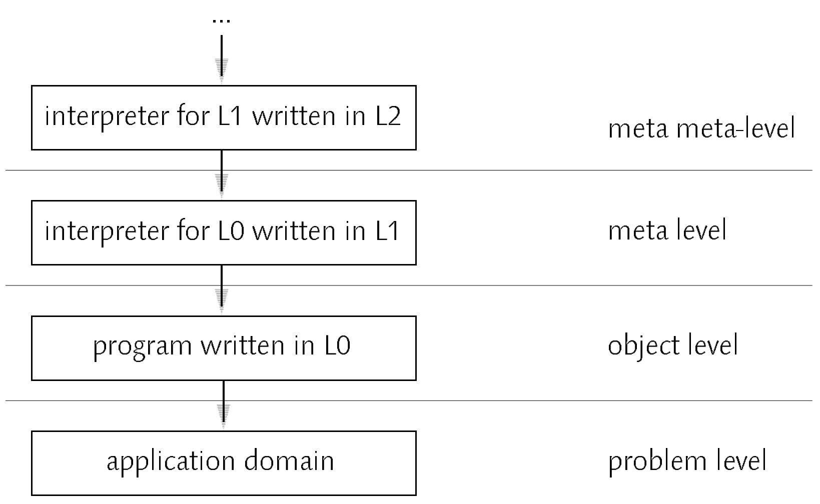


Fig. 2.1 – Levels of Activities

2.3 Conflicting Terminology

Two usages of the term metaprogramming that differ considerably from the definition used in this thesis have been found in the literature. One is the usage of the term metaprogramming in connection with project management, the other is the usage of metaprogramming in the sense of distributed programming. Both definitions are discussed below.

Metaprogramming: A Software Method

[Sim77] defines the term metaprogram as follows: "Metaprograms are informal, written communications from the metaprogrammer, who creates the local language, to the technicians who learn it and actually write the programs. Metaprograms are characterized more by their purpose than by any specific form or syntax".

The idea behind this definition is that in a group of programmers there should be one distinguished person (the metaprogrammer) who defines the rules for programming. Writing those rules down is considered an act of metaprogramming. For example, the metaprogrammer can require some indentation or naming conventions, or that comments are to be written in German. Other examples are that programs should be designed following a certain design method or that software quality should be verified by using a certain quality assurance technique.

A possible criticism of this definition is, in our oppinion, that it is in contradiction to the common understanding of programming, viz. to formulate an algorithm in a machine readable form. In this sense, the task of the manager may be at a metalevel, but it is certainly not programming.

Metacomputer

A metacomputer as defined in [SC92] is a computer that consists of components linked together via some network. For example, all computers connected via the internet may be regarded as one metacomputer. [SC92] also defines a personal computer as a special sort of metacomputer (a mini-metacomputer) if it consists of independent processing units such as an integer arithmetic unit, a floating point unit, a memory management unit and/or a graphics coprocessor. All these units are connected via some links (e.g. a bus) and act together in a way that makes them appear as one metacomputer rather than many individual processing units. A meta-application according to [SC92] is an application that makes use of a metacomputer and metaprogramming is the activity of creating a meta-application.

It is our oppinion that the term *distributed programming* already covers this sort of computation very well, so there is hardly any need to invent another term for it. There is also hardly any computer system that is not at least a minimetacomputer. Thus, the term metacomputer does not provide much information.

2.4 Previous Work

This section describes previous work in the area of metaprogramming. We focus on two language families, the Smalltalk and the Lisp family, which both served as vehicles for research in metaprogramming. Smalltalk is a purely object-oriented language, i.e. a language where every action is described as a message sent to some object. Lisp is an expression-oriented language, i.e. a language where every action is described by an expression (with possible side effects). Both language families evolved over several generations, the later ones trying to remove weaknesses of the earlier ones. In the following, special emphasis is put on a comparison of the different generations with respect to metaprogramming. Besides Smalltalk and Lisp, we take also a short look at the design of a programming environment for the Beta programming language and at a meta-information protocol for C++.

2.4.1 The Smalltalk Family

Smalltalk-80

Smalltalk-80 (or Smalltalk for short) was one of the first languages which offered a considerable amount of meta-level facilities by means of so called metaclasses [GR83]. In Smalltalk, everything is an object. Objects which are able to generate other objects are called classes. Classes which are able to generate classes are called metaclasses. Objects which are generated by a class are called instances of that class. Looking at integer values, the is-instance-of relationship is

42 –> SmallInteger –> SmallIntegerclass –> Metaclass <–> Metaclassclass

where *a* –> *b* means that *a* is an instance of *b*. Metaclasses have been introduced in Smalltalk-80 mainly to handle class variables and class methods uniformly with instance variables and instance methods. Earlier versions, such as Smalltalk-76 used only one common metaclass for all classes, which implied the restriction that all classes had exactly the same class features. For example, it was not possible to have class-specific arguments for the *new* method, which are sometimes usefull for initializing a newly generated object. With an individual metaclass for each class, this restriction is removed. Metaclasses have also been used for various Smalltalk-80 system tools like the class browser or the inspector. For these meta-level tools some high-level form of accessing the system state must be provided in order to avoid dependence on low-level implementation details.

In retrospect, using metaclasses for two different tasks seems to be a questionable design decision. The one task, introducing class methods and class variables, acts on the object-level, the other task, providing access to meta-information, acts on the meta-level. The user community Smalltalk was aiming for, most notably children, were not able to clearly separate these two usages of metaclasses. There were also a number of irregularities in the subtype-of and instance-of hierachies that complicated understanding of Smalltalk-80 even for professional users. The most obvious one is the top of the is-instance-of hierarchy. In Smalltalk it is represented by two different classes (Metaclass and Metaclassclass) acting as there mutually respective metaclasses. Such constructs can probably only be understood if one knows how Smalltalk systems are bootstrapped. Hiding the existence of metaclasses to the object-level programmer required special tools that split a class definition into a class and a metaclass definition and generated a metaclass for every class implicitly. This automatism also established a sort of mysterium around the metaclass facility.

Like Simula-67, which introduced most of the object-oriented concepts without recognizing it as a new programming paradigm, Smalltalk-80 introduced metaprogramming without paying much attention to it. However, the problems introduced by Smalltalk-80's twofold usage of metaclasses have been addressed by two recent language designs in two different ways.

Self

The programming language Self [US87] avoids the need for metaclasses by removing the notion of a class from the language. Self objects are self describing and may serve as *prototypes* for the creation of new objects (cloning). An object consists of a set of slots, which are references to other objects and may inherit from any other object. An object's self-description is not available as first-class object but only via so-called *mirror* objects. A mirror is a vector of slot descriptions that reifys the state of the mirrored object. The creation of mirrors is handled by predefined run-time routines that access an object's self-description. In current Self implementations, this self-description is represented by data structures (so-called maps) shared among objects cloned from the same prototype. Although invisible to the programmer, this shared data plays a vital role for space and run-time efficiency if executing Self programs. Actually, a map can be regarded as an *invisible class*. The effect of Smalltalk's class variables and class methods is obtained in Self by inheritance of state in addition to inheritance of behavior. Inheritance of state is made possible by the delegation-based inheritance mechanism employed by Self. An object may delegate a message to its parent object without changing the identity of the receiver. Method lookup therefore always starts at the original receiver of a message even in case of self invocation in an inherited method. Although it can be shown that this behavior can be used to express the effect of class variables and methods [Stein87], it requires to follow some programming conventions in order to organize objects and inheritance relations properly. [UCCH91] describes a technique where so-called traits-objects play the role of classes.

Cecil

Designed as a successor of Self, Cecil [Cha93] takes a different approach by unifying the is-instance-of and inherits-from relationships. A new object can be defined as being derived from one or more anchestor objects. There are no prototypes that are cloned in order to generate new objects. The effect of class variables is obtained in Cecil by distinguishing between *copy-down* and *shared* object fields. This means that objects serve as building plans for other objects.

In this aspect Cecil is closer to class-based languages and, in some sense, might be regarded as a class-only rather than as a class-less language.

In summary, the problems introduced by Smalltalk-80's twofold usage of metaclasses have been circumvented in Self by simply removing classes (and thereby also metaclasses) as first-class objects. In Cecil, the is-instance-of hierarchy has not been dropped but merged with the inheritance hierarchy.

2.4.2 The Lisp Family

Lisp

The programming language Lisp [McCar60] unified the representation of data and programs by treating both as list structures. In Lisp it only depends on the interpretation of a given list whether it acts as data or as program. As a consequence, it is possible to write programs that compute and return other programs (higher order functions). Lisp is a dynamically-typed language, which means that each value carries with it a type tag that describes the kind of the value. Predefined functions can be used to test the type of a value at run time in order to invoke type-specific code. Due to Lisp's highly dynamic structure, Lisp interpreters make heavy use of dynamically allocated storage. In order to speed up execution and to guarantee memory consistency, many garbage collection techniques have been pioneered by the Lisp community. It should be noted that the implicit type tags are important prerequisites for garbage collection. The author believes that it was the advanced memory management technology and not the syntax which attracted many Lisp users. Programming language researchers were mainly attracted by the simplicity of Lisp, which allows experimentation with different language variants without too much programming overhead. It is for example very easy to write an interpreter for a new Lisp dialect in an existing Lisp version. Therefore, it is not surprising that for Lisp more dialects evolved over time than for any other programming language. Finally, experimentation with language variants and implementing language extensions have been elevated to a concept which is manifested in the Common Lisp Object System (CLOS). An important intermediate step, however, was the work on a reflective variant of Lisp called 3-Lisp.

3-Lisp

The unique feature of 3-Lisp [Smith 82, RS84] is the possibility to define reflective functions in addition to normal (non-reflective) Lisp functions. Execution of a reflective function brings the computation to the level where the interpreter of the current computation is run. Reflective functions always take two additional parameters (*env* and *cont*) which are causally connected to the state of the interpreter. Env is a list of variable bindings (the environment) and cont is a function that specifies the continuation (the run-time stack) of the object-level computation. Both parameters may be inspected and/or changed by a reflective function. Reflective functions are themselves implemented in 3-Lisp, which means that 3-Lisp is based on a meta-circular interpreter. Therefore reflection is not restricted to one level, but may also occur on the meta-level and the level above the meta-level and so forth. Some ideas of 3-Lisp, especially the idea of making continuations explicit, have also shown up in other Lisp dialects such as Scheme, but, due to efficiency considerations, in a restricted form only.

Examples of meaningful usage of reflective functions are the introduction of non-standard control flow as in the case of coroutines or exception handling. Reflective facilities allow to omit these kinds of concepts from a core language and to introduce them later on by user programs. There is no (known) possibility to introduce such features in a clean way without reflective facilities. This can be seen e.g. if one tries to introduce coroutines in a traditional system based on static compilation. In this case the procedure activation stack has to be manipulated, which breaks the abstractions of any high level language since the activation stack belongs to the interpreter (the processor) and not to the user program. Reflective languages try to reify the whole or at least parts of the state of the interpreter and make it available to reflective programs.

CLOS – The Common Lisp Object System

Experimentation with object-oriented programming in Lisp led to many slightly different dialects which are an inherent problem for the portability of object-oriented Lisp programs. To overcome these problems, CLOS [Ste90] has been designed as an open language, i.e. CLOS is not a single point in the language design space but subsumes a whole range of possibilities. The main concept behind the design of CLOS is to base the semantics of important and orthogonal language constructs on so-called meta-objects. A meta-object defines the particular semantics of a language construct by executing its associated methods. There exist default meta-objects which define a default semantics of CLOS, but additional meta-objects may be added as CLOS programs. The designers of CLOS point out that they invested great care into the performance of CLOS programs [KRB91] by a proper design of the meta-object protocol (mop). The main idea is that CLOS programs should be compilable and that the efficiency of the compiler has no effect on the efficiency of the compiled program. Hence, the meta-object protocol should only be used at compile time but not at run time. [HDKP92], however, points out that CLOS did not really succeed to reach this goal because accessing for example a data slot, which is a very time critical operation, involves a dynamic protocol, i.e. one that must be executed at run time. The involved overhead is at least one normal plus two generic function calls and a table lookup. The overhead is not inherently coupled with the idea of meta-objects, but due to the particular design of the CLOS mop. Static slot access protocols such as the one described in [HDKP92] are feasible without restricting the flexibility.

CLOS is meta-circular, since programming at the object-level and programming at the meta-level are both done in CLOS. The meta-object protocol is based on five different kinds of meta-objects which may be specified as optional parameters to the special forms that introduce classes, generic functions and methods. The five meta-object classes are specializer, generic-function, slot-definition, method and method-combination. Specializer meta-objects serve as spezializers of generic functions. Unlike most other object-oriented languages, CLOS is not a single-receiver language, but allows specialization on any number of parameters. Consequently, activation of a dynamically bound procedure is not called message send but generic function call. An important kind of specializers are class meta-objects which are used to describe the properties of instances. In contrast to Smalltalk-80, CLOS does not use class meta-objects to introduce class variables or class procedures. Special class meta-objects may be used, for instance, to specify a certain inheritance priority which in case of multiple inheritance differs in the various object-oriented Lisp dialects. Generic-function meta-objects specify the properties of generic functions, which play the role of multiply dispatched methods in CLOS. An important property of generic-function meta-objects is the argument precedence list, which is used to resolve ambiguities in generic function calls, i.e. if it is not possible to determine the most specific method uniquely. Again, this can be used to emulate the various generic function call strategies of different object-oriented Lisp dialects within CLOS. Slot-definition meta-objects contain information about the definition of a slot (a data field). This information includes whether a slot is shared among all instances of a class or not, how to access a slot via reader and writer functions, the location (offset) of the slot and much more. Slot-definition meta-objects may be used to intercept access to slots for example to interface CLOS objects with the non-CLOS world. An example of using this mechanism for introducing persistent objects in CLOS has been described in [PA90]. Method and method-combination meta-objects are used to specify the properties of methods and the order of execution in case of optional pre- or post processing methods.

2.4.3 Beta

The purely object-oriented programming language Beta [MMN92] is a descendant of Simula-67, the language that pioneered the object-oriented programming paradigm. Beta is aiming at a maximum of regularity by introducing the term *pattern* for anything that encapsulates state and behavior. Not only objects are denoted by patterns but also modules, procedures and types. Beta is implemented with an object-oriented metaprogramming system called *Yggdrasil*. Metaprograms in the Beta implementation are for example the syntax directed editor, the compiler and browsing tools. The whole Beta programming environment can be considered as a collection of metaprograms that all work on one of three levels of a common representation of Beta programs. The lowest level is the tree level, which can be compared to S-expressions in Lisp. Beta programs at this level are represented as a tree structure without obeying any syntactic or semantic requirements. This level is useful for any kind of table-driven tool that is used for Beta or any other language in the same environment. A higher level of abstraction is provided by the context free level, which obeys the context free syntax of Beta. Yggdrasil is actually grammar based, i.e. any language that is specified by a context free grammar can be accessed at an according context free level. The top level is the semantic level, which may be used to add semantic attributes to the abstract syntax tree. This level is tool-dependent and usually reflects context sensitive aspects of the language. In the case of a compiler, the semantic attributes may for instance be the storage class of an expression.

2.4.4 C++

Despite its complexity, the programming language C++ [Str91] in its current version (2.1) completely lacks all kinds of metaprogramming facilities. In many cases C++ programmers have introduced the missing functionality by means of programming conventions, preprocessors, libraries, or even by modifying the compiler. An ANSI standardization working group is now actively discussing a way to introduce meta-level facilities in C++. One of the proposals which are considered to be standardized is based on a multi-layer approach [BKPS92]. The first layer of meta-facilities allows access to type information of an object. In particular, it allows type comparison which can be used for safe (i.e. run-time checked) down-casts. Layer two provides access to information about inheritance relationships of classes. Layer three provides low-level information about relative addresses of instance variables in order to support object input/output, and finally, layer four provides detailed information about instance methods, class methods, method parameters and friend functions. The meta-information protocol provides also information about built-in types such as int or float.

One of the intricate problems of even only introducing layer 1 (type tags) into the C++ standard is that compatibility with existing programs written in pure C or with already compiled C++ libraries should not be affected. The additional type information, however, has to be stored within an object, which changes the storage layout. Note that the vtable pointer (the pointer to the virtual function table) of standard C++ implementations can hardly be used to hold the type information since it does not have a fixed position within all objects and might be missing at all for objects of certain classes.

2.5 Related Fields

Since programming paradigms are usually not defined in a mathematically precise way, one has to cope with grey areas when talking about paradigms. We discuss several such grey fields with respect to the meta-level programming paradigm.

Compiler Construction

A compiler transforms a program from one programming language into a semantically equivalent formulation in another language. It has as its domain other programs, hence it can be called a meta-level program. However, the field of compiler construction is usually not considered to fall into the meta-level programming paradigm. The reasons are that traditional compilers are not built on top of a meta-level architecture and they also don't introduce one, i.e. they don't provide a framework for possible language extensions. A compiler *implements* a language, it does *not extend* it. An exception might be the Beta programming system, where the compiler actually is a meta-level program which makes heavy use of the underlying meta-level architecture.

Partial Evaluation

Partial Evaluation is a technique which is sometimes used for the construction of optimizing compilers or static program checkers. It essentially means the specialization of expressions (or procedures) by replacing variables by constants or by reducing the range of variables. Partial Evaluation has other programs as its domain but it is not a typical meta-level program for the same reasons as mentioned for compilers.

Interpreters

An Interpreter takes a program as input and produces the output of this program by interpreting it. Interpreters are acting on a meta level, but again, they are not typical examples for metaprogramming since they implement a language at the first place, they don't extend it. If implemented on top of a meta-level architecture, an interpreter could of course also be a more typical metaprogram. A Lisp interpreter written in Lisp, for instance, could make use of Lisp's built in meta-level facilities and must therefore be considered a very typical meta-level program. Actually, this was and is the most common way to experiment with Lisp dialects.

Higher Order Functions

A Higher Order Function is a function which takes one or more functions as input parameters and/or returns a function as result. The possibility to pass functions as parameters must be built into the programming language in use. Higher order functions make use of advanced language constructs. However, they don't extend the language interpreter. Therefore, they should not be considered to be meta-level.

Recursion vs. Reflection

Recursion means to refer to something of the same kind, usually of a smaller size. For example, *factorial*(n) might be partially defined as $n \ast$ *factorial*($n - 1$). There is no self reference to the term *factorial*, but only a reference from one instance (*factorial*(n)) to another (smaller) instance (*factorial*($n - 1$)). When looking at the interpretation of recursive functions, this fact can be seen easily by the existence of an activation frame for every recursive function call. Reflection, on the other hand, means the existence of a self reference. For example, the sentence "This sentence has property x." refers to itself, not to any other instance of the same kind.

3 The Oberon Language and System

The name Oberon refers to both a modular operating system and a hybrid object-oriented programming language [WG92]. Oberon has been chosen as the basis for our case study on metaprogramming for various reasons. First, the programming language Oberon reflects the state-of-the-art in software engineering by not only providing facilities for programming in the small and programming in the large, but by supporting also the concept of extensible software. This was previously the domain of interpreted, dynamically typed languages aiming more at rapid prototyping than at programming industrial strength software. A second reason was that Oberon is both a language and a supporting environment. Experiments with metaprogramming require modifications of both the compiler and the run-time system. The simplicity of Oberon promised to enable such modifications within reasonable time bounds. Third, the implementation of the Oberon language and system are available in source form and last but not least, the author was familiar with a particular implementation of Oberon [Te91]. The following sections give a short survey of the Oberon programming language and operating environment as far as it relates to this thesis. Existing metaprogramming facilities will be discussed in detail.

3.1 Language

Oberon [RW92] is a general purpose programming language in the tradition of Pascal and Modula-2. It combines the well proven type system and module concept of its ancestors with the new concept of record extensions in a seamless way. Additional improvements such as basic string operations and numeric type inclusion make the language more convenient to use. Oberon also removes some features known from Pascal or Modula-2, most notably variant records, nested modules, subranges and enumerations. Variant records are fully replaced by record extension. Nested modules in Modula-2 have been used rarely and complicate the language semantics. In many cases, subranges and enumerations have added more to the verbosity of a program than to its readability and they also don't fit nicely into the numeric type inclusion hierarchy. With the principal new concept of record extension, Oberon aims at extensible programs. Any extensible component of a system can be expressed in the type system as an extensible record type. Dynamic binding of messages to methods is expressed by procedure variables or type-bound procedures. Programming in Oberon means to extend a given system (e.g. the Oberon operating system, an extensible text editor, ...) by new components. Of course, by ignoring extensibility it is in principle also possible to write more traditional stand-alone programs by statically linking all necessary modules into a sealed object file although this is not the way Oberon programs are intended to work.

The programming language Oberon is designed to be efficiently compilable into native machine instructions. There is, however, also a run-time system involved in the execution of Oberon programs, which maintains the illusion of infinite heap space by means of automatic garbage collection. Taking the burden of memory management from the programmer is an invaluable improvement in program robustness since fatal errors such as dangling pointers simply cannot occur any more. Moreover, automatic garbage collection is a necessity in the case of extensible systems since a programmer is not able to keep track of all references to data structures which are introduced as extensions possibly years later. It is to a large degree the garbage collection argument why the author believes that Oberon as a language will eventually replace other widespread statically typed languages such as Pascal, Modula-2 or C and therefore research based on Oberon is not only an academic exercise.

In the following chapters, the concept of record type extension and the various kinds of dynamic binding in Oberon will be used regularily. To keep this work self-contained, these features will be shortly outlined below.

```
TYPE
   Object = POINTER TO ObjectDesc;
   ObjectDesc = RECORD object fields END;
   SpecializedObject = POINTER TO SpecializedObjectDesc;
   SpecializedObjectDesc = RECORD (Object) additional fields END ;
```

A record type may extend another record type and introduce additional fields. The type *SpecializedObject* is said to be a direct extension of type *Object*, which is the direct base type of *SpecializedObject*. An extended type inherits all fields of its base type and is therefore upward compatible with it. Roughly speaking, anything that can be done with the base type can also be done with the extended type, but not vice versa.

Dynamic binding in its most flexible form is expressed using a procedure field and extensible message records for sending arbitrary messages to an object. The following type declaration introduces objects that encapsulate not only state but also behavior by means of a message handler that takes the receiver of a message and the message itself as parameters. This style of object-oriented programming is sometimes referred to as *instance-centered* as opposed to *class-centered*, since every instance may have its own individual behavior.

```
TYPE
   ObjMsg = RECORD END ;
   Object = POINTER TO ObjectDesc;
   ObjectDesc = RECORD
      object fields
      Handle: PROCEDURE (o: Object; VAR msg: ObjMsg)
   END;
```

An example of a concrete message is the request to an object to generate a copy of itself.

```
   CopyMsg = RECORD (ObjMsg) to: Object; deep: BOOLEAN END ;
```

The message handler which is installed in an object can distinguish explicitly between different kinds of messages by the boolean type-test operator IS. *o* IS *T* yields TRUE if and only if the dynamic type of *o* is either *T* or an extension of *T*.

```
PROCEDURE Handle (o: Object; VAR msg: ObjMsg);
BEGIN
   IF msg IS CopyMsg THEN handle copy message
   ELSIF ... look for further messages
   END
END Handle;
```

Message records introduce messages as first-class objects which can be manipulated just like any other data structure. It is for instance possible to broadcast a message to many receivers by a generic Broadcast procedure that only deals with arguments of the abstract base type of a message. Message records can also be regarded as a means to express open parameter lists. By explicitly programming the message dispatch mechanism in a message handling procedure, there is – beside the type system – also no restriction in the inheritance structure. Inheritance from the base type, super calls, delegation or forwarding of messages, and abstract (i.e. unimplemented) classes follow naturally without any special language constructs. Section 3.3.1 gives more details about the connection between message handlers and meta-programming. For those cases where this flexibility is not needed, Oberon-2 type-bound procedures [Moe93] can be used to express dynamic binding more conveniently and more efficiently.

```
PROCEDURE (o: Object) Copy (VAR to: Object; deep: BOOLEAN); ...
PROCEDURE (o: SpecializedObject) Copy (VAR to: Object; deep: BOOLEAN); ...
```

A message type is implicitly defined by the signature of a type-bound procedure and there is no need (and no way) to explicitly program the message dispatch mechanism as in the message-record example above. Procedures bound to a base type are inherited by a derived type but may be overridden by binding a procedure to the more specific type. Oberon-2 type-bound procedures are covariant in the type of the receiver parameter and invariant in the rest of the signature.

One can also think of various mixed forms of type-bound procedures and message records. They are, however, not relevant for this work and are therefore omitted in this description.

3.2 System

Oberon is a modular, single-threaded operating system aiming especially at single-user operation of workstations. The Oberon system removes the distinction of system and user programs by treating both as system extensions wich are loaded on demand. The unit of loading is the module, which is also the unit of compilation. Oberon removes the notion of a statically linked application. Instead, it allows the direct execution of parameterless procedures (commands) exported by a module. When a command is to be executed, the system first checks if the exporting module has already been loaded and, if not, loads it and all directly and indirectly imported modules. After executing the procedure, all modules remain loaded until they are explicitly freed by the user. This behavior enables consecutive commands to communicate via global variables rather than via files or other persistent objects. Execution of commands is supported by a command interpreter, which in the Oberon system is integrated in the text editor. Commands are textually denoted in the form M.P, where M denotes the module and P the procedure name. In principle, commands can also be invoked using a graphical representation (button, menu), but this is not part of the core system.

The Oberon system is implemented in the Oberon language and provides a supporting environment for programming and execution of Oberon programs. The system implements the run-time support needed for the Oberon language, in particular heap management and module loading. Oberon has originally been implemented on the Ceres personal computer [Ebe87] but is now available on many commercial machines as well [BCFT92]. These implementations build the Oberon system on top of another operating system [Fra93], from which it mainly uses the device drivers. All implementations hide the underlying operating system as much as possible and provide a target machine independent application programmers interface (API) by means of the set of library modules which comprises the Oberon system. Except for the modules that offer a device driver abstraction, all modules are portable across all target machines. The device driver modules have a target machine independent interface but, of course, the implementation depends on the underlying operating system such as Unix, MacOS, DOS or Windows. With respect to program portability, the Oberon system offers unprecedented convenience. Taking a module and recompiling it on the target machine suffices to port programs.

If Oberon is not used as the native operating system of a machine, there are at least two further application areas. The first one is to regard it as a framework for writing extensible applications. Oberon systems running under Unix, for example, can be started multiple times and every Oberon process can be regarded as an extensible Unix application. The second application area is to use Oberon as a customizable programming environment for almost any kind of language. The Oberon system has been successfully used (and customized) for developing Fortran, Modula-2, C, Lisp and Maple programs to mention just a few.

3.3 Metaprogramming in Oberon

As mentioned in [Maes87], metaprogramming and reflection are a matter of degree in most systems and languages; and this is also true for Oberon. We find some meta-level facilities in both the language and the system without making Oberon a fully meta-level or reflective programming system. Surprisingly, we find meta-level facilities exactly at those points in the language and system that distinguish Oberon from its predecessors.

3.3.1 Language

By introducing the two operations *type test* and *type guard*, Oberon implicitly defines on the language level that every object carries with it some form of type description. With regard to metaprogramming, testing a type is the more interesting operation as it is actually a sort of introspection under the control of a program. In the following we shall discuss the two operations and also an important application of type tests, viz. message handlers, which have also a close relationship to metaprogramming.

Type guard

A type guard *v*(*T*) is a run-time check that ensures that variable *v* refers to an object of at least type *T*. This check might be implicit as well, but requesting the programmer to make it explicit leads to annotating source texts with possible exception points and therefore adds in readability and maintainability of programs.

Type test

A type test *v* IS *T* returns true if and only if the dynamic type of *v* equals T or an extension of *T*. A type test allows the program to inspect the dynamic type of a variable and according to the result of the test to invoke some type-specific code. Therefore, type tests are actually a reflective facility of the Oberon language. It should be noted that the need for this reflective facility has been ignored by popular, object-oriented Pascal dialects, earlier versions of Eiffel and by the current version (2.1) of C++. There is, however, some discussion for introducing meta-level facilities in C++ (c.f. 2.3.4).

Procedure Variables

Another potentially reflective facility of Oberon are procedure variables, i.e. variables that can be assigned a procedure. Procedure variables can be used, for example, to encapsulate state and behavior in the sense of object-oriented programming. In this case, an object can test for its own behavior by comparing the procedure variable that holds its behavior with a known procedure constant. It can also change its own behavior by assigning another procedure to its behavior field. An example of such a self-modification is to adapt the implementation of an abstract data type according to a changing environment.

Message Handlers

Both type tests and procedure variables are used extensively in the standard Oberon way of object-oriented programming, i.e. by using message records and handlers (c.f. 3.1). Although this technique is not part of the language specification proper, it has a close relationship to it especially when compared with other object-oriented languages. In all object-oriented languages, there is a mechanism for dynamic binding of messages to methods, i.e. to the code that is invoked to handle a request. Usually, there is also a mechanism for automatic code inheritance, e.g. a subclass inherits all the methods of the superclass. If a specific inheritance mechanism is part of the language, it is sometimes not the right one for a given problem. Therefore, some object-oriented languages provide for possibilities to adapt the standard inheritance mechanism to ones own needs by means of metaprogramming facilities. CLOS, for example, provides the concept of class and generic-function metaobjects in order to allow the programmer to deviate from the standard inheritance mechanism. Oberon message records and handlers provide even more flexibility by leaving out a static description of inheritance from the language and requesting the programmer to specify the message dispatching and code inheritance explicitly as an Oberon procedure. Conceptually, the Oberon message handler plays the role of a message-dispatching metaobject. Therefore, it is not surprising that the handler technique has much more flexibility than any other language-defined dispatching mechanism. It even allows dynamic inheritance, i.e. to change the inheritance relation at run time. Although in most cases simple forms of message dispatching are sufficient, the potential flexibility of the handler technique should not be underestimated. Some people have criticized the handler technique as being too low-level (in the sense of assembly level) for object-oriented programming. We point out that the term low-level is not appropriate here. A possible criticism would be that handlers are too *meta-level*.

3.3.2 System

The Oberon system contains in its inner core the run-time support necessary for executing Oberon programs. We shall discuss the two almost independent components, viz. heap management and module loading, in turn. Other components of the inner core such as device drivers and most parts of the outer core will not be discussed here since they are not relevant for our purposes.

Heap Management

The language Oberon allows to dynamically allocate data structures in a conceptually infinite area. The infiniteness of this area is implicitly introduced by not providing a possibility to deallocate data structures explicitly. On today's computers, however, memory is finite. Thus, maintaining the illusion of an infinite heap requires to deallocate unused, i.e. unreferenced data structures automatically. The current Oberon systems use mark-scan garbage collection in order to detect and remove unused data structures. We note that garbage collection is actually an example of a built-in meta-level computation as it has as its domain the data structures of other programs. It is to a significant extent the introduction of this meta-level facility that distinguishes Oberon from its predecessors and it is also the reason why interfacing Oberon with more traditional languages is not as trivial as it seems to be from a syntactical point of view.

Module Loading

Oberon programs consist of a set of components called modules. Modules at the language level serve as name spaces. In Oberon, modules allow selective import and export of names and may not be nested. A module may import another module and refer to its exported names. At the system level, modules typically serve as units of compilation and loading. If a computer system provides a memory hierarchy with separate address spaces between the different levels, then there must be a program that establishes executable images at the level which is appropriate for execution by the processor. In practice, this means that compiled Oberon modules stored on non-volatile external storage must be loaded into internal memory to be executable. The terms external and internal refer to the address spaces with respect to the processor. Therefore, besides transferring the binary image, the loader has also to perform address transformations (fixups) in order to map addresses from the external to the internal address space. Additionally, all directly or indirectly imported modules must be loaded in turn. Generally speaking, the loader deals with other programs, hence it can be regarded a metaprogram. Seen from the system point of view, module loading is a highly reflective activity as it introspects and extends the system itself. The ability to load modules on demand and thereby to extend the running system (dynamic linking) is one of the distinguishing features of Oberon at the system level.

In the Oberon system, dynamic linking is introduced by providing a programmatic interface to the module loader, which in Oberon is embodied in module *Modules*. By examining this metaprogramming protocol, we shall detect more meta-level facilities.

```
DEFINITION Modules;

  TYPE
    Module = POINTER TO ModDesc;
    ModDesc = RECORD
      next: Module;
      name: ARRAY 20 OF CHAR
    END ;
    Command = PROCEDURE;

  VAR
    ModList: Module;
```

```
        res: INTEGER;
        importing, imported: ARRAY 20 OF CHAR;

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

END Modules.
```

An abstract data type *Module* is introduced and instances of this type are sequentially linked and anchored in a global variable. A call of function *ThisMod*(*name*) returns the module with the given name, which is first searched in the list of loaded modules and if it is not found, it is loaded from external memory. The global variable *res* returns a result code and variables *importing* and *imported* contain additional diagnostic information. A call of the function *ThisCommand*(*mod*, *name*) returns the command with given *name* from module *mod*. Commands in Oberon are parameterless exported procedures which serve as units of interaction. The user of an Oberon program is expected to execute a sequence of commands rather than a main program with only one entry point. Executing a command is nothing else than calling the procedure installed in a procedure variable. A loaded module remains loaded until an explicit call of *Free* is made, which means that the module with given name is to be unloaded from memory and, depending on the second parameter, that all imported and no longer needed modules should be unloaded recursively. Due to the property that modules stay loaded as long as possible, subsequent commands can communicate simply via a module's global variables.

The Oberon system offers three basic commands for system introspection. A command to show the list of loaded modules (System.ShowModules), a command to show the commands provided by a given module (System.Show-Commands) and a command to display the state of global variables of a module (System.State). It also offers a command to unload unnecessary modules (System.Free) and it provides a command interpreter which, as a possible side effect, extends the system by loading additional modules. The four mentioned commands are potentially reflective, as they can even be applied to themselves (e.g. System.Free System ~).

Two subtle problems exist with loading and unloading of modules which are not satisfactorily solved in current implementations of Oberon. The first problem is related to calling the module loader when the module list is in an inconsistent state. Such situations may arise if a preceding module loading has been terminated by an exception or if the loader is called recursively in the body of a module. The second problem is related to unloading of modules. A procedure exported by a higher level module (client) might be installed in a procedure variable belonging to a lower level module. Unloading the client module is possible, but in this case, the installed procedure variable refers to a non-existent procedure, i.e. it becomes a dangling procedure variable. It is one of the side-goals of this work to provide a solution to these problems (c.f. 4.3.5).

Trap Handling

Another kind of introspection is used for handling of run-time errors. If an exception occurs, the system trap handler inspects the run-time stack and lists the procedure activation chain together with the values of local variables. After that, the system falls back into the central event loop and waits for user input. Producing a trap listing is also a meta-level facility which – in the unintended case of a trap within the trap handler – might even be used recursively.

Module Types

Those Oberon versions which implement the Oberon-2 extensions, additional meta-level facilities are typically provided by means of a module *Types*, which provides run-time access to types defined in Oberon programs. Technically speaking, module *Types* reifies the type descriptors that are available in the Oberon run-time system anyway. As Oberon does not support untyped pointers, a SYSTEM level type PTR is used to express compatibility with every pointer type.

```
DEFINITION Types;

  IMPORT SYSTEM, Modules;

  TYPE
    Type = POINTER TO TypeDesc;
    TypeDesc = RECORD
      name: ARRAY 32 OF CHAR;
      module: Modules.Module;
    END ;

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

END Types.
```

A call of function *TypeOf* returns the run-time type of the given object. The *name* of the type and the *module* in which the type is declared are provided as fields. The inheritance structure of a type may be examined by the function procedures *BaseOf* and *LevelOf*. *LevelOf*(*t*) returns the extension level of *t* where the topmost base type has level 0. *BaseOf*(*t*, *n*) returns the base type of *t* at level *n*. Procedure *NewObj* allows to generate new objects of a given type and *This* returns the type with the given name from a given module.

Although module *Types* is not unconditionally necessary, it simplifies tasks such as internalization/externalization of extensible data structures by providing generic mechanisms to deal with type information [GPHT91]. The fundamental problem lies in the unsymmetry of an object-oriented approach since an object which exists in internal memory can respond to a write message but an object which exists on external storage cannot respond to a read message. In order to solve the problem, the information stored per object must be split up into type and contents information. Internalization/externalization then consists of reading/writing the type information, generating an internal/external object of that type, and sending a read/write message to the internal object. The type specific behavior is only needed for the contents of an object, not for the type information. A solution without module *Types* would be to represent type information indirectly in form of class specific procedures (generators) formulated as Oberon commands. A generator is a procedure which allocates and initializes a new object of a particular class. An object may return the name of its corresponding generator as response to an appropriate message and instantiation can be done by calling the generator as an Oberon command. Passing parameters or return values to or from generators has to be done via global variables since Oberon commands are parameterless procedures. It should be noted that generic instantiation via module *Types* is more convenient but less flexible than using generators. For example, module *Types* cannot be used for dealing with objects implemented in an instance-centered style as instantiation also requires to explicitly install an object's message handler. Using type-bound procedures, this explicit initialization is not necessary and generic instantiation without class specific code is possible.

4 A Meta-level Architecture for Oberon

This chapter describes a variant of the Oberon system which incorporates a meta-level architecture as its principal new addition. We give a definition of the introduced metaprogramming facilities by specifying a programming interface to meta-level facilities. This interface is called the *metaprogramming protocol*. The description of this protocol follows a top-down approach by introducing abstract concepts in the form of abstract classes first, which are then specialized to concrete classes. Finally, we describe a proper modularization, which follows naturally from the top-down approach. The Appendix summarizes the interfaces of the introduced modules. Implementation aspects of the proposed architecture are discussed in Chapter 5 and a selection of possible applications is described in Chapter 6.

4.1 Overview

The basic idea of the introduced metaprogramming protocol is to treat important components of Oberon modules, in particular record types and procedures, as first-class objects, i.e. as objects which can be manipulated by a program (almost) like other objects. In traditional Oberon systems, these objects only exist implicitly in the compiler and are stored in some internal format on a file (the object file). From there, they can be accessed by the module loader but not by normal user programs. Since the life time of type and procedure objects is not bound to any executing process, we shall regard them as persistent objects.

Due to the modular structure of Oberon programs, types and procedures are collected in an enclosing entity called a module. In the introduced meta-level architecture, modules are a special kind of collection, viz. a persistent collection of type and procedure objects. Inspired by Oberon System-3 [Gut93], modules and other forms of persistent object collections have been generalized to an abstract library concept, which forms a common base for the object and metaprogramming system (Section 4.2.1, 4.2.2). In contrast to System-3, where libraries resulted from a generalization of fonts (super fonts) aiming especially at graphical end-user objects, our libraries are a generalization of program modules aiming especially at a seamless integration of metaprogramming facilities into the Oberon system.

The proposed metaprogramming protocol consists of two essential parts. One is a facility for generic access to arbitrary objects (Section 4.2.3) and the other is a generic interface to take control over Oberon procedure calls (Section 4.2.4).

4.2 The Metaprogramming Protocol

Metaprogramming facilities are introduced for different kinds of Oberon language objects including modules, types, variables, and procedures. No metaprogramming support is introduced for constants. In the following sections we shall describe the metaprogramming protocol for the various object kinds in turn. We start with the concept of libraries on which the rest of the protocol is based. We shall use the corresponding Oberon type and procedure declarations for describing this protocol. Note that in the presented record type definitions all of the record fields are considered to be used read-only. The only exception is for implementing subclasses of abstract base classes in which case it might also be necessary to write onto inherited fields. An alternative solution would have been to export the fields as get/set pairs of type-bound procedures, which can be overridden in subclasses only, but this would have introduced unjustified verbosity and inefficiency.

4.2.1 Libraries

According to [Gut93], a library is defined to be an indexed collection of persistent objects. In analogy to traditional files, which are persistent arrays of bytes, libraries allow direct access to objects. Fig. 4.1 shows the apparent analogy between files and libraries.

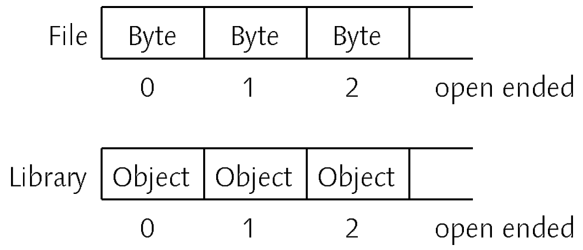


Fig. 4.1 – Files and Libraries

Libraries are typically used as common resources shared between different programs or data structures. Sharing of resources avoids wasting main memory and increases the efficiency of resource loading since a shared resource need be loaded only once into main memory. Well known examples of libraries are fonts (collections of character glyphs), libraries of graphical macros, or libraries of graphical end-user objects. Our notion of libraries extends even further and includes also modules, which are regarded as collections of objects such as types and procedures. The variety of libraries and the need for efficient access to exported objects suggest the use of a class-centered programming style for libraries. We introduce an extensible library mechanism based on an abstract class *Library* which can be refined to specific library classes by means of object-oriented programming techniques.

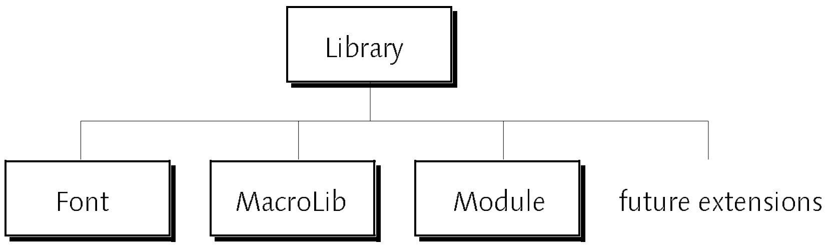


Fig. 4.2 – The hierarchy of library classes

Objects collected in libraries cover a large spectrum. Therefore, the base type *Object* has to be kept as general as possible. Fig. 4.3 outlines the hierarchy of object types. Objects within a library are always accessible via their reference number, which acts as an index into the library. In addition, objects may also be accessible via their name, in which case they are called *exported* objects.

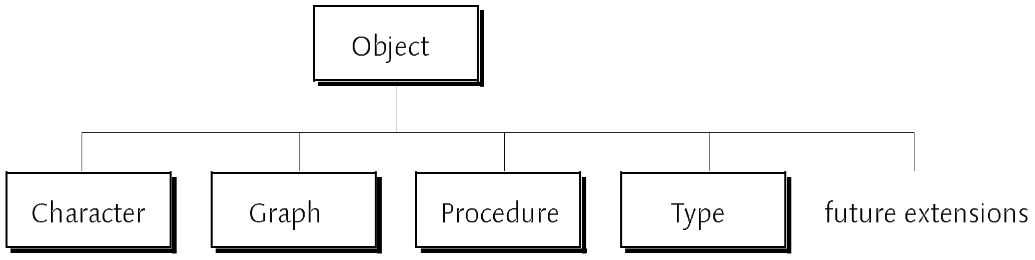


Fig. 4.3 – The object hierarchy

We are now prepared to go into more detail and to discuss the programming interface of the library system. We shall introduce the interface step by step and remind the reader that the Appendix contains the complete module definitions.

```
TYPE
    Name = ARRAY 20 OF CHAR;
    Library = POINTER TO LibraryDesc;
    LibraryDesc = RECORD
        name: Name;
        nofClients, nofImports, nofObjects: LONGINT;
        init, fini: PROCEDURE (L: Library)
    END ;
    ThisProc = PROCEDURE (name: ARRAY OF CHAR): Library;

VAR
    res: INTEGER;
    importing, imported: Name;

PROCEDURE This (name: ARRAY OF CHAR): Library;
PROCEDURE Free (L: Library);
PROCEDURE Install (ext: ARRAY OF CHAR; this: ThisProc);
```

Every library is identified by its *name*. A library may depend on (import) other libraries and may itself be imported by other libraries (clients). The number of loaded clients of a library is given by *nofClients*, the number of imported libraries by *nofImports* and the number of exported objects by *nofObjects*. Cyclic imports are not allowed, thus, the import graph forms a directed acyclic graph (DAG). Fig 4.3 outlines the import relationship between five sample libraries. The arrows point to the client library, i.e. L1 imports L2 and L1 imports L3 and so on.

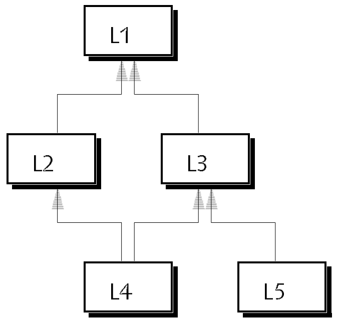


Fig. 4.4 – A sample import graph

The import relationship between libraries is well known for the case of modules. It is, however, also meaningful for other kinds of libraries. A graphical object exported by a library might for instance be composed of imported objects such as icon images, drawing macros or it might need a font for labeling the object.

This

Procedure *This* provides access to a library with a given name and internalizes the library and all its imported libraries if necessary. *This* may be called for all kinds of libraries, thus it is not a class-specific procedure but a generic interface for loading of different library kinds. Therefore, *This* is also called the *generic This* as opposed to a *specific This*-procedure (c.f. procedure *Install* below for more details about class specific library loading).

In order to enable sharing of resources, multiple accesses with the same name return the same library instance. In the case of libraries, the name is considered to be the same kind of reference as a pointer variable. Unlike a pointer reference, which is under control of the run-time system, a reference by name is under control of the user. Hence, libraries are potentially always reachable and therefore are never reclaimed automatically. Similar to module *Modules* in Oberon, procedure *This* returns a result code in variable *res*. Possible result values are *Done*, *LibNotFound*, *InvalidLib*, *InvalidKey*, *CyclicImport*, and *OutOfMemory*. The variables *imported* and *importing* are used for reporting those libraries which could not be loaded.

Free

Procedure *Free* frees the specified library if no clients of this library exist and sets *res* to *Done*; otherwise to *ClientsExist*. Freeing a library does not mean to physically unload the library from main memory but simply to make it unaccessible to further calls of *This*. *Free* actually sets the name field of the library to the empty string to signal that the library is to be considered *anonymous* from now on. Anonymous libraries are subject to garbage collection just like any other object on the heap. In addition to making a library anonymous, *Free* decrements also the *nofClients* counts of all libraries imported by the freed library by one. A client module is not automatically freed even if the *nofClients* count reaches zero.

Install

In order to support extensibility at the level of library loading, procedure *Install* is provided to install a specialized *This*-procedure for a given library type. The type is derived from the library name or more precisely, from the postfix of the library name. In case of multiple matching postfixes, the *This*-procedure associated with the longest postfix is chosen. In particular, this means that the empty postfix is allowed, but that it is only used if no other extension matches. It is recommended to use a dot as the start of the library name extension (e.g. ".Fnt" for fonts). Specialized *This*-procedures are expected to load a library from whatever medium the library name suggests (most often a disk file) into main memory without inserting this library into the set of loaded libraries. It is the task of the generic *This*-procedure to give the newly loaded library a name and to maintain the set of named libraries (i.e. those that are accessible by name), which is necessary for accessing the same library instance later on via its name. Imported libraries must be loaded by means of recursively calling the generic *This*-procedure. The *life cycle* of a library is illustrated by the state diagram in Fig. 4.5.

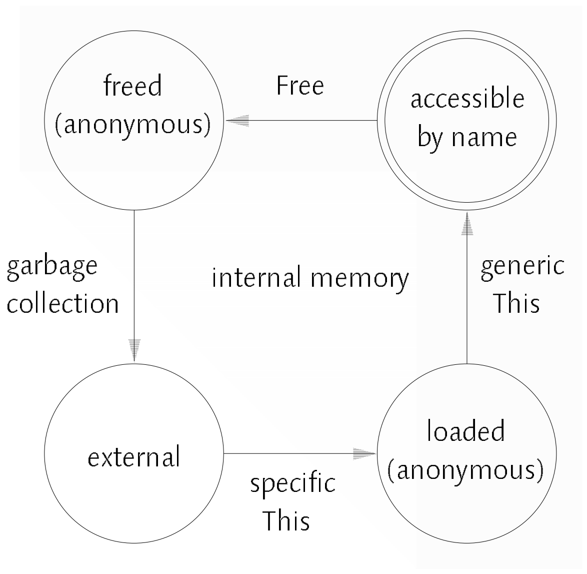


Fig. 4.5 – Library Life Cycle

Initialization / Finalization

An initialization and a finalization procedure (*init* and *fini*) may be associated with a library. Since these procedures are intended to represent for example a module's body or a cleanup procedure, they are bound to a library instance and not to a type. Initialization (i.e. calling *init*) is performed after a library is first loaded into main memory and finalization (i.e. calling *fini*) is done when a library is freed. During initialization the library is accessible by name whereas during finalization it is not. This asymmetry is justified by the need that cyclic loading is possible but cyclic frees are hardly meaningful.

To summarize, the process of loading a library is under control of *This* but includes two upcalls. The first one is the library-specific *This*-procedure which is expected to return an anonymous, uninitialized library. The second is the initialization after the library has been made accessible. Freeing a library is under control of *Free* and includes one upcall, viz. the library-specific finalization routine. Both anonymous states are of a temporary nature, although the anonymous state after freeing a library may last for a longer time.

Invariants

From a robust library loading mechanism we expect that it guarantees at least the following three invariants.

- Uniqueness: no two libraries with the same name are accessible at the same time.
- Completeness: all libraries imported by an accessible library are also accessible.
- Consistency: the *nofClients* counts hold the exact number of client libraries.

The uniqueness property guarantees that every library is uniquely identified by its name. The completeness property guarantees that the import graph does not contain holes, which might for instance occur if loading is prematurely terminated by an exception. It guarantees also that imported libraries can be shared. The third property guarantees that freeing a library is possible at the right time, i.e. when no clients of this library exist. We shall see in Section 5.1 that a framework for library loading which guarantees the above invariants is not as trivial as it might seem at the first glance. Other requirements, e.g. correct *nofImports* and *nofObjects* counts, are much simpler to implement correctly since they are invariant against the activities of the loader, i.e. they are not changed during loading or unloading of libraries.

Library handling is enhanced by two further procedures which don't have side effects on the set of loaded libraries.

```
TYPE
   EnumProc = PROCEDURE (L: Library; VAR cont: BOOLEAN);

PROCEDURE Enumerate (do: EnumProc);
PROCEDURE Lookup (name: ARRAY OF CHAR; VAR L: Library);
```

Procedure *Enumerate* calls the provided *do*-procedure for all libraries in an unspecified order. A *do*-procedure may stop the enumeration by setting *cont* to FALSE. For the common task of searching in the current set of accessible libraries (without changing this set), a more convenient way than *Enumerate* is provided by procedure *Lookup*, which returns either the named library or NIL if the library is not accessible.

Extensibility

As mentioned earlier, libraries are supposed to provide efficient access to their components despite the fact that they unify object collections such as modules and fonts, which are usually stored in a highly specialized binary format. Furthermore, it should be possible to introduce new kinds of libraries later on. Thus, libraries need type specific behavior in order to provide the requested efficiency and flexibility. This behavior is provided by the set of type-bound procedures (methods) specified below. We deliberately left out any methods for writing libraries, i.e. there are no corresponding *Put*–Procedures. This omission is justified by the goal to introduce meta–level facilities into Oberon rather than to reinvent the sophisticated Oberon System-3 library mechanism. Furthermore, all libraries necessary for the Oberon system (modules, fonts) are used in a read-only way. The compiler or a font editor don't necessarily need a common interface for writing their output to a file.

```
PROCEDURE (L: Library) GetImport (n: LONGINT; VAR imp: Library);
PROCEDURE (L: Library) GetObj (ref: LONGINT; VAR o: Object);
PROCEDURE (L: Library) GetRef (name: ARRAY OF CHAR; VAR ref: LONGINT);
PROCEDURE (L: Library) GetName (ref: LONGINT; VAR name: ARRAY OF CHAR);
PROCEDURE (L: Library) GetVersion (ref: LONGINT; VAR n: LONGINT);
```

Method *GetImport* returns the *n*-th imported library if 0 <= *n* < *nofImports*. *GetImport* is useful for inspecting the import graph of a loaded module. A concrete example is to implement a command that recursively frees a module and all its imported modules which have no clients any more. *GetObj* indexes the library and returns the object with the given reference number or NIL, if such an object does not exist. The type *Object* will be explained below. Exported objects may also be identified within a library by their name. Method *GetRef* provides this directory service by mapping names to reference numbers. If the name is not found, the value -1 is returned. Method *GetName* provides the inverse mapping. It returns the empty string if the object does not have a name, i.e. if it is a private object. The interface of an object may be specified by a number (e.g. a version number, a finger print, or a time stamp), which can be retrieved by method *GetVersion*. The version number can be used to dynamically check compatibility between objects.

Objects

The atomic persistent entities collected in libraries are called *Objects*. The following introduces the protocol for the abstract base class *Object*.

```
TYPE
   Object = POINTER TO ObjectDesc;
   ObjectDesc = RECORD
      lib: Library;
      ref: LONGINT;
      handle: Handler
   END ;

ObjectMsg = RECORD END ;
Handler = PROCEDURE (O: Object; VAR M: ObjectMsg);
```

Similar to Oberon System-3, persistent objects exported from a library are self describing in the sense that they carry their external address as well as their behavior in form of instance variables. The external address is described by the pair (*lib*, *ref*) where *lib* refers to the library the object is exported from, and *ref* is the index of the object in the library. If *lib* is NIL, the object is said to be unbound otherwise it is said to be bound to *lib*. The reference number *ref* uniquely identifies a bound object within the library it is bound to. Unbound objects exist only in internal memory and have no external address at all.

The design decision that objects contain their external address within themselves probably needs more justification. The main advantage is that whenever the external address of an object is needed (e.g for externalization of data structures) it can be immediately accessed without any additional computation. Storing the external address within an object is the simplest and most efficient mapping from internal to external addresses. The alternative would be to introduce additional data structures (e.g. hash tables or search trees) which provide this mapping. This would, however, not save any memory (except for unbound objects) but introduce more complexity. Another (minor) advantage is that the *lib* pointer naturally expresses the fact that an object can be bound to at most one library. It also simplifies garbage collection of libraries, since the *lib* pointer within an object establishes a reference to the exporting library and prevents it from being garbage collected as long as at least one of its objects is reachable.

The behavior of an object is expressed by the instance variable *handle*. In order to prepare objects for utmost flexibility and adaptability, we use procedure variables and message records to express object behavior. However, we do not introduce any message types except the abstract *ObjectMsg* in the core library system. Passive objects, i.e. objects which are not prepared to respond to messages, may set their *handle* field to NIL.

4.2.2 Modules

In the proposed metaprogramming system, modules are represented as a special kind of persistent object library, viz. one that exports type and procedure objects. Besides that, the protocol defined for modules is similar to the interface for module *Modules* as described in Section 3.3.2.

```
TYPE
  Module = POINTER TO ModuleDesc;
  ModuleDesc = RECORD
    (LibraryDesc)
    nofProcs, nofTypes: INTEGER;
    data: Data
  END ;

  Command = PROCEDURE;
  Data = POINTER TO RECORD END ;

PROCEDURE ThisMod (name: ARRAY OF CHAR): Module;
PROCEDURE ThisCommand (mod: Module; name: ARRAY OF CHAR): Command;
```

Procedure *ThisMod* returns the specified module or NIL if it does not exist. *ThisMod* is actually a shorthand notation for *This* followed by a test which ensures that the type of the returned library is *Module*. A call of *ThisCommand* returns the specified Oberon command from the given module or NIL if the command does not exist. As in standard Oberon implementations, module names have the empty library name extension. On external storage, however, modules are typically stored with the file name extension ".Obj" (i.e. ThisMod("xxx") loads module "xxx" from file "xxx.Obj"). Objects exported from a module are procedures and (record) types. The fields *nofProcs* and *nofTypes* hold the number of exported procedures and record types respectively. Global variables are not considered as persistent since they are reinitialized whenever a module is loaded. Therefore, global variables are not exported as objects in the strict sense but made accessible for meta-level programs (e.g. a debugger or the System.State command) via a module's *data* field.

Types

A type object represents an Oberon record type. The restriction to record types follows from the fact that in Oberon only record types introduce polymorphism and need run-time data structures to represent them. Instances of type *TypeDesc* are exactly these type descriptors. Types are passive objects, i.e. they are not prepared to respond to messages. Therefore, the *handle* field of type objects is always NIL.

```
TYPE
   Type = POINTER TO TypeDesc;
   TypeDesc = RECORD
      (ObjectDesc)
      level: INTEGER;
      base: ARRAY maxExt OF Type
   END
```

Oberon allows to construct a record type hierarchy by means of type extension. This hierarchy is reflected by the attributes *level* and *base*, which describe the record extension level and the direct and indirect base types. A type with level *n* has base types *base*[*i*] with *i* in [0..*n*]. Any record type *T* extends itself, i.e. *T.base*[*T.level*] = *T*. The remaining base table entries are set to NIL. The level and base type table can be used to express type tests in constant time [Coh91]. The Oberon type test (*o* IS *T*) can be thought of being a shorthand notation for *o.type.base*[*T.level*] = *T* where we simply write *o.type* to access the dynamic type of object *o*. Level and base table have been introduced as record fields instead of accessor functions to allow efficient access to the information which reflects the record extension hierarchy.

Procedures

Procedure objects represent Oberon procedures. More precisely, they represent exported procedures and procedures which are assigned to procedure variables. In principle, it would be possible to represent all procedures (including private and nested procedures) by procedure objects. This would, however, not give an adequate advantage for the introduced storage overhead and conceptual complications. Nested procedures, for instance, would have to be treated specially, since they can only be activated within a given context.

```
TYPE
   Procedure = POINTER TO ProcedureDesc;
   ProcedureDesc = RECORD
      (ObjectDesc)
   END ;
```

Like type objects, procedures are normally passive, i.e. they don't have a message handler. However, it is also possible to think of procedures as active objects which respond to messages such as a request to execute themselves. The idea of active procedure objects will be put forward in Section 4.2.4.

4.2.3 Generic Access to Objects

The purpose of introducing a meta-level architecture for Oberon is to allow the construction of meta-level programs written in Oberon which act upon other Oberon programs and data structures. Meta-level programs must have a possibility to access data structures of other programs in a generic way. In particular, the access mechanism must be independent of the actual type of a data structure since meta-level programs cannot know all possible types in advance and the accessed objects cannot (and should not) know all possible meta-level operations in advance. By the term *generic access* we therefore mean *not type specific* access.

We introduce generic access to arbitrary data structures by means of iterators which can be used by a metaprogram to iterate over the fields of an object and to access them for reading and writing. Following the terminology of Oberon and ETHOS [Szy92], we call these iterators *Riders*. Object riders can be used for meta-operations on arbitrary objects including global data of modules, objects allocated on the heap and even for procedure activation records. In contrast to file riders, which only allow purely sequential access, object riders have to allow hierarchical access in order to allow zooming into structured components such as records or arrays.

A rider is not causally connected to the object it is based on but reifies the state of this object by means of a set of *Read* and *Write* procedures which will be explained below. Examples for usage of generic object manipulation are mainly in the field of mapping data structures from one format or address space into another without relying on type or instance specific code as it would be necessary by an object-oriented approach (c.f. Chapter 6).

The following definition of a generic access mechanism is aiming at simplicity, efficiency, and run-time safety. We intentionally do not provide access to the complete compiler symbol table in order to be independent of a particular compiler and not to affect garbage collection performance due to a large number of objects needed to represent the symbol tables of all loaded modules. Also, we deliberately do not treat objects themself as libraries which allow access to their components by means of *GetObj* or *GetName* methods since objects are not necessarily persistent and components of objects are not necessarily objects in turn.

```
TYPE
  Rider = RECORD
    mode, class: SHORTINT
  END ;
```

```
PROCEDURE (VAR R: Rider) GetLocation (VAR name: ARRAY OF CHAR; VAR vis: SHORTINT);
PROCEDURE (VAR R: Rider) ReadX (VAR x: X);
PROCEDURE (VAR R: Rider) WriteX (x: X);
PROCEDURE (VAR R: Rider) Pass (VAR from: Rider; VAR res: INTEGER);
PROCEDURE (VAR R: Rider) Skip;
```

In order to allow efficient access to different kinds of structured variables, different kinds of riders with type-bound procedures are introduced. The abstract base type *Rider* contains fields *mode* and *class*, which are necessary to describe the attributes of the variable at the rider's current location. Additional information about this variable is provided by method *GetLocation*, which returns the name and the visibility of the variable at the rider's location. *Name* and *vis* have been handled separately because this information is not always needed and, if not needed, should not slow down iterating over objects.

```
vis = Private | Exported | ReadOnly.
```

The field *mode* specifies whether the rider is positioned on a variable, a VAR-parameter, a record field, an array element or none of them because the end of the object has been reached. Value parameters are treated like variables, i.e. they have mode *Var*.

```
mode = Var | VarPar | Fld | Elem | None.
```

The field *class* specifies the type class of the element a rider is positioned on. The classes comprise all Oberon standard types plus classes for pointers, procedures, arrays and records.

```
class = Byte | Bool | Char | SInt | Int | LInt | Real | LReal | Set | Ptr | Proc
    | Array | Record | DynArr.
```

Methods *ReadX* and *WriteX* serve to read and write a value at the current position and to advance the rider's position. *X* stands for any of the basic type classes. Concrete examples are:

```
PROCEDURE (VAR R: Rider) ReadInt (VAR x: INTEGER);
PROCEDURE (VAR R: Rider) ReadPtr (VAR x: REFANY);
PROCEDURE (VAR R: Rider) ReadString (VAR x: ARRAY OF CHAR);
etc.
```

The parameter type *REFANY* in *ReadPtr* stands for any pointer type. Since Oberon requires a static type for all pointers, it has been necessary to express the type REFANY by means of the pseudo module SYSTEM as TYPE REFANY = SYSTEM.PTR. It should be noted that type errors can still be checked dynamically.

Method *Pass* may be used to pass a value or a variable directly from a source to a destination rider and to advance the position of both riders. *R.Pass*(*from*, *res*) can be seen as *from.Read*(*X*) followed by *R.Write*(*X*) but allows also to pass structured data and to pass variables by reference in case that R.mode = VarPar. The result variable *res* is set to zero if passing was successful. A typical application of *Pass* will be seen in Chapter 6, the passing of parameters to a parameter record in a generalized command interpreter.

A rider may be advanced to the next position (e.g. the next record field or the next array element) without reading or writing the variable at its current location by calling method *Skip*.

Special rider classes have been introduced for iterating over records, arrays, and procedure activation records. Fig. 4.6 shows the hierarchy of the introduced rider classes.

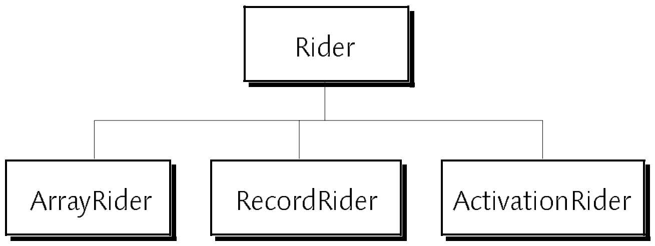


Fig. 4.6 – The Rider Hierarchy

RecordRider

A *RecordRider* may be used to iterate over the fields of a record structure. Record riders introduce one additional attribute, the *level* of the field a rider is positioned on. The level starts at zero after opening a record rider, i.e. at the fields of the very base record type.

```
TYPE
   RecordRider = RECORD (Rider)
      level: INTEGER
   END ;

PROCEDURE OpenRider (VAR R: RecordRider; o: REFANY);
PROCEDURE SetLevel (VAR R: RecordRider; level: INTEGER);
```

Procedure *OpenRider* sets up a record rider on a record object specified by a pointer to the record. Generic access to global module data is also provided via record riders. The Oberon command System.State, for instance, uses *OpenR*ider(*R*, *m.data*) to open a rider *R* on the global data of module *m*. Procedure *SetLevel* may be used to set the rider's location to the first field of the specified extension level.

ArrayRider

An *ArrayRider* may be used to iterate over the elements of an array structure. Array riders introduce two additional attributes, the length of the array and the index of the rider's location. Method *GetLocation* returns *index* in form of a string as the name of the current location.

```
TYPE
   ArrayRider = RECORD (Rider)
      len, index: LONGINT
   END ;

PROCEDURE SetIndex (VAR R: ArrayRider; idx: LONGINT);
```

Procedure *SetIndex* may be used to directly position the rider to the specified array index. As a minor restriction, we do currently not allow to open an array rider via a pointer to an array (fixed size or open). Array riders may be opened only as zooming operations on structured components of a given base rider (see below).

ActivationRider

A special kind of rider is provided for operating on procedure activation frames. The main purpose of activation riders is to allow the implementation of human readable stack dumps without relying on implementation-dependent information. Activation riders reify the run-time stack of Oberon programs in order to allow the implementation of trap handlers as meta-level programs.

```
TYPE
   ActivationRider = RECORD (Rider)
      module, proc: ARRAY 32 OF CHAR;
      dlink, retpc, relpc: LONGINT
   END ;

PROCEDURE OpenFrame (VAR R: ActivationRider; sp, pc: LONGINT);
```

For setting up an activation rider with *OpenFrame*, the frame must be identified by the pair (*sp*, *pc*) where *sp* means the stack pointer and *pc* means the value of the program counter. The latter acts as a tag for the otherwise untagged activation frame at *sp* and allows to recover the structure of the activation frame. The values for *sp* and *pc* are supposed to be provided by an exception handling mechanism, which – in the case of Oberon – is not part of the language but provided in the form of library routines. Fields *module* and *proc* return the module and procedure name of the identified procedure activation. *OpenFrame* also sets fields *retpc*, *relpc* and *dlink*. *retpc* is the return address of the procedure call, *relpc* is the program counter value relative to the beginning of the module's code section, and *dlink* (dynamic link) is the stack pointer of the calling procedure. Thus, *OpenFrame*(*R*, *R.dlink*, *R.retpc*) may be used to open a rider on the calling procedure activation.

Zooming into Structured Components

Structured data types can be accessed via hierarchically refining (zooming) a rider which is located on a structured component such as a record or an array. Zooming into structured components may be done by calling the appropriate *Zoom* procedure as specified below.

```
PROCEDURE ZoomRecord (VAR R: RecordRider; VAR base: Rider);
PROCEDURE ZoomArray (VAR R: ArrayRider; VAR base: Rider);
```

ZoomRecord opens a record rider on the current position of the *base* rider, which may be any kind of rider with *base.class* = *Record*. Similarly, *ZoomArray* opens an array rider on the current position of the base rider, which may be any kind of rider with *base.class* IN {*Array*, *DynArr*}. (In principle, these three procedures could have been introduced as type-bound procedures as well. As a matter of taste, we tried to stay with normal procedures wherever possible.)

Textual Representation of Objects

The Oberon system provides a *built-in* abstract data type *Text* and access mechanisms for reading and writing called *Readers* and *Writers* respectively. Since texts play a central role in the Oberon system, we provide two general purpose procedures for writing a variable starting at a rider's current location to a text writer. In principle, these procedures could be implemented as regular meta-level programs solely based on the previously introduced protocol.

```
PROCEDURE WriteObj (VAR W: Texts.Writer; VAR R: Rider; expand, indent: INTEGER);
PROCEDURE WriteItem (VAR W: Texts.Writer; VAR R: Rider);
```

Procedure *WriteObj* may be used to map a variable starting at *R*'s current location until the end of the object to a textual representation by iterating over all components of the variable. Every iteration step produces text of the form *"location = value"*, where procedure *WriteItem* is used to format the values of simple data types including pointers and procedure variables. Structured components (records, arrays) are represented by so-called *folds*, which are text pieces embraced by special fold marks that can be interactively expanded and collapsed when displayed in the standard Oberon text editor [Hau93]. The first *expand* nesting levels are initially expanded, the others are collapsed in the generated text. Each refinement of a structured component increases the indentation level specified by *indent*. The indentation information is used to indent the output by an appropriate amount of white space. Character arrays are always represented as double quoted strings. Pointers and procedure variables are represented by a special text element (RefElem) which prints either as "NIL", as "↑" for pointers or as "@" for procedure variables. The purpose of reference elements is not only to display pointer values (this could also be done textually) but to establish a reference to an object that might otherwise be unreferenced and thus become subject to garbage collection. Note that the existence of a textual representation of a pointer does not imply the existence of the pointer itself although the pointer existed at the time the textual representation has been constructed. The reason is that eventually the rider will not be accessible any more or it might have been set to another object. Thus, the object which contains the pointer might become unreachable and thereby any pointers anchored in this object would disappear although the textual representation still exists. A typical place where this happens is the procedure activation stack, which contains references into the heap but is destroyed after the stack has been mapped into a textual representation and control is returned to the Oberon main event loop.

Example:

In order to give the reader a feeling of how the introduced metaprogramming protocol may be used, we present parts of the implementation of *WriteItem* and *WriteObj* as an example. Examples for applications of *WriteObj* will be presented in Chapter 6.

```
PROCEDURE WriteItem (VAR W: Texts.Writer; VAR R: Rider);
  VAR si: SHORTINT; i: INTEGER; ... p: REFANY;
BEGIN
  CASE R.class OF
    | SInt: R.ReadSInt(si); Texts.WriteInt(W, si, 0)
    | Int: R.ReadInt(i); Texts.WriteInt(W, i, 0)
```

```
        ...
        | Pointer: R.ReadPtr(p); Texts.WriteElem(W, reference element for p)
        ...
    END
END WriteItem;

PROCEDURE WriteObj (VAR W: Texts.Writer; VAR R: Rider; expand, indent: INTEGER);
    VAR R1: RecordRider; R2: ArrayRider; W2: Texts.Writer;
        name, s: ARRAY 64 OF CHAR; vis: SHORTINT;
BEGIN
    WHILE R.mode # None DO
        R.GetLocation(name, vis);
        WriteName(W, name, indent);
        IF R.class < Array THEN WriteItem(W, R)
        ELSIF R.class = Record THEN
            ZoomRecord(R1, R);
            Texts.WriteElem(W, opening fold element); Texts.OpenWriter(W2);
            IF expand <= 0 THEN Texts.WriteLn(W2);
                WriteObj(W2, R1, expand – 1, indent + 1); Indent(W2, indent)
            ELSE Texts.WriteLn(W);
                WriteObj(W, R1, expand – 1, indent + 1); Indent(W, indent)
            END ;
            insert contents of W2 into opening fold element;
            Texts.WriteElem(W, closing fold element);
            R.Skip
        ELSIF R.class IN {Array, DynArr} THEN
            ZoomArray(R2, R);
            IF R2.class = Char THEN R.ReadString(s);
                Texts.Write(W, 22X); Texts.WriteString(W, s); Texts.Write(W, 22X)
            ELSE
                ...
            END
        END ;
        Texts.WriteLn(W)
    END
END WriteObj;
```

Generic Instantiation

Similar to module *Types* as described in Section 3, generic instantiation, i.e. creation of objects where the type is given as a variable, is provided by procedure *New* and the type of an arbitrary object may be examined by function *Type*. Note that the apparent name conflict of using *Type* for two different purposes (TYPE Type and PROCEDURE Type) is resolved by a proper modularization (c.f. Section 4.3).

```
PROCEDURE New (VAR o: REFANY; t: Type);
PROCEDURE Type (o: REFANY): Type;
```

The remaining functionality of type objects results directly from the library mechanism. In particular, the exporting module can be accessed via the inherited *lib* field, and the name of the type with *GetName*.

Example

Creation of a variable of type "M.T" may be done by the following sequence of steps:

```
VAR mod: Module; ref: LONGINT; typ: Object; obj: REFANY;
BEGIN
   mod := ThisMod("M");
   mod.GetRef("T", ref);
   mod.GetObj(ref, typ);
   New(obj, typ(Modules.Type));
```

4.2.4 Active Procedures

Besides access to data structures, metaprograms need also be able to control procedure activations by means of appropriate facilities. We shall introduce these facilities by exploiting the idea of active procedure objects, which are procedure objects that do have a message handler. Applications of the introduced facilities range from generalized command interpreters over customization and debugging tools (tracing, pre- and post-condition checking) to advanced concepts such as remote procedure calls.

Eval

As a first step towards meta-level facilities for controlling procedure activations, we introduce procedure *Eval*, which takes two parameters, namely a procedure object and a record that represents the parameters of a procedure activation. For the latter, we introduce the abstract data type *Parameters*. Procedure *Eval* provides a generic interface to invocation of Oberon procedures by evaluating the given procedure *P* with the provided arguments *par*. A generalized command interpreter may for example not only activate parameterless procedures (as usual in standard Oberon systems) but also procedures with parameters.

```
TYPE
   Parameters = RECORD END ;
   ParamRider = RECORD (Rider) END;

PROCEDURE Eval (P: Procedure; VAR par: Parameters);
PROCEDURE GetParams (p: Procedure; VAR par: Parameters);
PROCEDURE OpenParams (VAR R: ParamRider; VAR par: Parameters);
```

Every procedure has its own parameter record with the procedure specific parameters. The parameters can be read and written using generic access to objects as discussed in the previous section. A special kind of rider (*ParamRider*) is provided, which can be used to iterate over parameter blocks. A special property of parameter riders is that they allow to *Pass* VAR-parameters by reference. Procedure *GetParams* may be used to create a parameter record corresponding to the parameters of a particular procedure. A rider on a parameter record can be opened by calling procedure *OpenParams*. The typical sequence of steps is to create a parameter block first, then to set up a parameter rider, assign or pass the actual parameters and finally call *Eval*. Chapter 6 presents as an example a generalized Oberon command interpreter based on this protocol.

Procedure Handlers

As the second step towards controlling procedure activations, we introduce the notion of *active procedures*. These are procedures which have a message handler installed, i.e. which can react to messages sent to them (c.f. 4.2.1). We postulate that calling an active procedure results in sending an appropriate *invocation message* to this procedure object. Thus, by installing a message handler in a procedure object, we get control over execution of this procedure by reacting to invocation messages. This mechanism is supposed to be transparent to the caller, i.e. it should not be necessary to recompile all modules where an active procedure is called. Instead of, we provide procedure *InstallHandle* to install a message handler in a procedure object and to maintain internal data structures which are necessary for the implementation of this facility.

```
TYPE
   InvocationMsg = RECORD (ObjectMsg)
      par: Parameters
   END ;

PROCEDURE InstallHandle (p: Procedure; handle: Handler);
```

The handler mechanism provides a generic interface for taking control over procedure calls. To illustrate this, we introduce a simple procedure handler which simulates the behavior of passive procedure objects by calling *Eval* as response to an invocation message.

```
PROCEDURE PassiveHandle(P: Object; VAR M: ObjectMsg);
BEGIN
   WITH P: Procedure DO
      IF M IS InvocationMsg THEN Eval(P, M(InvocationMsg).par) END
   END
END PassiveHandle;
```

By means of calling *InstallHandle*(*P*, *PassiveHandle*) for any procedure *P*, procedure *PassiveHandle* acts as an interpreter for this procedure object independent of its parameter list. Thus, *PassiveHandle* is really a generic interpreter for all procedures. This genericity is gained by some loss of efficiency and convenience since there is an implicit transformation of the actual parameters of a procedure call into a message record and vice versa. Accessing parameters (if needed) can not be done directly but only by using the rider mechanism. To trade some of the overhead of message passing for reduced flexibility, we introduce a second facility for controlling procedure activations called *filters* and let programmers choose the flexibility they need.

Filters

For the case that a specific interpreter or one or more interpreter extensions for a specific procedure are needed, we introduce the concept of *filters*. A filter is a procedure which takes control over the activation of a specific kind of procedure, viz. a procedure with exactly the same parameters as it has itself. A filter procedure can access the parameters of a procedure call directly as its own parameters rather than by applying the rider mechanism to a parameter record. Fig. 4.7 shows a procedure *P* with three parameters *x*, *y*, *z* which has two associated filter procedures *F1* and *F2*. Calling *P* results in activating the outermost filter *F2* first, which might itself call *P*. This results in the activation of the filter one level below – *F1* in this case. IF *F1* calls *P*, the body of *P* is activated. Recursive calls of *P* at this point should result in activating the outermost filter again. Thus, the filter chain is supposed to be cyclic in case of recursion.

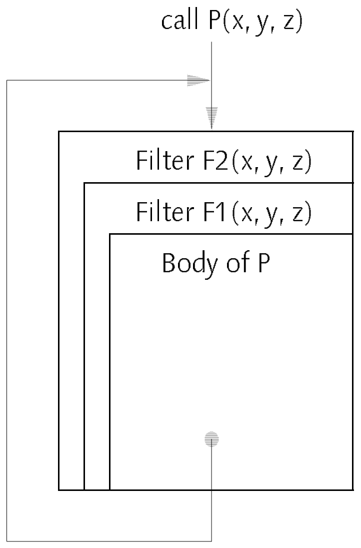


Fig. 4.7 – A Filter Chain

The concept of filters is introduced by means of the special procedure handler *FilterHandle*, which allows to perform *Push* and *Pop* operations on this procedure to install and deinstall filters in a stack-like fashion.

```
PROCEDURE FilterHandle (O: Object; VAR M: ObjectMsg);
PROCEDURE Push (P, Filter: Procedure);
PROCEDURE Pop (P: Procedure): Procedure;
```

For convenience reasons, we allow also to apply *Push* to passive procedure objects (handle = NIL), in which case an implicit installation of *FilterHandle* is performed. Symmetrically, we define that *Pop* resets a procedure object to passive state if the filter chain eventually becomes empty.

The filter concept could in principle be implemented solely as a regular meta-level program based on procedure handlers. But for simplifying the implementation, it is advantageous to have it integrated in the core system.

To summarize, we would like to point out explicitly that using active procedures should not be regarded as the regular programming style but should be used for the sole purpose of dealing with procedure calls within meta-level programs. In passing, we note that Oberon's modular structure naturally supports restricting the usage of certain facilities. It is easily possible not to provide the definition (symbol file) of the module which implements active procedures to a novice programmer.

4.3 Modularization

The presented metaprogramming protocol has not been implemented as one unit but has been distributed into several modules by considering the natural division of functionality and the levels of abstraction. The advantage of a proper modularization is to make explicit in the import list of a module which kinds of meta-level facilities are to be used and which are not.

Libraries

The basic library mechanism is implemented in module *Libraries*. For each library subclass (e.g. Modules, Fonts) we introduced a separate module.

GenericObjects

Generic access to arbitrary data structures and generic instantiation are encapsulated in module *GenericObjects*.

ActiveProcedures

Module *ActiveProcedures* implements the facilities to deal with parameter lists, procedure evaluation, procedure handlers and filters.

Figure 4.7 outlines the import relationships between the introduced modules where an arrow from A to B means that B imports A. There are in fact more imports, however, we focus on the most relevant ones to show the system's overall structure.

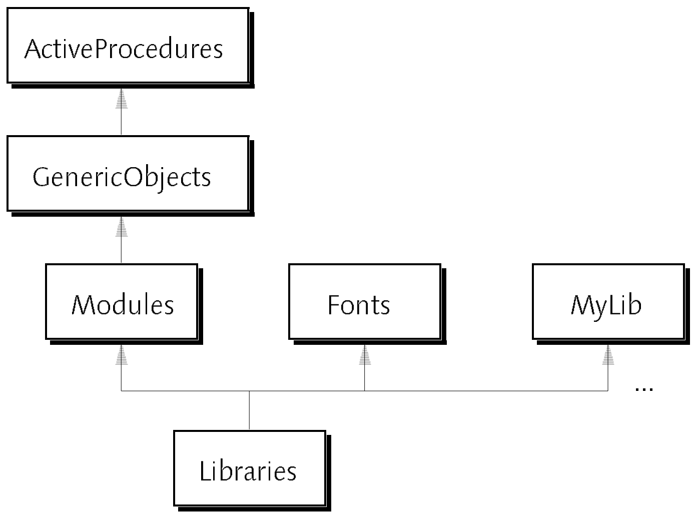


Fig. 4.8 – Modularization

Comparison with Standard Oberon and Oberon System-3

In order to stress the difference between our approach and other Oberon system architectures, Fig. 4.9 shows the subtyping relationships between important system components. Standard Oberon as described in [WG92] introduces no subtyping relations between Modules, Fonts or other libraries such as macro libraries for the Draw package. Oberon System-3 introduced a dual system architecture consisting of modules on the one side and libraries on the other side. All kinds of persistent object collections are defined as subtypes of Libraries. Our approach integrates also modules into the library framework.

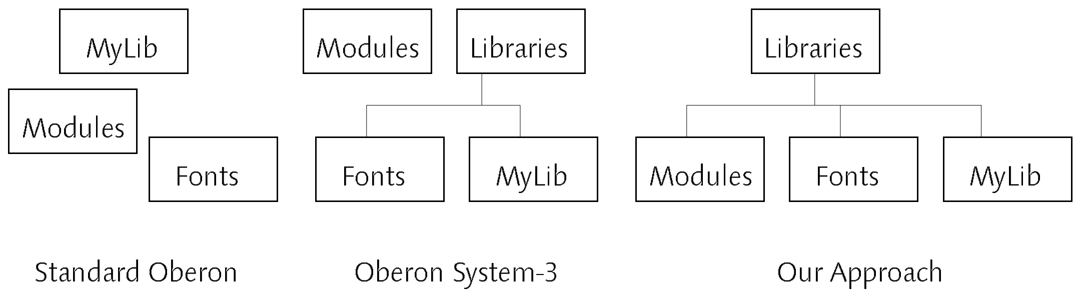


Fig. 4.9 – Architecture Comparison

This concludes the definition of our meta-level architecture for Oberon. The next chapter will focus on implementation aspects.

5 Implementation

This chapter describes implementation aspects of the introduced meta-level architecture. We try to follow the same order as used for the definition of the metaprogramming protocol. Therefore, we start with the basic library mechanism. After that, the implementation of two special kinds of libraries (Modules and Fonts) will be discussed. Special emphasis is put on module loading and unloading as well as on garbage collection issues. Subsequently, the implementation of generic access to objects and the implementation of active procedures is discussed. Most implementation aspects in this chapter are independent from the underlying hardware or operating system platform, those which are not refer to the Oberon implementation for SPARC–based workstations [Te91] and are marked as *Note*.

5.1 The Library Mechanism

Libraries play multiple roles in the presented meta-level architecture. These different roles impose different requirements and challenges on the implementation of the library mechanism. In particular, the library mechanism provides a way for system extension, a means for shrinking the system and libraries represent the state of the system and by that they are the roots for garbage collection. In the following, we shall discuss the various roles and associated implementation problems in detail.

5.1.1 A Framework for Library Loading

First of all, libraries serve as units of system extension. The set of accessible libraries, which constitutes the state of the system, may be extended at run-time. The task of the library loading mechanism is to internalize libraries and to resolve references to other libraries (imports). Since libraries are an abstract class which can be specialized to modules, fonts and the like, we can only provide a framework for library loading. The specialized classes participate in the loading process via upcalls.

Although library loading seems to be a straight-forward recursive process, it is surprisingly intricate. This can be seen by the fact that in the existing Oberon implementations many different solutions for module loaders exist, none of them satisfies all expectations, though. It is not difficult to implement a loader which works well for the common cases, however, in the presence of library initialization routines (e.g. module bodies) there are situations which are likely to leave the system in an inconsistent state. By inconsistent state we mean that the main invariants for libraries as described in Section 4.2.1 (uniqueness, completeness, consistency) are not established after the loader returns control. There are two problematic situations, cyclic loads and abnormal termination, which are both possible due to the existence of arbitrary initialization routines. We shall study these problems by taking a closer look at the special case of module loading. The reader might keep in mind that module loading is just an instance of the general problem of library loading.

Cyclic Loads

The library import graph is defined to be a directed acyclic graph (DAG), i.e. cyclic imports are not allowed. However, if the loader can be invoked directly for instance within a module's body, arbitrary libraries may be loaded, including clients of the initializing module or even this very module itself. The following example outlines the situation where the library loader is invoked explicitly within the body of *M1*, which expresses a reflective computation in which M1 accesses itself.

```
MODULE M1;
  IMPORT Libraries;
  VAR L: Libraries.Library;
BEGIN
  L := Libraries.This("M1")
END M1.
```

In order to distinguish such situations from cyclic imports we call them *cyclic loads*. Note that cyclic loads are not restricted to single module cycles but may include multiple modules as well.

One might argue that a cyclic load within a single module is a programming error and has not to be considered by the loader. Anyway, this would not really solve the problem and it is hard to predict that there will never be a useful application of it (cf. Section 5.1.2). Furthermore, single module cycles are the simplest among the problematic situations and they are handled gracefully in most existing Oberon loaders.

More subtle problems arise if several modules participate in a load cycle as shown in the next example.

```
MODULE M1;
   IMPORT M2;
END M1.

MODULE M2;
   IMPORT Libraries;
   VAR L: Libraries.Library;
BEGIN
   L := Libraries.This("M1")
END M2.
```

In existing Oberon loaders, it is likely to happen that two modules named *M1* will be inserted into the list of loaded modules or that the system crashes. In some loaders, the result depends on whether *M1* or *M2* is to be loaded first. Fig. 5.1 summarizes the mentioned single- and multi-module load cycles.

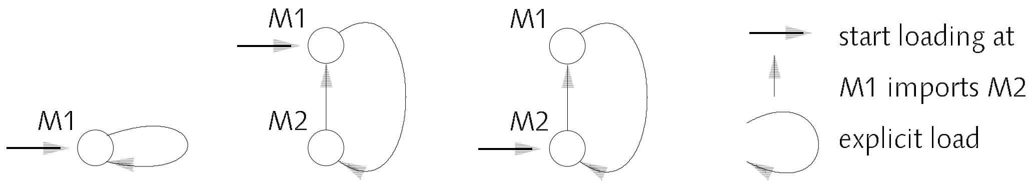


Fig. 5.1 – Cyclic loads

In realistic examples, there are usually more modules involved, several of them may form a cycle. Such situations are not very common but they do appear in practice. An example can be found in the boot process of the Oberon system, where module *Oberon* explicitly loads module *System*, which in turn imports module *Oberon*. Other examples occur whenever a program (e.g. Oberon's *Edit* package) tries to load optional extensions.

Exceptions

Another problem is the correct treatment of abnormal termination of a library's initialization routine by a run-time error, a user interrupt, or a HALT statement. The interested reader may check the *nofClients* count of module *M2* in his/her favorite Oberon loader after trying to import module *M1* of the following example (it is likely to be one although no client of *M2* can be loaded).

```
MODULE M1;
   IMPORT M2, M3;
END M1.
```

```
MODULE M2;
END M2.

MODULE M3;
BEGIN HALT(99)
END M3.
```

Since libraries are vital system resources, we expect the loading mechanism to guarantee the main invariants for all situations, i.e. even for the examples presented above. If the library mechanism is to be extensible, we have of course to rely on the correctness of the specialized libraries but given a correct implementation of all library classes, it should not be possible to produce an inconsistency simply by *using* the mechanism.

It is well-known that by means of a tricky sequence of compilation and editing steps cyclic imports can be constructed which cannot be detected by an Oberon compiler. In contrast to cyclic loads, we do consider this a programming error. In our opinion, an endless recursion, which leads to a stack-overflow or out-of-memory exception, is an acceptable implementation of this pathological case.

Possible library loading mechanisms can be classified into iterative and recursive strategies. Recursive strategies can be further divided into pre- and post-order approaches (in-order does not make sense here). We have come to the conclusion that a recursive post-order strategy is the ideal solution for our library loading framework. The following sections describe the advantages and disadvantages of the different approaches.

Iterative Approach

[Szy92] describes an iterative module loader which guarantees consistency in all cases. The central idea is to maintain three disjoint sets of modules (load, init, ready) in order to handle cyclic loads and to be able to resume loading after a module initialization has trapped. The load-set contains all modules which have to be loaded, the init-set contains all completely loaded modules which have to be initialized, and the ready set contains the set of accessible modules. During a module's life time in internal memory, it travels from the load-set to the init-set and then to the ready-set. In case of a trap in a module body, the init and load sets have to be cleared by an exception handler. The described algorithm guarantees uniqueness, completeness, and correct client counts. However, it is fairly complex and it is difficult to verify that it is correct. This can be seen by the fact that two errors have been found in an earlier version of the iterative loader despite extensive testing, using, and reasoning. One of the errors could have been fixed locally, the other one uncovered a contradiction in the specification of the loader. The correction required substantial changes in the loader's specification and implementation. Without going into details, the essence is that an iterative approach does not simplify the loading process.

Some of the introduced complexity arises from the fact that the import graph is flattened by the described approach. This destroys information (e.g. initialization order) which is necessary and needs to be recomputed afterwards. The optimal solution would be a recursive loader which can also handle problematic situations gracefully. In contrast to [Szy92] we claim that such a loader is possible without giving up the elegance of the recursive approach.

Pre-order Traversal

Pre-order traversal means to make a client library accessible before all of the imported libraries are loaded. Obviously, this technique removes problems with load cycles in an elegant way and has been used in the original Oberon implementation. The problem with this approach is that the completeness property is violated as long as not all imports are resolved. If an initialization routine traps or if a cyclic load occurs, there may be accessible modules which are not initialized or whose imports are not completely resolved. To fix the problem, one has to introduce additional state information for each library. This information must describe the set of unresolved imports. Thus, when accessing a library, the existence of unresolved imports can be detected and the missing libraries can be loaded silently. Initialization of the library has also to be performed if the set of unresolved imports becomes empty.

The required additional state for expressing the set of unresolved imports and the necessary checks for finding incomplete libraries are the most important drawbacks of this approach. Furthermore, it is unsatisfying that inconsistency is introduced and eliminated later on instead of being avoided at all.

Post-order Traversal

Post-order traversal means to make a client library accessible only after all of its imports have been loaded. This automatically guarantees completeness. It needs, however, special attention for the case of load cycles in order to guarantee also uniqueness. Note that termination is not a problem since load cycles always involve at least one module which is already loaded (and initialized) and therefore breaks the cycle. A module loader following the post-order approach but ignoring the problems with cyclic loads and exceptions has been described in [WG92]. Fig. 5.2 outlines the situation of a load cycle consisting of two modules.

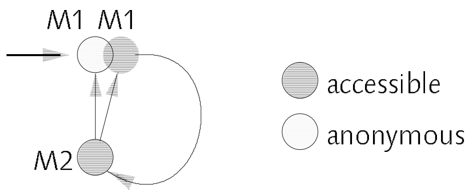


Fig. 5.2 – Post-order traversal of a load cycle between two modules

Suppose that loading starts with module *M1*, which imports module *M2*. Due to the post-order strategy, *M1* is not accessible by name as long as *M2* is not loaded and initialized. *M2* in turn accesses explicitly module *M1* which will be loaded despite the fact that it is already in the process of being loaded. The recursive activation of the loader can produce as a side effect an accessible library in addition to an anonymous one. Thus, in order to guarantee uniqueness, it is necessary to check for the existence of a library before inserting a new element into the set of accessible libraries. Note that this guard is not only necessary but also sufficient for the uniqueness property. The reason is that there is only one place where libraries are added into the library set, this place is guarded and there are no side effects between the guard and the actual insertion which might affect the library set.

We can now outline a framework for library loading based on a recursive approach using post-order traversal.

```
PROCEDURE This (name: ARRAY OF CHAR): Library;
   VAR lib, h: Library; this: ThisProc;
BEGIN Lookup(name, lib);
   IF lib = NIL THEN
      get library generator(name, this)
      IF this # NIL THEN
         lib := this(name);
         IF (lib # NIL) & (lib.name = "") THEN
            Lookup(name, h);
            IF h # NIL THEN lib := h
            ELSE Publish(lib, name);
               IF lib.init # NIL THEN lib.init(lib); res := Done END
            END
         END
      ELSE res := TypeNotFound
      END
   ELSE res := Done
   END ;
   RETURN lib
END This;
```

Accessing a library starts with checking if the mentioned library is already accessible (*Lookup*). If not, the generator of the library, i.e. the *this*-procedure installed for the particular library name extension (cf. Section 4.2.1), must be searched. If found, it will be called and is expected to load and return a library or NIL. In addition, *this* sets the result variable *res* to *Done* or to an appropriate error code. A library generator may return both accessible and anonymous libraries, the latter is the regular case, though. An accessible library may be returned in case of a cyclic load or in case that one library is substituted for another library. An example is font substitution in case a font library is not found. If the generator returns an anonymous library, a second *Lookup* is performed to guarantee uniqueness after inserting a new element into the set of accessible libraries. Recall that a library might have been inserted into the set of accessible libraries as a side effect of *this*. Insertion into the set of accessible libraries is done by the auxiliary procedure *Publish*. *Lookup* and *Publish* must be performed atomically, i.e. they must either be performed as a whole or not at all. This can be achieved by disabling interrupts (esp. keyboard interrupts) but is highly platform specific and therefore not discussed here. After initialization of the newly loaded library, the (global) result code is reset to *Done* in order to avoid propagation of error messages across library initializations.

```
VAR libs: Library;

PROCEDURE Publish(L: Library; name: ARRAY OF CHAR);
   VAR imp: Library; i: INTEGER;
BEGIN
   COPY(name, L.name);
   L.nofClients := 0;
   L.next := libs; libs := L;
   FOR i := 0 TO L.nofImports - 1 DO
      L.GetImport(i, imp);
      INC(imp.nofClients)
   END
END Publish;
```

The set of accessible libraries is represented as a sequentially linked list anchored in the global variable *libs*. The post-order traversal has the nice effect that by simply inserting new libraries at the beginning of the library list, the list is topologically sorted, i.e. a library is always in front of all of its imported libraries.

Note that neither the *name* of a library nor the *nofClients* count are allowed to be modified outside the library loading framework. To guarantee this restriction, we use Oberon-2 read-only exports for the *name* and *nofClients* fields.

Correct treatment of reference counts for libraries is guaranteed in the proposed solution by delaying the increment of an imported library's reference count until the client is inserted into the library set. Thus, if an initialization routine of one of the imported libraries traps, the reference counts of all other imported libraries are not changed.

The reader might miss a recursive invocation of the loader within the library loading framework. Actually, recursion does not appear directly but indirectly. If a library is loaded, the library generator may invoke the generic *This*-procedure recursively in order to load imported libraries.

Example

To illustrate the mechanism, we show the main steps for loading a configuration consisting of modules *M1* and *M2* which form a two-module load cycle (cf. Fig. 5.1). We assume that we start loading with module *M1*.

```
This("M1")                                   library loader
   m1 := thismod("M1")                       module loader
      import := This("M2")                   resolve imports
         m2 := thismod("M2")
         Publish(m2, "M2")
         m2.init                             body of M2 loads M1
            L := This("M1")
               m1:= thismod("M1")
                  import := This("M2")       M2 is published
               Publish(m1, "M1")
               m1.init
               RETURN m1
         RETURN m2
      Lookup("M1", m1)                       avoid a second instance of M1
      RETURN m1 m1.name = "M1"
   RETURN m1
```

Summary

The proposed framework for library loading is based on a variant of the recursive post-order traversal described in [WG92]. The differences are that it is extensible and it guarantees all three main invariants even in problematic situations. *Completeness* is guaranteed by using a post-order traversal, *Uniqueness* is guaranteed by checking the library set before inserting a new node and *Consistency* is guaranteed by incrementing the number of clients counts only when a client is actually inserted into the library list.

5.1.2 Library Unloading

Symmetrically to the task of system extension, libraries serve also as units of unloading. From a library mechanism we expect the same robustness in case of reducing the system as for extending it. However, as described in Section 3.3.2, safely unloading a module, which in our architecture is a specialized library, is an unsolved problem in all existing Oberon implemen- tations. In the following, we try to present a possible solution by taking a closer look at the specification of the problem. We discuss advantages and disadvan- tages of several implementation strategies according to the traditional specification. Again, since module and library unloading are so closely related, we cannot always distinguish between the two problems.

Traditional Specification

Modules.Free(*m*) unloads module *m* if it is not imported by other modules.

The intention behind this specification is that the freed module is physically unloaded, i.e. the associated storage is disposed by calling *Modules.Free*. Unfortunately, a module which is not imported by other modules may still be referenced via procedure variables, Oberon-2 method tables (allocated within type descriptors) or module pointers. In order to cope with the introduced dangling references, a number of possibilities exist:

• *Unload modules by unmapping them from virtual memory.* This technique uses a memory management unit (MMU) for making an unloaded module inaccessible. The virtual memory address space assigned to a module is unmapped and never reused as long as the system is not rebooted. By means of the MMU, however, physical memory will be reused. At run-time, access to dangling references can be trapped by the memory management hardware without any run-time penalty to legal memory references.

The disadvantages of this technique are that hardware support is required and that virtual memory will eventually get exhausted, although very slowly. Moreover, there is the problem that unexpected run-time errors may occur. The problem with these traps is that they may force the user to reboot the system unintentionally. Consider for example the case where an open viewer contains a dangling reference because the module which implements the contents frame has been unloaded. Closing the viewer results in a message sent to the contents frame which in turn causes a trap. The viewer cannot be resized or closed from that time on. Thus, rebooting the system is the only way to get rid of the viewer. Such situations happen quite often during the development stage of a program, especially because in a system which is based on garbage collection the user (=programmer) does not pay any attention to dangling references. Closing a viewer followed by unloading a module is expected to be the same as unloading a module followed by closing a viewer.

• *Use indirection for external calls*. This technique uses an additional indirection for external procedure calls via a so-called link table. Freeing a module is coupled with disposing the module space and resetting the link table to dummy entries which, when invoked, result in a run-time error. The link table itself is never reused.

The disadvantages of this solution are that it slows down external procedure calls, it needs a distinction between external and internal procedure calls in the compiler, it will not catch dangling module pointers and memory will also get exhausted although much slower. The problem with run-time errors remains the same as with the unmapping strategy.

• *Never dispose modules*. This strategy makes a module only inaccessible, but does not dispose memory (although this is not the intention of the specification). The advantage is that it neither requires an MMU nor indirect external calls, and it also avoids run-time errors.

The obvious disadvantage of this approach is that it requires a large physical memory in order to be practical. The approach is also appropriate for systems with virtual memory and demand paging (Unix) where unloaded modules are simply swapped out by the operating system if physical memory is needed for other purposes. In any case, memory will eventually get exhausted, although the paging device is normally significantly larger than the physical memory. A less obvious but more severe problem is that there is a memory leak if the garbage collector does not regard all unloaded modules as roots for accessible heap objects. The reason is that calling a procedure of an unloaded module might access global pointer variables of this module which point to a reclaimed heap block. If the garbage collector takes also unloaded modules as roots, data structures rooted in unloaded modules including those which are unreachable will never be released. If all global pointers are set to NIL when unloading a module, run-time errors may occur as in the previously discussed approaches, although less frequently since references to procedures of unloaded modules are legal now.

• *Check for references before unload.* This approach has (to the best of our knowledge) not been implemented so far, but it has been proposed by several people. The idea is to check for references to a module before unloading it.

Only if there are no references, the module is allowed to be unloaded otherwise unloading results in an error message.

Although the proposal seems to be appealing at a first glance, there are several problems with it. First, it needs more run-time type information than available in current Oberon implementations. In particular, it needs to know where procedure variables are located in memory. Second, if a module is referenced via a variable which is not under explicit control by the user, unloading is impossible and the user normally does not know why. Even if the system reports where a reference is anchored in, the user can normally not change it. Third, if a group of modules is to be unloaded and there exist references only within this group, unloading should be possible. This would further increase the complexity of this approach.

Weakened Specification

Modules.Free(*m*) removes module *m* from the set of accessible modules if it is not imported by other modules.

This specification allows to separate the task of making a module inaccessible from the task of physically unloading it. The former is done by *Free*, the latter can be delegated to the garbage collector. The specification further allows several generations of modules to coexist in memory. At most one is accessible by name, though.

Obviously, the weakened specification can be generalized to libraries without any problems. One has simply to replace the word module by library. A solution where library disposing is delegated to the garbage collector imposes two new requirements on the collector:

• *Procedure variables must be treated like pointers*. This implies that procedures which might be assigned to a procedure variable must carry a type tag. In our architecture this is the case since procedures are treated as objects anyway. Fig. 5.3 outlines the representation of procedure objects.

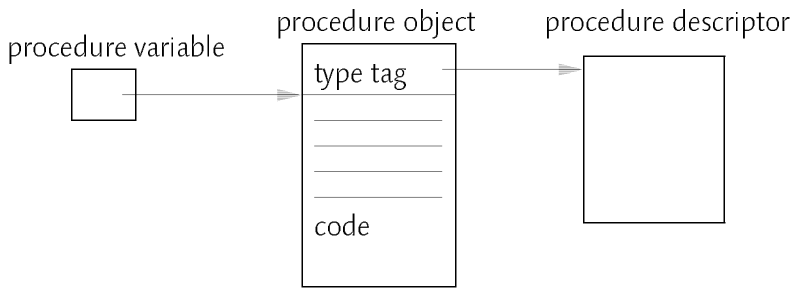


Fig. 5.3 – Procedure variables treated as pointers

> Note: Care has to be taken if a procedure variable points to a procedure outside the Oberon system, e.g. a procedure of the underlying operating system as it is sometimes the case in SPARC-Oberon. In this case, the compiler needs a hint that the variable is to be ignored by the garbage collector. We employed the *sysflag* mechanism of the OP2 compiler [Cre90] to signal untraced procedure variables (e.g. VAR setjump: PROCEDURE [1] (env: TrapEnv)).

• *Type tags must be treated like pointers*. This is also done easily (at least conceptually) since type tags are pointers to type descriptors and type descriptors in our architecture are first-class objects.

> Note: There is a subtle problem involved if mark/sweep garbage collection is used and the type tag of an object is not treated as a normal pointer by the marking algorithm. This is for instance the case if the marking technique proposed in [GPHT91] is used. This technique avoids any restriction in the number of pointers per object by not treating type tags as regular pointers but by using them essentially as counters which span the whole address space.
>
> Type descriptors contain exactly one interesting reference, the *lib* field pointing to the exporting module. If this module is accessible (lib.name # ""), it will be marked anyway. Only if it is anonymous, marking must be done explicitly for this module. Therefore, we have to invoke the *Mark* procedure recursively in order to mark anonymous modules referenced within type descriptors. The recursive invocation of *Mark*, which is an iterative process otherwise, is justified here since the recursion depth is limited by the (small) number of anonymous modules and not by the number of reachable heap objects.

A particular problem in the context of the Oberon tasking system is unloading of a module which implements an installed task. A *task* in Oberon is an object with an associated procedure, which gets activated periodically. Without any provisions, the task would continue to run and the module continue to be reachable via the global list of installed tasks until the task is removed. But how can it be removed? If the module exports a command to deinstall the task, this would not help since the module (and the command) is no longer accessible. The solution is to use the library finalization mechanism for such cases. A module might install a finalization routine which is up-called within *Free* after the module has been made inaccessible. (Installing a finalization routine is an example for a reflective computation which produces a single module load cycle as described in Section 5.1.1).

```
MODULE TaskMod;

  PROCEDURE TaskHandle ...
  PROCEDURE StopTask ...
  ...
```

```
BEGIN
    L := Libraries.This("TaskMod");
    L.fini := StopTask;
END TaskMod.
```

A simple framework for library unloading can now be outlined. It is represented by procedure *Free*, which takes a library as parameter. We call this procedure a framework because it contains upcalls of methods and of the finalization routine. The second parameter of standard Oberon *Free* (the *all* parameter) has been left out because unloading of imported libraries can be expressed in terms of recursively calling *Free* with the knowledge of *nofImports* and *GetImport*. Modifications of the library set should be done atomically, i.e. interrupts should be disabled. In practice, however, this is not necessary since *Free* is so fast that a user interrupt via keyboard is almost impossible.

```
PROCEDURE Free* (L: Library);
    VAR imp: Library; i: INTEGER;
BEGIN
    IF L.name # "" THEN
        IF lib.nofClients = 0 THEN
            lib.name := "";
            FOR i := 0 TO lib.nofImports – 1 DO
                lib.GetImport(i, imp); DEC(imp.nofClients)
            END
            IF lib.fini # NIL THEN lib.fini(lib) END ;
            res := Done
        ELSE res := ClientsExist
        END
    ELSE res := Done
    END
END Free;
```

After a library has been freed, it remains in memory until the garbage collector detects that it is not reachable any longer and can be reclaimed safely. It is the task of the garbage collector to remove an unreachable and anonymous (name = "") library node from the list of loaded modules. The details about garbage collection of libraries in general and modules in particular will be discussed in sections 5.1.3 and 5.3.5 respectively.

Summary

The proposed strategy for unloading libraries separates the task of making a library inaccessible and the task of physically unloading the library. Physical unloading is delegated to the garbage collector, which avoids the need for a memory management unit, avoids run-time errors, and avoids wasting of memory. We claim that the proposed solution fits naturally into an operating environment based on garbage collection.

5.1.3 Libraries as Roots for Garbage Collection

The accessible libraries represent the state of the system and are therefore the principal roots for garbage collection. If garbage collection is performed between commands (synchronous collection) we can rely on the fact that no objects are rooted on the procedure activation stack whereas if garbage collection is performed during command execution (asynchronous collection) we have also to take into account that objects may be anchored in the stack.

Considering a traditional *mark-and-sweep* collector, libraries impose no fundamentally new kind of problem. However, if libraries are supposed to contain a large number of objects, this raises the problem of how to tune the heap manager and the garbage collector such that annoying garbage collection pauses are avoided. This section focuses on exactly this problem. It is assumed that the reader is familiar with basic garbage collection techniques especially with mark/sweep collection ([Wil92] gives an excellent overview of state-of-the-art garbage collection techniques in uni-processor systems).

Tuning Strategy

Garbage collection pauses might result from both the mark and the sweep phase. The efficiency of the mark phase depends on the number of reachable objects and the number of references within each object. The efficiency of the sweep phase depends on the number of both reachable and unreachable objects. Therefore, we have to aim for reducing the number of objects which are touched during both garbage collection phases. The biggest advantage can be expected by reducing the number of reachable objects, since both phases depend on them. If a library contains many objects, we must take care that not every single object is regarded as a root for the reachability analysis whenever this is possible.

Subobjects

In order to reduce the number of individual objects on the heap, we can introduce a sort of hierarchical heap and think of a library as being one big memory block (a compound object) which contains an arbitrary number of components. The idea is to allocate and deallocate compound objects as a whole rather than as many individual parts. If a component does not have references pointing outside the set of objects which are reachable from its container, we call it a *subobject* (see Fig. 5.4). Subobjects need not be regarded as roots for the mark phase, since they don't introduce any references that might affect the result of the reachability analysis. Examples for subobjects are character objects collected in fonts. A character typically has a reference to a data structure which describes its raster image but this pattern object is also a component of the font. Other examples are type and procedure objects exported by a module. Fig. 5.4 shows a compound object with three subobjects.

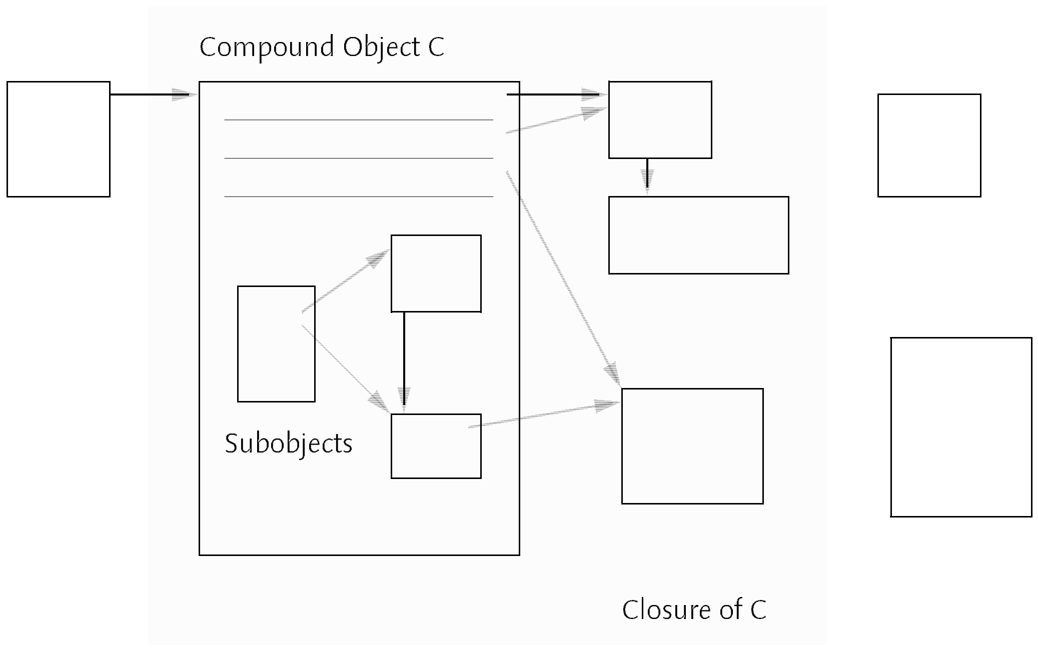


Fig. 5.4 – Subobjects

[Szy92] describes a technique which supports a simplified form of subobjects within a mark/sweep garbage collector by specially marking compound objects and distinguishing between atomic and compound objects during the mark phase. The introduced simplification is that a compound object must be leaf, i.e. it must not contain references to other objects. The reason for this restriction is that a subobject is never marked itself but marking is propagated to the encapsulating compound object. In order to keep the marking algorithm simple and iterative, the leaf restriction had to be introduced. Subobjects must have a back pointer to their container in order to enable propagation of marking.

The consequence and advantage of this approach is that subobjects need not be unmarked in the sweep-phase since they are not marked in the mark-phase. Thus, the goal of reducing the number of objects touched in both phases has been achieved.

The most striking disadvantage of this solution is the restriction of compound objects to be leaf, which precludes its usage for libraries. Recall that libraries in general do have references to other objects, e.g. to imported libraries. Another disadvantage is that during the mark phase an additional test has to be performed for every pointer. The detection of subobjects is done by testing an implicit flag expressed by a certain alignment of objects, which introduces some storage overhead. Thus, there is a certain run-time and storage overhead which exists even if subobjects are not used at all.

In contrast to this approach, we tried to trade the leaf restriction for another one and to move any overhead from the garbage collector to the place where subobjects are actually used. The technique outlined below shares with [Szy92] the unpleasant property that it is not implementable within the safe subset of Oberon, i.e. library implementations which want to take advantage of subobjects have to import the pseudo module SYSTEM.

Libraries as compound objects

The main idea is to restrict compound objects to libraries in contrast to restricting them to be leaf. The implication of the new restriction is that marking of subobjects need not be propagated to their parent objects since (accessible) libraries are marked anyway. Thereby, we can avoid any complication in the marking algorithm. Subobjects can be marked exactly like other objects. The only difference is that subobjects will not and need not be unmarked in the sweep phase. Only after unloading a library, an exact reachability analysis of compound objects and their components has to be performed (see below). This is, however, expected to be the exceptional case and can therefore be accepted.

Since subobjects may remain marked after garbage collection, we need to treat them as *possibly marked* whenever their type tag is used (because the type tag is the place used in SPARC-Oberon and other Oberon implementations to store the mark bit). When using the type tag (e.g for type tests or type guards), the mark bit must be masked out before dereferencing the tag. On modern processors this additional operation requires at most one machine cycle. Of course, this requires a cooperating compiler, which has to insert the mask instruction whenever we specify an appropriate hint for a record type. The necessity of this hint is an unpleasant property of the introduced optimization. Furthermore, due to polymorphism in Oberon the hint must be provided for the very base type of possibly marked records. In particular, we have to provide this hint for the base type *ObjectDesc* in order to allow optimized handling of libraries. Fortunately, the hint has no damaging effect on unoptimized libraries, since masking an unset bit is idempotent. The hint is also completely invisible to users of libraries and to implementors of unoptimized libraries.

Unmarking Subobjects

In order to eventually provide for correct physical unloading of anonymous libraries, we must detect when a library is not reachable any more. For this purpose we have to unmark all subobjects of *anonymous* libraries before every garbage collection. A library is unreachable if it is anonymous and it is not reachable itself and none of its subobjects is reachable. Reachability of a library via one of its subobjects is implicitly tested by requiring that every subobject has a back pointer to its parent object. The consequence is that whenever a subobject gets marked, the corresponding parent object gets also marked. The only important point is that for an exact reachability analysis all subobjects must be unmarked before marking. We introduce method *Unmark* for exactly this purpose. In order to provide greater flexibility, we introduce a way to handle also components which violate the subobject property of having only references to objects which are reachable from the containing library. Those objects must be unmarked before every garbage collection regardless whether the library is anonymous or not. The *all* parameter specifies whether all components or only the non-proper subobjects should be unmarked (an example for a non-proper subobject will be seen in Section 5.3.2).

```
PROCEDURE (L: Library) Unmark (all: BOOLEAN);
```

Method *Unmark* is not abstract but implemented as the empty procedure in order to provide a suitable standard behavior for non-optimized libraries. Note that the existence of procedure *Unmark* is not in contradiction with our goal of tuning libraries with a large number of subobjects. Unmark(FALSE) affects only the exceptional case of non-proper subobjects, and Unmark(TRUE) is called only for the exceptional case of freed libraries. Experiments have shown that on a 20 MHz SPARCstation 1, 5000 heap objects already produce noticeable garbage collection pauses. This number of objects would result for instance by loading 10 fonts with 256 character and 256 pattern objects if fonts were not optimized (see Section 5.4 for more details about fonts).

Implementors of optimized libraries must be careful when overriding method *Unmark*. They must be aware of the fact that it is called as part of garbage collection and must therefore neither use nor affect the state of the heap except for clearing the mark bits of subobjects.

Out-Of-Heap Libraries

The last problem which we have to mention before we can present the structure of the garbage collector is the need to handle both libraries allocated on the Oberon heap and libraries allocated outside this heap. The latter is intended to be used especially for modules. The advantage of allocating module space outside the Oberon heap is that it avoids the restriction of a module's size (incl. global data) to the size of the heap and it helps also in reducing heap fragmentation. Libraries allocated outside the heap must be unmarked explicitly since the regular scan phase will not meet them.

The Garbage Collector

We are now ready to outline the overall structure of the garbage collector.

```
PROCEDURE GC(markStack: BOOLEAN);
   VAR L: Library;
BEGIN L := libs;
   WHILE L # NIL DO L.Unmark(L.name = ""); L := L.next END ;
   L := libs;
   WHILE L # NIL DO
      IF L.name # "" THEN Mark(L) END ;
      L := L.next
   END ;
   IF markStack THEN MarkStack END ;
   SweepLibs;
   Sweep
END GC;
```

Procedure *Mark*(*p*) marks all objects reachable from *p* including *p* itself. We shall not discuss *Mark/Sweep* in more detail here nor shall we discuss the conservative stack garbage collection represented by procedure *MarkStack* since both are standard techniques implemented in most existing Oberon implementations. The interested reader is referred to [GPHT91] and [Szy92].

Before *Sweep*, we have to unlink unmarked anonymous libraries and to dispose unmarked out-of-heap libraries. This is done in procedure *SweepLibs*. It is important to call *SweepLibs* before *Sweep* because the latter destroys the mark information which is needed by the former. The criterion used for the *out-of-heap* property is simply the address of the library, which can be compared with the address range of the heap.

```
PROCEDURE SweepLibs;
  VAR L, prev, next: Library;
BEGIN L := libs; prev := NIL;
  WHILE L # NIL DO
    next := L.next;
    IF (L.name = "") & ~marked(L) THEN
      IF L = libs THEN libs := next ELSE prev.next := next END ;
      IF out-of-heap(L) THEN dispose(L) END ;
      L := prev
    ELSIF out-of-heap(L) THEN unmark(L)
    END ;
    prev := L; L := next
  END
END SweepLibs;
```

Summary

The presented approach for handling bulky libraries is based on the technique of subobjects. It avoids any overhead in case subobjects are not used. There is only a small overhead when accessing the type tag of a subobject. Due to the escape to unsafe SYSTEM level features and the required compiler support, the tuning strategy is still not fully satisfying but works well in practice. More involved garbage collection techniques such as generational collection [Ung84] would solve the problem of large libraries in a conceptually cleaner way but at the same time it would introduce significantly more complexity into the system. Pointer assignments would have to be trapped by memory management hardware or by additional conditional statements. Stop-and-copy generational collection (the usual way) would also prohibit a conservative approach to stack collection. We think that this additional complexity is not justified.

5.2 Object Finalization

Objects which represent external resources such as disk files, processes, or network connections need to be notified upon deallocation in order to synchronize with their associated external resource. Typical examples are file objects, which can release their associated disk sectors when not accessible any more. Since the kinds of limited external resources are not known in advance, we cannot hard-code them in the garbage collector. What we need is an extensible mechanism to deal with such situations. A solution to this problem is *object finalization*. This means that an arbitrary procedure can be registered for an object which is to be finalized. Before reclaiming the object, the registered procedure is called in order to perform object specific finalization operations. It should be noted that finalization is not a problem which comes up when implementing an operating environment such as Oberon on another one such as Unix but is a problem which is inherent in every garbage-collection based system which deals with the world outside the main memory. [Szy92] describes the problem of finalization in more detail and presents a safe finalization technique. A less involved technique is used in most existing Oberon implementations. We shall shortly outline both of them. After that, we shall combine the advantages of both to create a new finalization mechanism.

To avoid a possible confusion, we would like to point out that object finalization is completely independent from library finalization. Object finalization is a garbage collection issue, library finalization happens when freeing a library, i.e. before the garbage collector gets into effect.

Safe Finalization

[Szy92] describes a technique for object finalization which is claimed to be safe in two respects. One is the absence of dangling pointers, the other is that finalized objects are not reachable from any garbage collection roots (externally unreachable) and not reachable from other finalizable objects (internally unreachable). A finalizable object is an object which has been registered for finalization and is not externally reachable. In other words, besides the absence of memory inconsistencies, this technique claims to guarantee that an object which has been finalized will never be used later on, not even from other finalizing objects.

The proposed solution is based on additional reachability tests between finalizable objects after the mark phase of a mark/sweep garbage collector. It uses additional marking colors to perform these tests by means of a specialized variant of the mark phase. The algorithm essentially enumerates all acyclic paths starting at finalization candidates (red) in a directed graph whose nodes are the unmarked objects (grey or white) reachable from finalization candidates and the edges are the pointers between these objects. A subtle point is that references to a finalizable object *o* which stem from a cycle starting at *o* and references which originate in another finalizable object have to be distinguished (see Fig. 5.5). The former can be ignored, the latter establish internal reachability and prohibit finalization of *o*. Any finalization candidate which stays red gets finalized after the sweep phase.

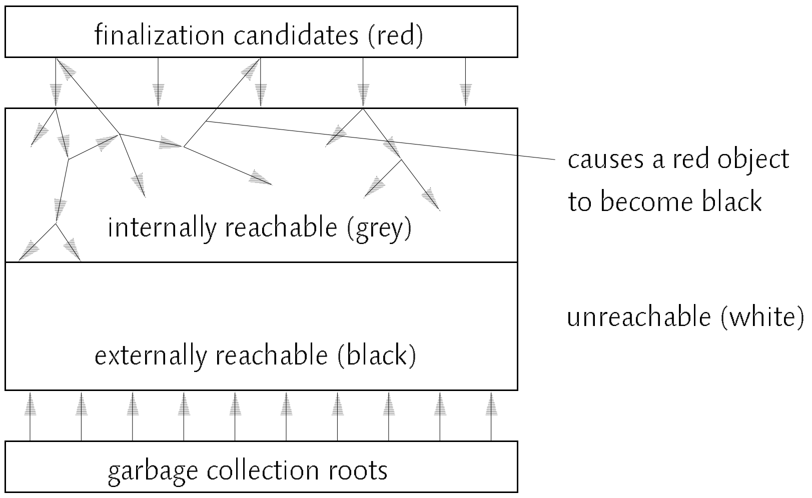

finalization candidates (red)
internally reachable (grey)
causes a red object to become black
unreachable (white)
externally reachable (black)
garbage collection roots

Fig. 5.5 – Safe Finalization

We would like to point out that a finalized object may always establish global reachability of itself during finalization, e.g. by assigning a reference to itself to a global variable. Thus, any particular finalization order cannot guarantee unreachability for finalized objects. Other drawbacks of this approach are that the specific finalization order leads to problems if cyclic references between finalizable objects exist. Actually, the finalization strategy inhibits finalization and storage reclamation in this case. Another problem is that the presented solution is rather complex. The specialized mark phase essentially doubles the static complexity of the collector and the dynamic worst-case behavior is exponential! Another minor disadvantage is that the technique can only be used for objects derived from a given base type because it uses additional state information per finalizable object.

Sweepers

A simple yet extensible technique for object finalization has been introduced in many Oberon implementations by providing a way to install additional procedures which extend the sweep phase of the garbage collector. These installable sweep procedures are called *sweepers*. A sweeper may test the mark bit of an object associated with a particular external resource to perform some additional sweep operation such as deallocating disk sectors or closing network connections if the object is unmarked. The advantage of this technique is that it avoids any additional static and dynamic complexity. Even if it is heavily used, it is very efficient.

The most striking disadvantage of this technique is that it is not safe and it relies on internal details of the run-time system such as the mark bit and the fact that mark/sweep collection is used. The technique is unsafe because sweepers are called between the mark and the sweep phase in order to have the mark information available. Allocating new objects or usage of type tags at this point will result in heap havoc.

Combining the Advantages

Since both techniques have disadvantages, we have to look at the specification of the problem more thoroughly in order to find a solution which allows to combine the advantages of both. We notice that the internal unreachability property in the safe solution causes the troubles without actually providing an adequate advantage. By omitting this property from the requirements, we get a simpler solution which is still safe with respect to memory consistency. In practice, we observe that finalization procedures are simple, and don't create new references. Thus, we can delegate the problem of re-establishing reachability of finalized objects to the programmer without introducing any practical problems.

Note: The worst case for file objects in SPARC-Oberon, for example, would be that a run-time error occurs after a finalized file object is used for reading or writing. This is due to the associated Unix file descriptor, which gets an invalid value upon finalization. Of course, a more sophisticated implementation could provide a mechanism for reopening closed files automatically. Such objects would have to be registered again for finalization. Thus, having references to finalized objects need not always lead to a disaster and need not necessarily be considered unsafe.

To summarize, what we want is a solution to the finalization problem which has the following properties:

- no effect on efficiency if not used
- efficient if used
- safe with respect to memory consistency
- applicable to arbitrary objects
- an object which is not externally reachable will be finalized

An object that is to be finalized before being reclaimed by the garbage collector has to be registered together with a finalization procedure. We provide the following interface to clients of the finalization mechanism:

```
TYPE
   ObjFinalizer = PROCEDURE(o: REFANY);

PROCEDURE Register (o: REFANY; finalize: ObjFinalizer);
```

The set of objects registered for finalization is internally maintained by a list of auxiliary nodes anchored in the global variable *fin*.

```
TYPE
   Node = POINTER TO NodeDesc;
   NodeDesc = RECORD
      next: Node
      o: WEAKPTR;
      marked: BOOLEAN;
      finalize: ObjFinalizer
   END

VAR fin: Node;
```

An object *o* which is to be finalized is referred to by a weak pointer, i.e. a pointer which is ignored in the reachability analysis of the garbage collector (WEAKPTR is actually expressed as LONGINT since there is no language construct for weak pointers in Oberon). This means that the mark phase will mark all the auxiliary nodes as reachable but not all the objects which are registered for finalization. After the mark phase, procedure *CheckFin* scans the node list, saves the mark state of the registered object in *marked*, and calls the regular *Mark* procedure for this object. This guarantees that all objects which are reachable by a finalization procedure will survive the following scan phase and precludes dangling references.

```
PROCEDURE CheckFin;
   VAR n: Node;
BEGIN n := fin;
   WHILE n # NIL DO
      IF ~marked(n.o) THEN n.marked := FALSE; Mark(n.o)
      ELSE n.marked := TRUE
      END ;
      n := n.next
   END
END CheckFin;
```

Unmarked nodes are finalized after the sweep phase. The finalized objects will be reclaimed in the next garbage collection cycle if none of the finalization procedures has re-established reachability of the objects. In general, if an object is registered *n* times, it will be finalized *n* times and it needs at least *n*+1 garbage collection activations to be reclaimed.

```
PROCEDURE Finalize;
   VAR n, prev: Node;
BEGIN n := fin;
   WHILE n # NIL DO
      IF ~n.marked THEN
         IF n = fin THEN fin := fin.next ELSE prev.next := n.next END ;
         n.finalize(n.o)
      ELSE prev := n
      END ;
      n := n.next
   END
END Finalize;
```

At least one registered unmarked object will be finalized and removed by one garbage collection activation. Usually, all of the inaccessible objects can be finalized at once since references between finalizable objects are very rare. The exact sequence of steps in the garbage collection algorithm is now:

```
mark;
CheckFin;
SweepLibs;
Sweep;
Finalize
```

Summary

The presented finalization mechanism has been implemented and turned out to work very well. Finalization has been used for implementing Oberon's *Files* module on top of a Unix file system, for interprocess communication under Unix, and for interfacing to an X-Windows server. The mechanism is simple to implement, simple to use, efficient, and safe with respect to memory consistency.

5.3 Modules

An important concrete subclass of libraries are *Modules*, which are the principal building blocks of Oberon programs. Modules are normally composed in a textual form but other representations (e.g. graphical) are imaginable as well. For efficient execution of Oberon programs, modules have to be translated from a human readable and writable form into a machine dependent format by a compiler. Since compilation is a time-consuming process which in general involves processing of additional files for interface checking across module boundaries, compiled modules are usually stored in (object) files from where they can be efficiently loaded without recompilation.

As it has been shown by [Fra94], an efficient object-file format is possible which is independent of any particular processor architecture by moving the machine dependent parts of the compiler (the back-end) into the module loader. It has been reported that this technique is fast enough to be practicable. Nevertheless, a performance penalty which has to be reduced by advanced object-file compression techniques is put on the loader. Since the work on a portable, efficient object-file format is still ongoing research, we decided to stay with the well-proven technique of generating machine dependent object files.

The object file format has been modeled after the one described in [WG92]. Some changes have been made to adapt the format to our specific needs. In the following we shall outline the structure and contents of object files in more detail, however, we shall not consider the machine specific parts of object files. We shall also discuss the run-time organization of modules.

5.3.1 The Object-File Format

Object files consist of a sequence of sections, each of them preceded by a tag which can be used for plausibility checks. The structure of object files may be described by the following EBNF grammar:

```
Module = Header Imports Entries Directory Const Code Links Ref.
```

The *Header* section contains the module name and key and information concerning the various other sections. In contrast to many existing Oberon implementations, the *Imports* block is immediately following the *Header*. This order is better suited for recursively loading imported modules before allocating the new module node (post-order traversal) since it avoids the need for temporarily saving data within auxiliary storage. The *Entries* section describes the addresses of all exported objects (including type objects) relative to the beginning of the code section. The *Directory* section provides the information required to access exported objects by name. The *Const* section contains constants of the module including type descriptor subobjects. Therefore, no extra type section in the object file is necessary. The executable code is contained in the *Code* section in a machine dependent format. External references within the code which have to be relocated by the loader are described by *Links*. Finally, *Ref* contains the run-time type information necessary for generic access to objects.

The most important deviations to standard Oberon object files are the generalization of the command section and the absence of the type and pointer section. The new command section (*Directory*) contains not only the names and entry points of commands but information for all exported objects. The directory consists of a set of pairs (entry number, reference position) which describes for each exported entry the location of the associated reference information, which includes the name of the object. The absence of the type and pointer sections will be explained in the following section.

5.3.2 Run-time Organization

In our particular implementation for SPARC–based computers, modules are represented internally as a single storage block consisting of several subobjects and allocated outside the Oberon heap. This organization helps to reduce the number of objects on the heap and avoids any restriction implied by the heap size. Moreover, allocating procedure objects on the Oberon heap would require to split them up into two objects, one being of a fixed size (record) and the other of a variable size (open array). Activation of a procedure would then require an additional indirection which is clearly unacceptable (a similar splitting would be required for type objects since they also contain a variable size part, viz. the pointer-offset table used for garbage collection). By representing procedures (and types) as subobjects, the variable size part can immediately follow the header. Fig. 5.6 shows the layout of module blocks. It starts with a fixed part defined in the *LibraryDesc* and *ModuleDesc* records which contains also the necessary pointers to various subsections following the header.

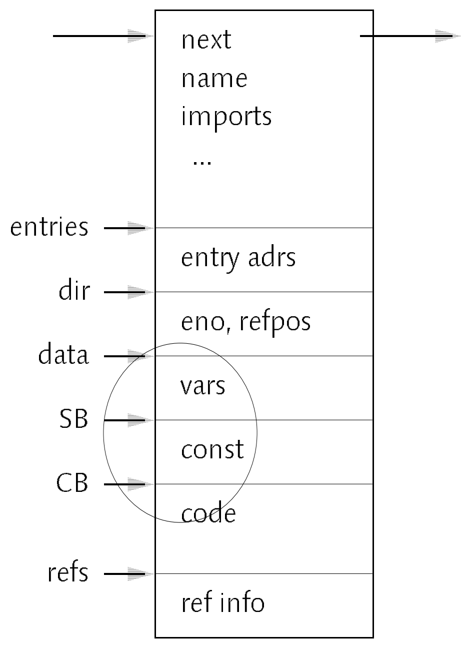


Fig. 5.6 – Structure of a Module Block

The entry table contains the absolute addresses of all exported type and procedure objects. The directory provides a mapping from entry numbers (an index in the entry table) to reference information by means of a pointer into the reference section. Fig. 5.7 shows more details of the marked part in Fig 5.6. In contrast to the original Oberon implementation, the reference block is not loaded on demand (e.g. by the trap handler), but permanently present within the module block. This storage overhead is easily justified by the improved access time to reference information. In a meta-level architecture, this access time must of course be short.

Subobjects of module blocks include an object which represents the global data of the module, record type descriptors and procedure objects. The global data object is not a proper subobject (c.f. Section 5.1.3) but has to be unmarked before every garbage collection. Type objects are allocated within the constant section, and procedure objects are allocated within the code area.

Note: Allocating type descriptors as subobjects within the constant area turned out to simplify code generation since access to types can be treated exactly like global data (or constant) access without the need for an additional indirection. It also helped to avoid a restriction in the number of pointers per record type. The only remaining restriction is the total size of the constant section. Another simplification is in the run-time system where the fact that Oberon-2 type descriptors grow in two directions (pointer table and method table) would complicate garbage collection of type descriptor objects.

Every subobject has a type tag, which points to the appropriate type descriptor object. For global variables of a module, a type descriptor is introduced which is used mainly to describe the position of global pointers. Therefore it is neither necessary to have a separate pointer offset table in the object file nor to treat global variables in a special way during garbage collection.

Module nodes themselves have a type tag pointing to the type object *Modules.ModuleDesc*. Thus, module *Modules*, which exports this type, describes itself by means of a type tag pointing to one of its own subobjects.

Type objects have a tag which points to the type object *Modules.TypeDesc*. Consequently, the object *Modules.TypeDesc* describes itself!

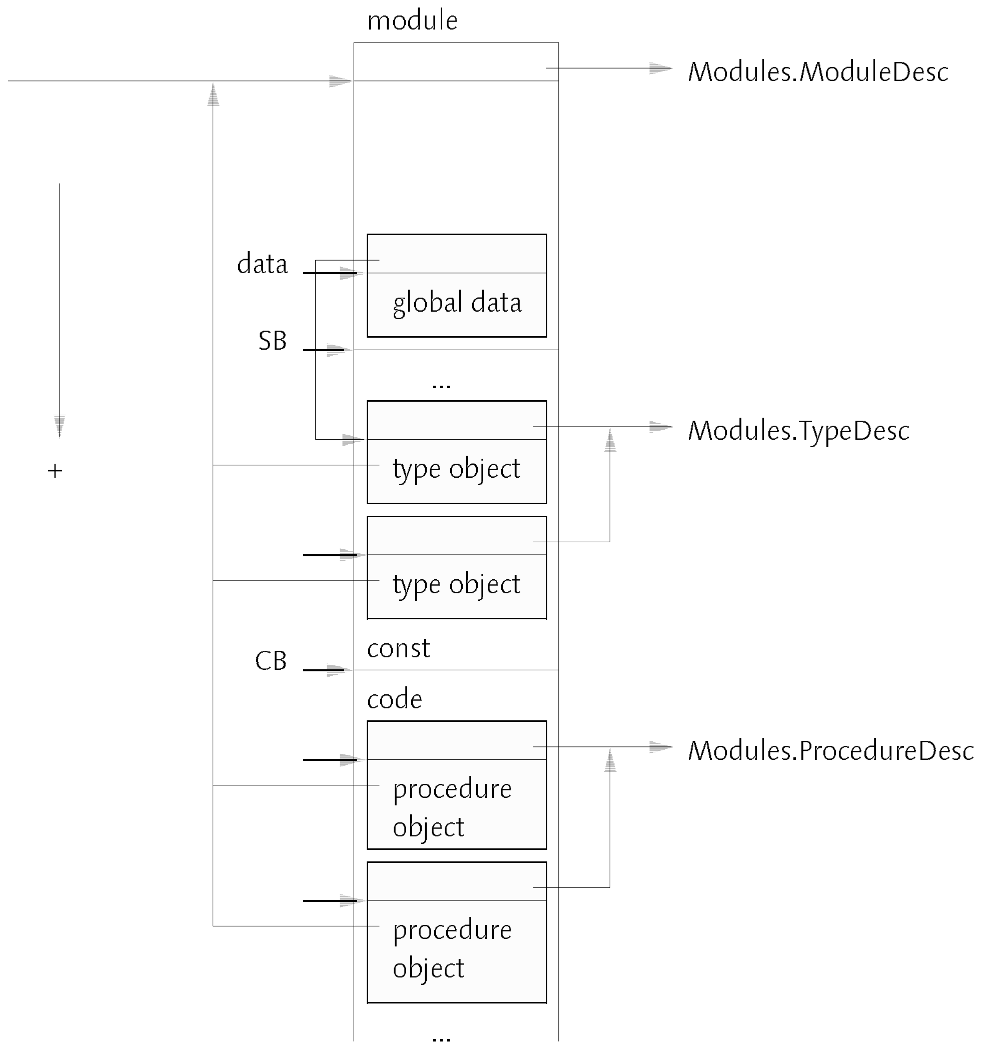


Fig. 5.7 – Subobjects in a Module Node

5.3.3 Reference Information

The data contained in the reference section provides run-time access to type information. The structure and contents of the reference section might be considered as a simplified symbol file [Gut85]. In contrast to the latter, it is not self-contained and does not contain objects such as modules or constants. An obvious commonality of symbol files and reference information is the linearized form of the data. The following EBNF grammar describes the structure of the reference information.

```
Ref = {TypRef | ParamRef | ActivationRef}.
TypRef = 0F7X refno name {name off Type} 0X.
ParamRef = 0F8X refno name {mode name off Type} 0X.
ActivationRef = 0F9X endpc name {mode name off Type} 0X.
name = len {char} [exported | readOnly].
mode = Var | VarPar.
Type = form [Pointer | Procedure | Array | Record | DynArr].
Pointer = Type.
Procedure = key.
Array = length elemsize Type.
DynArr = elemsize Type.
Record = mno tdoff.
```

The reference section has been designed in a way which allows efficient sequential access to the type information. In particular, it uses Pascal-style strings with a leading length byte to enable fast skipping of unused names without giving up a simple sequential format. The export mark belongs also to the name, however, it is omitted for parameters and local variables. Short numbers are encoded in a single byte, large numbers (off, endpc, key, length, elemsize, tdoff) use the variable length integer representation which is also used in the standard Oberon *Files* module.

There are only three different kinds of objects in the reference section. Record types (TypRef), parameter lists (ParamRef) and procedure activation records (ActivationRef). Note that the global variables of a module are described by the record type of the global data object.

Record types consist of a reference number, a name and a list of record fields. Every field consists of a name, an offset within the record and the type. The type consists of a form and optional form-specific attributes. Pointer types have a base type, procedure types have a fingerprint, arrays have a length, an element size and an element type, record types are described by a module number and a type descriptor offset and open arrays have an element size and an element type. The fingerprint of procedure types encodes the procedure's signature and can be used as a version key. Since calculating good fingerprints is a research topic on its own [Cre94], we used a very simple technique essentially as a placeholder for future improvements. The pair (mno, tdoff), which describes record types, contains the module number relative to the enclosing module (say *m*) and the type descriptor offset relative to the start of the referenced module's code section. More precisely, module number zero means *m* itself, and any module number $n > 0$ means module *m.GetImport*(n –1).

Parameter lists consist of the reference number of the associated procedure, the name of this procedure and a list of parameters which serve as actual parameters. Every parameter has a mode such as Var or VarPar, a name, an offset within the parameter block and a type.

The reference information for activation records consists of a program counter value, the name of the activated procedure and a list of formal parameters and local variables. The program counter is used to identify procedure activation records, which are untagged for efficiency reasons. Actually, the program counter value of the instruction following the procedure is stored (relative to the code base) since this simplifies this identification.

Note: Unfortunately, some of the reference information is actually redundant. Activation records might be seen as extensions of parameter records, which share the parameters as common objects. The problem encountered was that this requires to separate parameter objects and other local variables within a procedure's scope. In the portable Oberon compiler all parameter objects are sequentially linked, however, local variables are sorted in a binary tree together with parameters. Extracting the local variables from the procedure scope, which contains both parameters and local variables, seemed to be an unjustified complication. Moreover, addresses of parameter objects (off) might be different in parameter records and in activation records or in other words, from outside and inside of a procedure. If a parameter is for example passed in a register, the callee may store it to memory and use the memory address to access it. Therefore, two addresses would have to be maintained anyway. It should be noted that parameter records are provided for exported procedures only, which further reduces the redundancy.

The price to be paid for full run-time type information is the increase of the size of the reference section. Table 5.1 shows that the size is approximately twice as large as in a standard Oberon implementation with run-time type information reduced to unstructured local and global variables.

Module	reduced	full
Texts	2026	4755
TextFrames	3833	6621
Oberon	1123	2848

Table 5.1 – Size of Reference Sections

5.3.4 Module Loading

Loading of modules is enabled by installing a module specific *This*–procedure (the module loader) in the library loading framework. Since the modules *Libraries* and the *Modules* must already be loaded for this task, we have conflicting requirements for the order in which libraries are to be loaded. Such a conflict is commonly known as a bootstrapping problem. The conflict can be resolved in two ways. One is to implement a boot linker, which takes a specified set of libraries and links them into a boot file from where the libraries can be easily loaded by means of a primitive boot-loader. The other possibility is to write a less primitive boot loader which already includes the module loader. For reasons of simplicity we adopted the second strategy and implemented the boot loader in Modula-2. This Modula-2 program can be linked and executed without the need of any Oberon modules. The task of the boot loader is to execute the command Oberon.Loop, i.e. to load module *Oberon* and all directly or indirectly imported or loaded modules, and then to enter the central Oberon main event loop.

> Note: One disadvantage of this solution was that it became apparent that the particular Modula-2 implementation (Sun Modula–2) was not very well suited for low-level programming tasks, which are needed to implement a module loader. It was for instance not possible to read and write directly from untyped memory as it is possible in typical Oberon implementations by means of SYSTEM.GET and SYSTEM.PUT operations. One had to use pointer types and type casts to achieve the same effect. Furthermore there were a number of inconveniences in the library which made life harder than necessary. Almost every library operation had its result parameter defined as an enumeration type. All in all, this led to an unnecessary verbosity and to the feeling that the missing enumeration types in Oberon are a real progress. Another problem with the chosen approach was that it required some extra communication between the Modula-2 loader and the Oberon modules which constitute the library loading framework. Although not particularly complex, this sort of programming is on an inherently unsafe level and therefore notoriously dangerous.

The structure of the module loader essentially follows the one described in [WG92] but with the deviations described in Section 5.1.1 (A Framework for Library Loading). The program fragment below outlines the recursive structure of the loader's core. As an optimization for the case of cyclic loads, an additional *Lookup* has been introduced to avoid multiple allocation of module nodes. Note that superfluous nodes would not affect the uniqueness property of the set of accessible libraries (due to the guard in the library loader) but would fragment the module area and slow down the loading process.

```
PROCEDURE ThisMod(name: ARRAY OF CHAR): Library;
  VAR imp: ARRAY maxLib OF Library; ...
BEGIN L := NIL;
  open object file name.Obj;
  IF object file exists THEN read header block;
    IF res = Done THEN i := 0;
      WHILE (i < nofImports) & (res = Done) DO
        ReadLInt(impkey); ReadBytes(impname, 20);
        imp[i] := This(impname);
        IF (res = Done) & (imp[i].key # impkey) THEN res := InvalidKey END ;
        INC(i)
      END ;
      IF res = Done THEN Lookup(name, L);
        IF L = NIL THEN allocate module node L;
          IF L = NIL THEN res := OutOfMemory
          ELSE initialize L; read further sections; fixup links
          END
        END
      END
    END ;
  ELSE res := libNotFound
  END ;
  RETURN L
END ThisMod;
```

One additional check would still be possible, viz. to ensure that all libraries imported by a module are modules as well. We expect, however, that this property is encoded in the library's key and report an *InvalidKey* error if the type does not match.

Although we mentioned that cyclic imports are not allowed in Oberon, they can be constructed by means of a sequence of compilation and editing steps. We deliberately refrain from explaining this sequence in order not to motivate the reader to try this out. In the presented library loading framework, an endless recursion would occur which is terminated only by exhaustion of any of the consumed resources such as the procedure activation stack, file descriptors or heap space. If using a Unix file system, consuming all file descriptors is by far the most likely reason to terminate the loop since typically only 64 file descriptors are available to a Unix process. Thus, running out of file descriptors could be used as a poor-men's heuristics for detection of cyclic imports.

5.3.5 Garbage Collection

Since modules consist of a number of subobjects, we have to override the

Unmark method (see Section 5.1.3), which is necessary for correct reachability analysis of subobjects, and think of the subobject properties of these objects. The set of subobjects of a module is comprised of type descriptors, procedures, and the global data object. Not all of them are proper subobjects. In the following we shall discuss the different kinds of subobjects and their references in more detail.

Type Objects

Type objects contain the following three different kinds of references

- references to type-bound procedures
- a reference to the defining module
- a type tag

References to type-bound procedures are contained within the method table which is associated with an Oberon-2 type descriptor. Entries of the method table may be methods defined in the defining module itself or inherited methods defined for a base type in an imported module. The reference to the defining module (via field *lib*) provides the requested back pointer to the enclosing compound object. The type tag of a type object constitutes an implicit reference to object *Modules.TypeDesc*, which is assumed to be a subobject of module *Modules*. Beside the fact that it is hardly meaningful to unload *Modules*, it is always reachable via any other module's type tag, which points to object *Modules.ModuleDesc*. This implies that type objects can be safely regarded as proper subobjects.

Procedure Objects

Procedure objects have also three different kinds of references

- a type tag
- a reference to the defining module
- a reference to the handle procedure

For the type tag and the reference to the defining module, the same argumentation holds as for type objects. As long as procedures are passive objects (the regular case), the *handle* field is NIL. Active procedures have an arbitrary *handle* field, thus they cannot be regarded as proper subobjects. Therefore, changing the handler has to be done under the control of a special procedure (InstallHandle) in order to keep track of the subobject property. A simple solution is to introduce a counter per module (*nofActiveProcs*) which defines whether the module has active procedures or not.

Global Data

Since the global data of a module can hold arbitrary references, this object is in general not a proper subobject. The global data object is also missing a direct back pointer to the module node. Fortunately, the back pointer is available indirectly via the *lib* field of its type descriptor object. In order to guarantee reachability of the module node via the global data object, not only the global data object must be unmarked before every garbage collection, but also its type descriptor object although the latter is a proper subobject.

The following method implements unmarking of modules.

```
PROCEDURE (M: Module) Unmark (all: BOOLEAN);
   VAR i: INTEGER; o: REFANY;
BEGIN
   unmark(M.data); unmark(Type(M.data));
   IF all OR (m.nofActiveObjects > 0) THEN
      FOR i := 0 TO m.nofObjects – 1 DO M.GetObj(i, o); unmark(o) END ;
   END
END Unmark;
```

5.4 Fonts

Compared to modules, Oberon's bitmap fonts are much easier to handle since they are (at least in Oberon) always self contained, i.e. they don't import other libraries. Thus, font loading mainly consists of reading a font file and building the internal character patterns. There is only the problem that fonts consist of many objects. Thus, they would introduce a large number of objects on the heap if there were no optimizations. Note that every character is represented by two objects, one with a fixed size to hold the metric information and another with a variable size to hold the raster image of the character. A number of different implementations exist to optimize organization of fonts.

Don't treat characters as objects

In standard Oberon implementations, characters are not considered to be objects exported by a library. Therefore, access to metric information is provided by means of procedure *GetChar*, which returns the metric information rather than an object representing the character.

```
PROCEDURE GetChar(f: Font; ch: CHAR;
   VAR dx, x, y, w, h: INTEGER; VAR p: LONGINT);
```

Surprisingly, this procedure is exported from module *Display* rather than from module *Fonts* as one would expect. The font parameter is of type *Display.Font* which is a component of *Fonts.Font*. The main reason for this design is efficiency since in the original Oberon implementation on Ceres the *Display* module was implemented in assembly language (which allowed a slightly more efficient access to metric information) while module Fonts was implemented in Oberon.

A second anomaly is the treatment of the raster image *p* as a LONGINT variable. Conceptually speaking, *p* is a weak pointer to a subobject which represents the raster image. Weak pointers are to be ignored by the garbage collector. However, it is possible to generate dangling references with this arrangement if fonts as a whole are subject to garbage collection.

Treat characters as shared objects

Oberon System-3 has introduced fonts as libraries and characters as exported objects. In order to avoid a large number of character objects, there is only one globally defined object into which the metric information is copied before a pointer to this object is returned. Since pointers to characters are rarely stored, this optimization is in practice transparent to the user. Pointers to character patterns are handled in exactly the same (unsafe) way as in standard Oberon.

Treat characters and character patterns as subobjects

A more consequent approach is to treat all characters as individual (unshared) objects and patterns as regular pointers. The introduced memory overhead is easily justified by improved run-time efficiency since when the metric information of a character is accessed, only a single pointer has to be passed instead of copying all individual fields of the metric record. There is no need any more to implement something like *GetChar* in assembly language. The speed improvement is especially important in the context of multi pass text formatters (e.g. for WYSIWYG text formatting and automatic line breaks), where multiple accesses to printer and display character objects are necessary but only a few fields are actually used. The introduced storage overhead compared to the Ceres version and a standard Oberon implementation based on Sun's *pixrect* library is shown in Table 5.2. The pixrect implementation needs significantly more space than Ceres Fonts due to different alignment requirements (32-bit alignment for every raster line) and a more general (device independent) pattern format. The additional overhead for treating characters as objects is fairly small and by no means prohibitive. External constraints such as the required memory bitmap layout for a given graphics library dominate the effect of introducing characters and patterns as objects.

Font	Ceres	pixrect	pixrect subobjects
Syntax8.Scn.Fnt	1931	11228	11988
Syntax10.Scn.Fnt	2178	10724	12708
Syntax12.Scn.Fnt	2238	11200	12464
Syntax16.Scn.Fnt	2816	11900	13128
Syntax20.Scn.Fnt	3556	12492	13720
Syntax24.Scn.Fnt	4836	13368	14632
Syntax24b.Scn.Fnt	5314	14004	15268

Table 5.2 – Font Size in Bytes

Garbage collection efficiency is only affected in the exceptional case of unloading fonts, where all subobjects have to be checked for being referenced. A subtle point is the necessary back pointer from any subobject to the parent object. In case of patterns, this pointer does not exist directly but can be introduced indirectly via a specially prepared type descriptor object which points back to the font node via its lib field. All pattern objects of a font can share the same pattern descriptor. Fig. 5.8 outlines the layout of fonts.

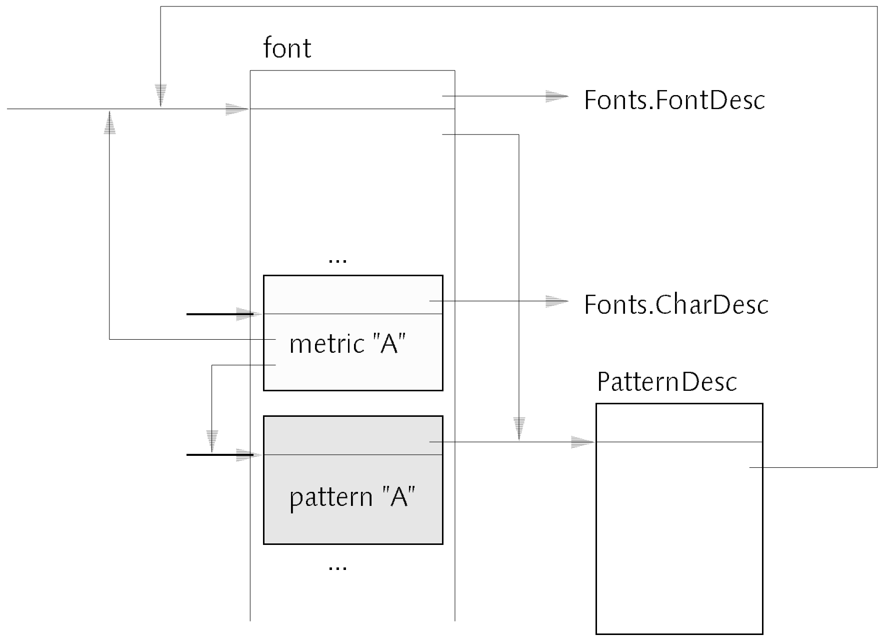


Fig. 5.8 – Fonts as Compound Objects

5.5 Generic Access to Arbitrary Objects

As introduced in Section 4.2.3, generic object access is based on the concept of object riders. Sequential riders correspond naturally with the linearized form of the type information provided in the reference section. It should not be necessary to build up complex data structures such as a compiler's symbol table for the task of generic object access. Maintaining such data structures would require significant heap space. Moreover, such an approach would increase the number of objects on the heap, thereby slowing down the garbage collector. In the case of our mark and scan collector, both phases would be affected. Note that a compiler has to build a temporary symbol table for only one module whereas for generic object access we would have to keep the tables of all loaded modules permanently in memory. Sequential access to objects turned out to be the dominant access pattern, therefore, the sequential iteration has never been felt as a restriction.

The algorithmic solution of generic object access is rather simple. The only complication is that it works mostly on untyped memory and that some tuning measures have to be taken in order to reduce the run-time overhead. In addition to the exported fields of object riders, several private fields have to be introduced. In the following, we describe this private data and outline the problems encountered.

```
TYPE
   Rider = RECORD
      public fields
      base: REFANY;
      adr, off, paroff, info: LONGINT;
      mod: Modules.Module
   END ;
```

base

keeps a reference to the object a rider is riding on in order to prohibit garbage collection of the object as long as a rider on this object exists. The unmodified *base* field is passed to all riders which zoom into a structured component of an object.

adr

holds the start address of the object a rider is iterating on. In case of zooming into a structured component of the object, the *adr* field is changed to the start of the component.

off

holds the offset of the variable at the rider's current position from *adr*. Thus, (*adr* + *off*) gives the address of the variable which corresponds to the position of the rider. Field *off* is changed whenever a rider is advanced to the next position.

paroff

Field *paroff* is used to manipulate VAR-parameters (e.g. in activation or parameter riders). VAR-parameters are passed by reference, i.e. only a pointer to the start of the parameter is passed. Depending on the kind of parameter, additional values such as the actual record type or length information for open arrays are passed. (adr + paroff) gives the address of the reference, which is passed for a VAR-parameter. In contrast to this, (adr + off) gives the address of the VAR-parameter.

info

the address of the type information in a module's reference section corresponding to the rider's position. The type information is read sequentially and the *info* value is updated whenever the rider is advanced to the next position except for array riders, where the *info* field is invariant. Every type descriptor object contains an *info* field, which points to the start of the corresponding reference information. When opening a record rider, the *info* field is taken from the object's type descriptor. To simplify the implementation of riders, the *info* field always points after the fixed part of a type, i.e. after the *form* information (see Fig. 5.9 below).

mod

a reference to the module which exports the type of the object the rider is riding on. More precisely, *mod* is an abbreviation for *t* := *Type*(*base*); *mod* := *t.lib*(*Modules.Module*). The reference section contains record and pointer base types as pairs (*mno*, *tdadr*) where *mno* is the module number relative to the exporting module *mod*. Module number zero means the exporting module itself, and any module number *n* > 0 means the result of calling *mod.GetImport*(*n* – 1). *tdadr* means the offset of the record type object from the exporting module's code base.

Accessing the current location of a rider is done as shown for the example of integer values below. A method *Advance* is assumed, which advances the rider's position to the next one. *Advance* is rider specific, i.e. it is different for riders on arrays, records, procedure activations and parameter blocks.

```
PROCEDURE (VAR R: Rider) ReadInt (VAR x: INTEGER);
BEGIN ASSERT(R.class = Int); SYSTEM.GET(R.adr + R.off, x); R.Advance
END ReadInt;
```

Specific riders introduce additional fields as appropriate. *RecordRiders*, for example, introduce a *typ* field, which points to the corresponding type object. This field is necessary for switching from one extension level to the next via the type's base table and the additional public field *level*. They introduce also a field *location*, which is a pointer into the reference section for retrieving the field name in *GetLocation*.

```
RecordRider = RECORD (Rider)
   public fields
   typ: Modules.Type;
   location: LONGINT
END ;
```

Analogously, *ArrayRiders* introduce a private field which describes the size of the array elements.

Example

```
VAR
   a: POINTER TO RECORD
      x: LONGINT;
      r: RECORD y: REAL END
   END ;
   R0, R1: RecordRider;
BEGIN
   NEW(a); a↑.x := 47; a↑.r.y := 11;
   OpenRider(R0, a); R0.Skip;
   ZoomRecord(R1, R0);
```

The state of the computation after *ZoomRecord* is sketched in Fig. 5.9.

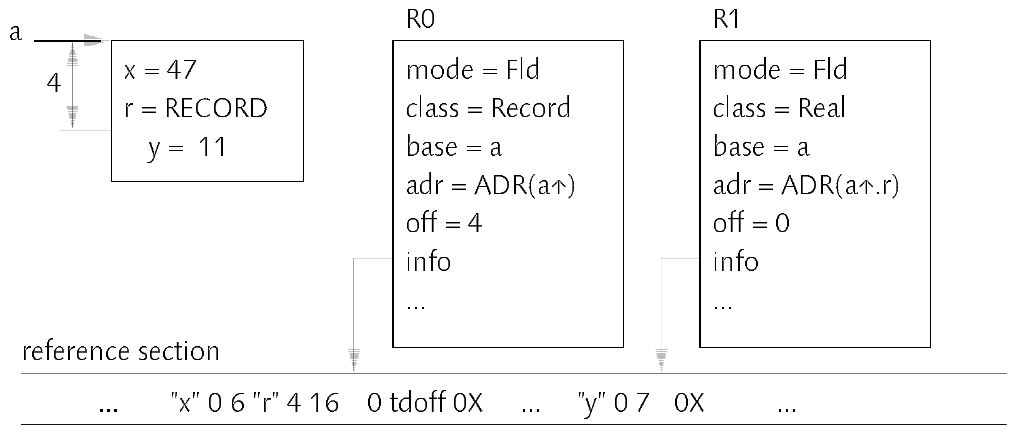


Fig. 5.9 – Nested Riders

5.6 Active Procedures

Implementing the facilities for controlling procedure activations is to a high degree machine specific, i.e. it depends on the machine architecture and the calling conventions being used. However, it is expected, that the general ideas behind the following approach, which has been implemented for SPARC-based machines, could be used at least for other reduced instruction set computers (RISCs) as well.

Parameters

At the center of implementing the facilities for controlling procedure activations are parameter records. These represent the actual parameters of a procedure call and are initialized by procedure *GetParams*. On a RISC machine, parameter passing is usually done both in registers and in memory. The first parameters are passed in registers, subsequent ones are passed in memory allocated on the procedure activation stack. In SPARC-Oberon, six integer registers and six floating point registers (or three floating point register pairs) are used for parameter passing. Function results are returned in an integer register or one or two floating point registers. The layout of parameter records reflects these calling conventions directly.

```
TYPE
  Parameters = RECORD
    ires: LONGINT;
    fres: LONGREAL;
    ireg: ARRAY 6 OF LONGINT;
    freg: ARRAY 3 OF LONGREAL;
    mem: ARRAY 6 OF LONGINT;
    p: Procedure
  END ;
```

Each procedure has its own kind of parameter record according to the parameters of the procedure. To allow checking of the correspondence between a procedure and a parameter record, every parameter record must be marked with the procedure it corresponds to. Field *p* has been introduced for this purpose. Procedure *GetParams* initializes *p* and sets all other fields of parameter records to zero. Since parameter records have a fixed maximum size, there is no allocation of dynamic memory involved.

Note: The official SPARC application binary interface specification (ABI) requires to pass floating point values in integer registers. Following these guidelines would simplify the above layout of parameter records (no need for field freg) but slow down parameter passing in some cases.

When iterating over parameter records, the meta-level programmer does not want to see the internal representation of parameter records but wants to see the list of parameters as declared for a particular procedure. Procedure *OpenParam* provides this mapping by setting the *info* field of the initialized *ParamRider* to the corresponding *ParamRef* entry in the reference section rather than to the *TypeRef* entry of type *Parameters*.

Eval

In a programming environment which is based on compilation rather than interpretation, procedure *Eval* has to set up an appropriate stack and register context and to transfer control to the compiled body of the procedure which is to be evaluated. The first activity of *Eval* is to check the correspondence between the parameter record and the passed procedure. The rest of *Eval* is also not particularly difficult, but of course somewhat machine specific. The trick to simplify passing of all the different register and memory parameters is to introduce an auxiliary procedure variable with as many parameters as ever reasonable. Transfer of control is done by calling this variable after it has been set to the body of the procedure to be evaluated. The body is immediately following the fixed size part of a procedure object. The evaluated procedure will silently ignore all unused parameters if removal of parameters from the activation stack is the task of the caller and not the callee. In our implementation for SPARC-Oberon, we support up to 18 parameters. At most six of them are passed as integer registers, at most six are passed as floating point registers (or three floating point register pairs) and the rest is passed in memory. Since the auxiliary procedure variable is not a proper procedure variable but points inside a procedure object, it has to be marked as untraced (expressed by the [1] mark in the declaration) to avoid any problems with garbage collection. After the procedure has been evaluated, possible function result values are stored in fields *ires* and *fres* respectively (register numbers > 31 refer to floating point registers).

```
PROCEDURE Eval (P: Modules.Procedure; VAR par: Parameters);
   VAR
      p: PROCEDURE [1] (i0, i1, i2, i3, i4, i5, m0, m1, m2, m3, m4, m5: LONGINT;
            f0, f2, f4: LONGREAL);
BEGIN
   ASSERT(par.p = P);
   SYSTEM.PUT(SYSTEM.ADR(p), SYSTEM.ADR(P↑) + SIZE(Procedure));
   p(par.ireg[0], par.ireg[1], par.ireg[2], par.ireg[3], par.ireg[4], par.ireg[5],
      par.mem[0], par.mem[1], par.mem[2], par.mem[3], par.mem[4], par.mem[5],
      par.freg[0], par.freg[1], par.freg[2]);
   SYSTEM.GETREG(8, par.ires);
   SYSTEM.GETREG(32, par.fres);
END Eval;
```

It should be noted that this schema makes the implicit assumption that structured value parameters are copied by the callee, i.e. within the called procedure and not before the call. Therefore, such parameters are internally passed by reference, which takes a small, constant amount of memory per parameter. If structured values had to be expanded into the parameter record, the simple layout with a fixed size would not be practical. We claim that assuming a copy-by-callee schema is justified since it is the more general technique. This can be seen by the fact that it can also be applied to open array parameters. Furthermore, copy-by-callee results in increased code density, because structure copying is done in one place only.

Independent of copy-by-caller or copy-by-callee, another subtle problem exists, namely the fact that parameter records contain weak references (VAR-parameters, pointers, procedure variables). Thus, it is in principle possible to create dangling references by inappropriate use of parameter records. In practice, however, this was never a problem since parameter records are typically allocated as local variables and never assigned to non-local variables.

Filters

Before we shall look at the more general problem of implementing active procedures, we show a possible implementation of the concept of *Filters* as introduced in Section 4.2.4. The main reason why this concept has been introduced in the basic system is that it requires additional state per procedure to anchor the filter chain. Therefore, we introduce an additional field *f* into procedure objects, which points to a cyclic list of filter nodes.

```
TYPE
   Procedure = POINTER TO ProcedureDesc;
   ProcedureDesc = RECORD (Libraries.ObjectDesc)
      f: Filter
   END ;

   Filter = POINTER TO FilterDesc;
   FilterDesc = RECORD
      link: Filter;
      p: Procedure
   END ;
```

The concrete situation of Fig. 4.7 (a procedure with two filters) corresponds to the data struture outlined in Fig. 5.10.

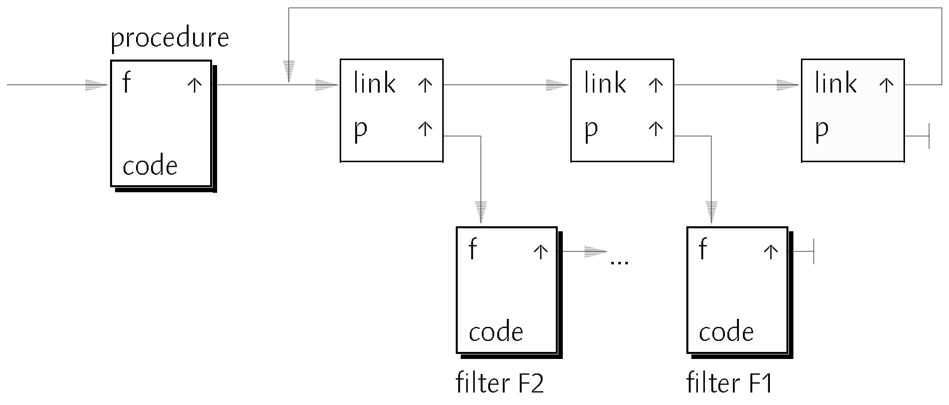


Fig. 5.10 – The Filter Chain

A possible implementation of filters is given in the procedure handler *FilterHandle* outlined below. The presented implementation is fully portable. However, by giving up portability, it could be tuned to avoid the overhead of calling *Eval* explicitly. In practice, it turned out that the presented implementation is sufficiently efficient.

```
PROCEDURE FilterHandle (P: Libraries.Object; VAR M: Libraries.ObjectMsg);
  VAR f: Filter;
BEGIN
  WITH P: Procedure DO
    WITH M: InvocationMsg THEN
      f := P.f; P.f := f.link;
      IF f.p = NIL THEN M.par.p := P; Eval(P, M.par)
      ELSIF f.p.handle = NIL THEN M.par.p := f.p; Eval(f.p, M.par)
      ELSE f.p.handle(f.p, M)
      END ;
      P.f := f
    ELSE
    END
  END
END FilterHandle;
```

Active Procedures

Installing a procedure handler into a procedure object requires to provide a mechanism that allows intercepting any calls to this procedure. Different strategies exist for this task, among them are the usage of an additional indirection for procedure calls via a link table or patching the entry code of the called procedure in an appropriate way. Since we tried to minimize the overhead on conventional programming tasks and since there is no hardware support for indirect procedure calls on SPARC, we implemented the latter alternative. We introduce a number of dummy instructions at the beginning of a procedure's code, which can be overridden by a call to an intercepting procedure. Although not unconditionally necessary, these dummy instructions help to simplify the implementation and to avoid undue machine dependencies. Fig. 5.11 shows the layout of passive and active procedure objects for SPARC-Oberon.

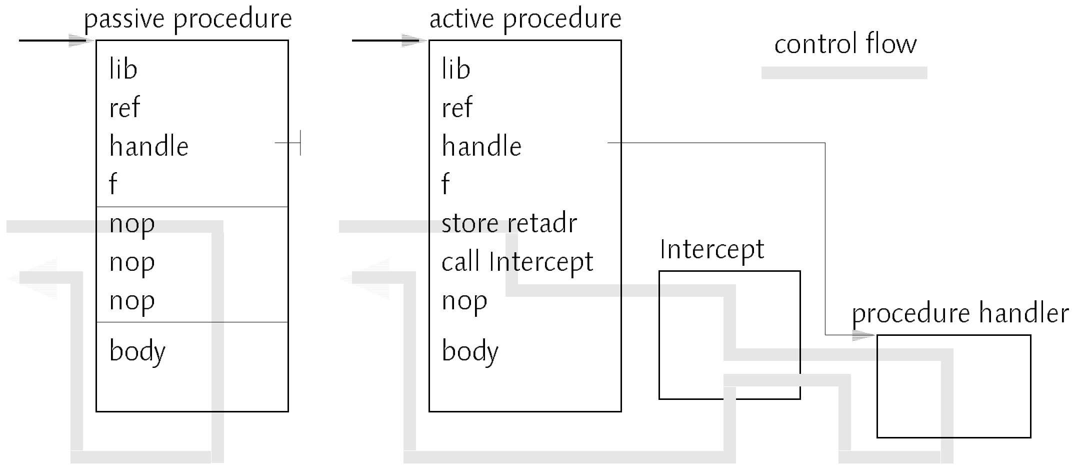


Fig. 5.11 – Passive and Active Procedures

The final form of the fixed size part of a procedure object is defined as:

```
TYPE
   ProcedureDesc = RECORD
      (Libraries.ObjectDesc)
      f: Filter;
      code: ARRAY 3 OF LONGINT
   END ;
```

Active procedures have to save the return address first and then to transfer control to an interceptor. Due to the particular SPARC architecture, it is necessary to have a dummy instruction, the so-called delay slot, after the call. This instruction cannot be used to save the return adress due to peculiarities in the semantics of delayed branches of SPARC. In total, we need to reserve three nop (no operation) instructions at the beginning of an external procedure, which implies only a very small run-time and storage overhead for exported procedures.

Similar to the auxiliary procedure variable used to implement *Eval*, the interceptor has as much parameters declared as reserved in parameter records. This guarantees that parameters passed in registers are not accidentally changed within *Intercept* and they can be easily accessed and stored into a parameter record. In addition, the interceptor has to fixup the return address such that the intercepting call is skipped when returning from it. In SPARC-Oberon, register 31 holds the return address of a call instruction, and register 1 holds the return address stored in the intercepting code. The auxiliary variable *p20* holds the address of the "call Intercept" instruction, which is at offset 20 from the beginning of the procedure object. After the constructed invocation message has been handled, a possible result value is returned or put into the appropriate floating point registers, which are not changed upon return.

```
PROCEDURE Intercept(i0, i1, i2, i3, i4, i5, m0, m1, m2, m3, m4, m5: LONGINT;
        r0, r2, r4: REAL): LONGINT;
VAR p20, ret: LONGINT; P: Procedure; M: InvocationMsg;
BEGIN
  SYSTEM.GETREG(31, p20);
  SYSTEM.GETREG(1, ret);
  SYSTEM.PUTREG(31, ret);
  P := SYSTEM.VAL(Procedure, p20 – 20);
  M.par.p := P;
  M.par.ireg[0] := i0; M.par.ireg[1] := i1; M.par.ireg[2] := i2;
  M.par.ireg[3] := i3; M.par.ireg[4] := i4; M.par.ireg[5] := i5;
  M.par.mem[0] := m0; M.par.mem[1] := m1; M.par.mem[2] := m2;
  M.par.mem[3] := m3; M.par.mem[4] := m4; M.par.mem[5] := m5;
  M.par.freg[0] := r0; M.par.freg[1] := r2; M.par.freg[2] := r4;
  P.handle(P, M);
  SYSTEM.PUTREG(32, M.par.fres);
  RETURN M.par.ires
END Intercept;
```

Measurements

On a 20 MHz SPARCstation1 with a unified data and instruction cache (write through), we obtained the following benchmark results. Note that due to mysterious cache behavior, the actual results may deviate by more than a factor of 10! We took those results that fit together in a reasonable way, i.e. we ignored extremely deviating values.

Operation	micro secs
Eval	6
Intercept	11
Empty Filter	19

Table 5.3 – Measurements of Active Procedures

Summary

This concludes the chapter on implementation aspects of the introduced meta-level architecture. It was surprising that everything could be implemented in Oberon itself partly by using Oberon's low level facilities and some knowledge about the underlying compiler. The set of available low level facilities (module SYSTEM) turned out to be sufficient for our purposes. We never found the need to use a more sophisticated in-line assembler or to escape to system programming languages such as C. In total, the low-level part is rather small compared to the fully portable part of the implementation.

6 Applications of Metaprogramming

This chapter describes several examples of applications which have been implemented with the introduced metaprogramming protocol. The emphasis is not put on outstanding functionality or sophistication of the individual examples but on the fact that they have been implemented by using the meta-programming approach. We claim that extending the functionality is only a quantitative difference whereas metaprogramming introduces a new quality. The user, i.e the programmer, benefits from this new quality by the ability to implement or customize tools which he/she simply could not do previously.

6.1 Overview

The presented examples can be divided into two groups, one focusing on generic object manipulation, the other on controlling procedure activations.

The generic object manipulation examples start with the implementation of parts of module *System*, which is the top level module of the Oberon environment and serves to configure and use the Oberon system. The subsequent examples have certain dependencies among each other. We shall proceed in a bottom-up order and present as the first application the mapping of arbitrary data structures to non-volatile memory or in other words the implementation of a sort of persistent objects. We will use this mechanism to implement the load/store functions in an extensible graphics editor and we shall use other generic object manipulations to support editing facilities without the need for type-specific code. The graphics editor will be used in a subsequent example to implement a tool for interactive two-dimensional data structure visualization.

Applications for controlling procedure activations can be divided into those which work for all procedures (Eval) and those which make use of active procedure objects. The latter cases can be further divided into examples for procedure handlers and filter procedures.

6.2 Module System

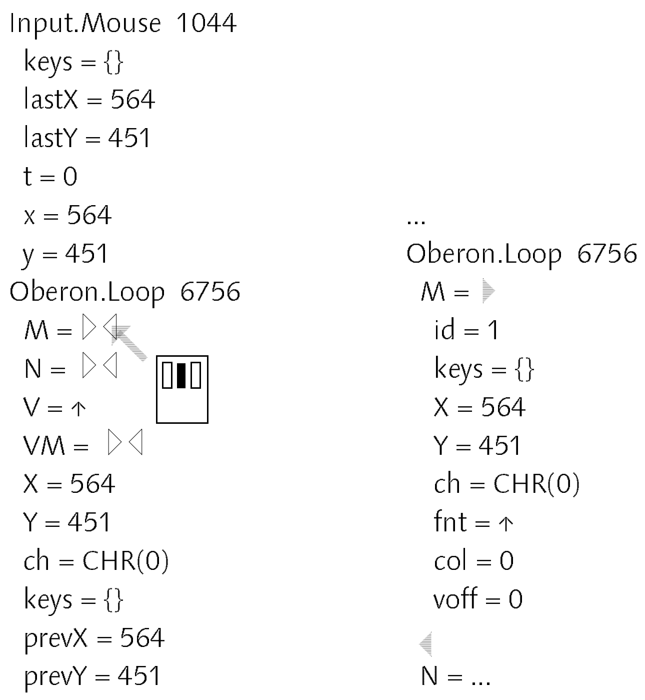
In standard Oberon systems, module *System* implements what the user sees of Oberon. It defines the startup screen layout, provides commands for handling viewers and other system resources and implements the trap handler and a facility for displaying the state of global variables (see also Section 3.3.2). The latter two tasks can benefit from metaprogramming as shown below.

Trap Handling

In case of an exception, the Oberon trap handler dumps the contents of the run-time stack to a text and displays it in a so-called trap viewer before transferring control back to the Oberon main event loop. Dumping the run-time stack can be implemented easily by employing *ActivationRiders* and the *WriteObj* procedure, which maps objects to a textual representation. Procedure *DumpStack* below is the core of the trap handler. Note that the module name *GenericObjects* has been abbreviated to *GO*. The initial values of stack pointer and program counter (sp, pc) are supposed to be provided by the system's interrupt handling mechanism, which will not be further described here.

```
PROCEDURE DumpStack(sp, pc: LONGINT; T: Texts.Text);
   VAR R: GO.ActivationRider;
BEGIN
   GO.OpenFrame(R, sp, pc);
   LOOP
      Texts.WriteString(W, R.module); Texts.Write(W, ".");
      Texts.WriteString(W, R.proc); Texts.WriteString(W, "  ");
      Texts.WriteInt(W, R.relpc, 0); Texts.WriteLn(W);
      GO.WriteObj(W, R, 0, 1); Texts.Append(T, W.buf);
      IF R.dlink > stackbot THEN EXIT END ;
      GO.OpenFrame(R, R.dlink, R.retpc)
   END
END DumpStack;
```

A stack dump may contain fold and reference elements and may look as shown in Fig. 6.1. Clicking the middle mouse button on a fold element (displayed by the fold brackets ▷◁) expands or collapses this element. For instance, expanding message record *M* in the activation frame *Oberon.Loop* leads to the expanded text shown to the right in Fig. 6.1.



```
Input.Mouse  1044
 keys = {}
 lastX = 564
 lastY = 451
 t = 0
 x = 564                          ...
 y = 451                          Oberon.Loop  6756
Oberon.Loop  6756                  M = ▶
 M = ▷◁                              id = 1
 N = ▷◁                              keys = {}
 V = ↑                               X = 564
 VM = ▷◁                             Y = 451
 X = 564                             ch = CHR(0)
 Y = 451                             fnt = ↑
 ch = CHR(0)                         col = 0
 keys = {}                           voff = 0
 prevX = 564                       ◀
 prevY = 451                       N = ...
```

Fig. 6.1 – A Stack Dump

Middle clicking reference elements (displayed as ↑) causes a new text viewer to be displayed which contains the referenced object. This simple mechanism essentially covers the functionality of traditional post mortem debuggers. An alternative tool which features a two dimensional representation of objects will be introduced in a subsequent section.

Displaying Global Variables

In standard Oberon systems, the command *System.State* takes a list of module names as parameters and dumps the values of the global variables of these modules to a text which is displayed in a text viewer. In our meta-level architecture, we allow arbitrary libraries as parameters and display the state of these libraries. Only in case of a module, we display the contents of the global data of this module. The core of the implementation consists of a library lookup, and in case of success the mapping of the state variables to a text. The output looks similar to stack dumps and contains also fold and reference elements for structured components and pointers respectively.

```
PROCEDURE State;
   VAR S: Texts.Scanner; W: Texts.Writer; L: Libraries.Library; R: GO.RecordRider;
BEGIN
   open scanner S, writer W, and a text viewer;
   WHILE S.class = Texts.Name DO
      Lookup(S.s, L);
```

```
        IF L # NIL THEN
          IF L IS Modules.Module THEN GO.OpenRider(R, L(Modules.Module).data)
          ELSE GO.OpenRider(R, L)
          END ;
          GO.WriteObj(W, R, 0, 1);
        END ;
        Texts.Scan(S)
      END ;
      append W to output text
    END state;
```

6.3 Persistent Objects

Folowing the terminology used in the field of object-oriented data base systems, implementation strategies for persistent objects can be divided into *declared* and *programmed* persistency. Declared persistency means that data structures are marked as being persistent by introducing a new storage class *persistent* in addition to storage classes *automatic* (local variables) and *static* (global variables). Besides the difference in their life time, persistent objects can be used in the same way as other objects. This approach requires language support and can be found in object-oriented data base systems such as Exodus with the integrated programming language E [RC93]. Another example is PCLOS [PA90], an implementation of declared persistency by using the CLOS metaobject protocol. Programmed persistency, on the other hand, means that mapping a data structure from or to non-volatile memory is actuated explicitly during execution of a program. Typically, all objects which are reachable from a given root object (the closure of this object) are mapped together. An example is the *Pickle* package available in Modula-3 [Nel91]. The persistent objects introduced in this section are similar to the latter, i.e. programmed and reachability based.

The implementation of this kind of persistency consists of two main subproblems, *data mapping* and *pointer swizzling*. Data mapping means to map atomic data such as integers or characters to an external representation (and back), pointer swizzling means to keep track of references between the nodes which comprise a data structure. We try to solve both problems in a generic way. However, in order to allow a wider range of applicability and reasonable performance in special situations, we must also allow customization of the mechanism in an application specific way. We shall introduce a class called *Map* which encapsulates the information necessary for pointer swizzling and customization. Before we describe maps in more detail, we shall outline some of the inherent problems of automatic persistency.

Closure Control

Following all references is sometimes not possible or meaningful. For example, if an object contains a reference to a font, it is sufficient to store the font name instead of the whole font. In general, any references to libraries or objects exported by libraries can be regarded as external reference, i.e. as a reference outside the data structure under consideration. The existence of the notion of libraries and persistent objects can be used to limit closures to a reasonable size. In fact, the notion of libraries is essential to automatic persistent objects. In many cases, it limits the closures to exactly those objects that would also be stored in a hand-crafted implementation. There may still remain some cases where a reference should be ignored or handled specially. An example is a reference into Oberon's display space (e.g a pointer to a viewer) which would cause all open viewers and associated data structures to become part of the closure. We decided to leave these rare cases to customization by the user.

Implicit Dependencies

Externalized objects may have implicit dependencies on other data structures. This relation can be expressed by an unsuspicious integer value, say, but could be disastrous if not handled properly. A prominent example are file objects, which may contain integers representing sector numbers. It is of course not sufficient to store and load sector numbers as long as data structures for sector management are not updated. For the concrete example of file objects, we decided to disallow them per default. In order not to rule out the important case of (file based) texts, which in Oberon almost has the character of a built in data type, we provide an appropriate default text handling.

Partially used Arrays

Sometimes, array structures are only used partially. There may be a counter outside an array which determines used an unused elements or there may be a sentinel in one of the array elements. The most popular example for the latter are character arrays which have a zero terminated string as contents. Since this case is so common, we decided to treat character arrays always as strings and to leave other special cases to customization by the user.

6.3.1 Maps

We introduce an abstract data type *Map*, which contains all necessary information for mapping a data structure to a file and vice versa. It contains also information necessary for customization of special types.

```
TYPE
   Map = RECORD END ;
   Mapper = PROCEDURE (VAR m: Map; VAR R: Files.Rider; obj: REFANY);

PROCEDURE (VAR m: Map) Init;
PROCEDURE (VAR m: Map) Reset;
PROCEDURE (VAR m: Map) WriteClosure (VAR to: Files.Rider; o: REFANY);
PROCEDURE (VAR m: Map) ReadClosure (VAR from: Files.Rider; VAR o: REFANY);
PROCEDURE (VAR m: Map) Register (t: Modules.Type; read, write: Mapper);
```

Procedure *Init* initializes a map with a default behavior. This implies that Oberon texts are handled appropriately and references to files are not allowed. Procedure *Reset* resets a map for reading or writing. Procedure *WriteClosure* may be used to externalize an arbitrary data structure rooted in *o*. Analogously, procedure *ReadClosure* may be used to internalize a previously externalized data structure. *ReadClosure* and *WriteClosure* maintain a set of already visited nodes in order to provide correct handling of arbitrary (possibly cyclic) data structures. These sets are cleared by the *Reset* operation.

We provide for overriding the default mapping by means of registering type specific *read* and *write* procedures (mappers). If the specified type *t* is at record extension level *n*, the registered mappers are responsible for handling only the fields introduced at level *n*. This is in contrast to the semantics of overriding in object-oriented programming, where the overriding procedure is responsible for the whole type and is even inherited by subclasses. The advantage of the finer grained overriding based on extension levels is that it can be done completely transparent to subclasses. If mappers are registered for a type *t*, all extensions of *t* inherit the default mapping, not the customized mappers of *t*. There is no need to register mappers for all subtypes of *t* in order to handle the extended fields. Fig. 6.2 outlines the situation.

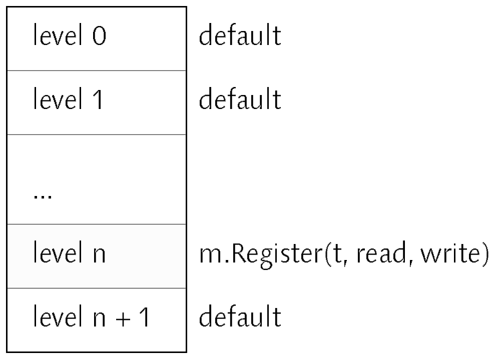


Fig. 6.2 – Customization based on extension levels

Customized mappers are allowed to call *ReadClosure* and *WriteClosure* recursively. In fact, that is the reason why we separated resetting a map from the *ReadClosure* resp. *WriteClosure* operation. Mappers have to be symmetric for reading and writing. Unfortunately, this symmetry can neither be guaranteed by the compiler not by run-time checks since files are an untyped array of bytes.

6.3.2 Implementation Aspects

The key for the implementation of maps is the file format being used. The requirements for this format are that it should be compact and it should allow efficient reading and writing where reading is considered to be the more important operation. We describe the external format by the EBNF-grammar below.

```
closure = NIL | ref | xref | typref [typinfo] {field}.
ref = 1 | 2 | 3 ... .
xref = libref | objref
libref = –1 name.
objref = –2 closure name.
typref = –3 | –4 | –5 ... .
typinfo = closure name.
field = simple | record | array | closure.
```

A closure is recursively defined as either the special value NIL, an internal reference, an external reference or a record consisting of type information followed by the record fields. Internal references (references to previously externalized objects) are encoded by positive numbers, external references by the negative numbers –1 and –2 and zero encodes the special value NIL. Types are encoded by negative numbers less than –2. Any externalized object (incl. references to libraries and exported objects) implicitly gets a sequence number and can be referred to by this internal reference number later on. This provides for a compact file format since these numbers are not explicitly stored on the file and there is no redundant information such as repeated library or object names.

Two kinds of external references exist, references to libraries (–1) and references to objects exported by libraries (–2). The former is followed by the library name, the latter by a closure representing the library and followed by the object's name.

Whenever an object's type is referenced for the first time, it is fully specified like an external reference of kind –2. The fields of an object may be of a simple type, a structured type, or it may be a reference to another object which is recursively treated as a closure.

The following outlines the structure of procedure *ReadClosure*, which uses a table of objects, a table of types, and counters for the number of objects and types (m.objtab, m.typtab, m.nofObjs, m.nofTyps). In order to avoid restrictions due to fixed table sizes, we used Oberon-2 open arrays and resize the tables in case of overflow (amortized doubling). For the sake of simplicity, we don't show this resizing in the procedure below.

```
PROCEDURE (VAR m: Map) ReadClosure (VAR from: Files.Rider; VAR o: REFANY);
   VAR ref, onum, level, h: LONGINT; name: ARRAY 24 OF CHAR;
      lib: Libraries.Library; lo: Libraries.Object; typ: TypeMap;
BEGIN
   Files.ReadNum(from, ref);
   IF ref < 0 THEN onum := m.nofObjs; INC(m.nofObjs);
      IF ref >= –2 THEN
         IF ref = –1 THEN Files.ReadString(from, name); o := Libraries.This(name)
         ELSE m.ReadClosure(from, lib); Files.ReadString(from, name);
            lib.GetRef(name, h); lib.GetObj(h, lo); o := lo
         END ;
         m.objtab[onum] := o
      ELSE ref := –ref;
         IF ref = m.nofTyps THEN
            NEW(typ); m.typtab[ref] := typ; INC(m.nofTyps);
            m.ReadClosure(from, lib); Files.ReadString(from, name);
            lib.GetRef(name, h); lib.GetObj(h, lo);
            typ.this := lo(Modules.Type);
            InitMappers(m, typ)
         ELSE typ := m.typtab[ref]
         END;
         GenericObjects.New(o, typ.this); m.objtab[onum] := o;
         FOR level := 0 TO typ.this.level DO typ.read[level](m, from, o) END
      END
   ELSE o := m.objtab[ref]
   END
END ReadClosure;
```

The mentioned type table essentially is an array of records, which contain the associated mapper procedures (typtab[n].read[level], typtab[n].write[level]) for a given type and a reference to the type itself (typtab[n].this). Procedure *InitMappers*(*m, typ*) is supposed to initialize the mappers for all extension levels of *typ* according to map *m*.

Writing data to a file is mostly symmetrical to reading from it. The most important difference is that the mapping from reference numbers to objects cannot be inverted directly. We used hashing to map objects and types from internal addresses to reference numbers. With load factors below 2/3, linear probing for collision handling turned out to provide the best results.

The depth first traversal of graphs, which we assumed silently, has one noticeable disadvantage, viz. the handling of degenerated graph structures such as sequentially linked lists. In this case, the algorithm uses many levels of recursion, one for each list element. This behavior can slow down reading or writing significantly due to bad data locality, which reduces cache effectivity and may cause swapping of memory to disk. It can also lead to stack overflow exceptions. Avoiding recursion by a breath first strategy, on the other hand, requires additional storage and is more complex. We decided to stay with the simple depth first strategy and to leave tuning of list structures to the user as we shall see in the next section.

6.4 Metaprogramming in an Extensible Graphics Editor

This section describes the experiment of using the introduced metaprogramming facilities and the persistency mechanism of Section 6.3 in an extensible graphics editor. Before we take a closer look at this editor, a few words about the general idea behind it may be in order. The extensible graphics editor is called *Kepler* [Te93] and is the outcome of an attempt to create an editor which can be extended with as less programming effort as possible. *Kepler* was also the first non-trivial program that was written with Oberon-2 style type-bound procedures and thereby served as a test bed for the introduced syntax and semantics. The usage of type-bound procedures instead of message handlers or explicit method records was a first step towards improved extensibility in terms of reduced programming effort. Another step was the chosen editing model, which is based on the idea that not every individual class should provide an *Edit*-method but editing should be done generically on the level of *handles*. In *Kepler*, every object depends on the position of a number of handles and changing a handle indirectly means to change one or more objects which depend on this handle. Fig. 6.3 shows a few object classes with the handles selected.

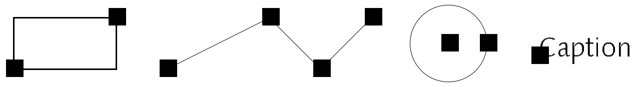


Fig. 6.3 – Kepler objects with selected handles

This editing model avoids the need for type specific editing methods and essentially reduces the number of methods to three – *Draw*, *Read* and *Write*. It is obvious that *Draw* is inherently type specific, but methods *Read* and *Write*, which account for 2/3 of the programming effort of many classes, are almost always trivial and can be replaced by the persistency mechanism described in the previous section. This is the third step of reducing the programming effort for extensions. The result is that only one method – *Draw* – has to be overridden for each concrete class.

In a real world graphics editor, it must also be possible to change the attributes of objects such as line width or fill pattern later on. Since these attributes are in general type specific, there must be type specific commands to set (or get) the individual attributes. In order to avoid these almost always trivial commands, we shall also employ metaprogramming for implementing generic attribute handling.

As a final point, we mention that there may be cases where objects would like to take control over the mouse or keyboard. This is the case for example if we want to use the graphics editor for composing graphical user interfaces with buttons or other sorts of end-user objects. Only in this case, the object must be able to respond to user input in a type specific way.

The reader might miss several functions such as printing, copying or selecting objects in the previous discussion. The solution is that printing is the same as drawing if a device independent display abstraction is used. This independence can also be used for getting the bounding box (size) of an object by drawing into a specialized display port which calculates the size of an object rather than actually drawing it. Copying of objects can be expressed by *Write* followed by *Read* (or by a generic deep-copy procedure). Similar to editing, selection can be done on the level of handles rather than on the level of objects. Thus, there is no other type specific behavior needed except for drawing.

Read/Write

A *Kepler*-graph consists of a list of handles and a list of objects. The handles are called stars and the objects are called constellations due to the analogy of the editing model to constellations in astronomy.

```
TYPE
   Star = POINTER TO StarDesc;
   StarDesc = RECORD
      x, y, refcnt: INTEGER;
      sel: BOOLEAN;
      next: Star
   END ;
```

```
    Constellation = POINTER TO ConsDesc;
    ConsDesc = RECORD
       nofstars: INTEGER;
       s: ARRAY 4 OF Star;
       next: Constellation
    END ;

    Graph = POINTER TO GraphDesc;
    GraphDesc = RECORD
       cons, lastcons: Constellation;
       stars, laststar: Star;
       seltime: LONGINT
    END ;
```

Although our persistency mechanism of Section 6.3 would work with the above data structures, we want to customize it to avoid a number of problems. One is the deep recursion which would occur due to the sequentially linked lists. Another is the role of the selection flag and the *refcnt* field (the number of constellations depending on a star), which are not necessary to store, and the last one is the array *s*, of which only *nofstars* elements are used.

Note: If we look at the mentioned problems more closely, we see that they are all in some way symptoms of a disease rather than the disease itself. Selection, for instance, could be regarded as a property of the view and not the model, which would remove the *sel* field from *Star* objects. The array of stars could be expressed as an open array, thereby being totally filled and avoiding a restriction on the number of stars. The sequential lists could be replaced by an appropriate two dimensional tree structure, which could also be used to speed up redraw operations significantly. Also, it would be more convenient for programmers of more complex extensions to eliminate the *refcnt* field. The reasons why we implemented it not this way is to keep the number of objects on the heap small and the editor's core simple. We note that applying the persistency mechanism did not introduce the problems but made them visible.

The following procedures have been used for customizing the persistency mechanism. (The type cast in the first line of each procedure is only necessary because the OP2 compiler [Cre90] does not allow type guards on untyped pointers.)

```
PROCEDURE WriteGraph (VAR m: Files2.Map; VAR R: Files.Rider; o: REFANY);
   VAR G: Graph; s: Star; c: Constellation;
BEGIN G := SYSTEM.VAL(Graph, o);
   s := G.stars; WHILE s # NIL DO m.WriteClosure(R, s); s := s.next END ;
   m.WriteClosure(R, NIL);
   c := G.cons; WHILE c # NIL DO m.WriteClosure(R, c); c := c.next END ;
   m.WriteClosure(R, NIL);
END WriteGraph;
```

```
PROCEDURE WriteCons (VAR m: Files2.Map; VAR R: Files.Rider; o: REFANY);
  VAR c: Constellation; i: INTEGER;
BEGIN c := SYSTEM.VAL(Constellation, o);
  Files.WriteNum(R, c.nofstars);
  FOR i := 0 TO c.nofstars – 1 DO m.WriteClosure(R, c.s[i]) END
END WriteCons;

PROCEDURE WriteStar (VAR m: Files2.Map; VAR R: Files.Rider; o: REFANY);
  VAR s: Star;
BEGIN s := SYSTEM.VAL(Star, o);
  Files.WriteNum(R, s.x); Files.WriteNum(R, s.y)
END WriteStar;

PROCEDURE ReadGraph (VAR m: Files2.Map; VAR R: Files.Rider; o: REFANY); ...
PROCEDURE ReadCons (VAR m: Files2.Map; VAR R: Files.Rider; o: REFANY); ...
PROCEDURE ReadStar (VAR m: Files2.Map; VAR R: Files.Rider; o: REFANY); ...
```

The corresponding *Read*-procedures follow easily by inverting the *Write*-procedures. Recall that the introduced customization is completely transparent to subclasses of *Constellation* such as rectangles or lines.

Performance measurements have shown that the mechanism provides acceptable performance, which is for both reading and writing about a factor of two below a carefully hand-crafted implementation. This means that except for very large graphics the difference in speed is not observable by the user.

Get/SetAttributes

Changing object attributes such as line width, fill pattern or arrow kind can also benefit from meta-programming. We provide two commands *Get* and *Set* which allow to read and write the value of the specified attribute of the most recent graphics selection. Two examples are "Kepler.Get linewidth" or "Kepler.Set linewidth 8", which look for the attribute *linewidth* of the selected objects and, if present, display or set it. Of course, a class can also provide additional commands to handle attributes which cannot be handled generically. Note that in the procedure *Set* below the module name *GenericObjects* is again abbreviated as *GO*.

```
PROCEDURE Set*;
  VAR S: Texts.Scanner; R: GO.RecordRider; R2: GO.ArrayRider;
    attr, field: ARRAY 32 OF CHAR; vis: BOOLEAN;
BEGIN
  open scanner S; Texts.Scan(S);
  IF S.class = Texts.Name THEN
    COPY(S.s, attr); Texts.Scan(S);
    for each selected constellation c do
```

```
            GO.OpenRider(R, c); GO.SetLevel(R, 1);
            LOOP
               IF R.mode = GO.None THEN EXIT END ;
               R.GetLocation(field, vis);
               IF (field = attr) & (vis = GO.Exported) THEN EXIT END ;
               R.Skip
            END ;
            IF R.mode # None THEN
               CASE R.class OF
               | GO.Char: IF S.class = Texts.Char THEN R.WriteChar(S.ch) END
               | GO.Bool: IF S.class = Texts.Name THEN R.WriteBool(S.s = "TRUE") END
               | ...
               | GO.Array: GO.ZoomArray(R2, R);
                  IF R2.class = GO.Char THEN WriteString(R, S.s) END
               ELSE
               END
            END
         end foreach;
         update views
      END
   END Set;
```

6.5 Data Structure Visualization

Our next example is built on top of the graphics editor introduced in the previous section. It is a facility for visualization of dynamic data structures in a two dimensional way, i.e. as a graphics rather than as a text. For this purpose, it suffices to extend *Kepler* by two specialized classes, one for displaying heap objects and one for displaying connections between those objects. In the following, we shall outline the functionality of the introduced classes without going into details of the implementation.

The visualization facility is based on the observation that it is in general not possible to display an arbitrary data structure with an appropriate layout automatically. This would be as complex as solving the routing and placement problem in VLSI design tools and it would in most cases display a lot of data which the user is not interested in. Therefore, we provide a way for interactively dereferencing pointers and give the user control over placement of dereferenced objects. Starting from a root object, which might be taken from a trap viewer for instance, the user can point to a pointer field and drag the mouse to the place where the dereferenced object is to be displayed. Fig. 6.4 outlines this activity.

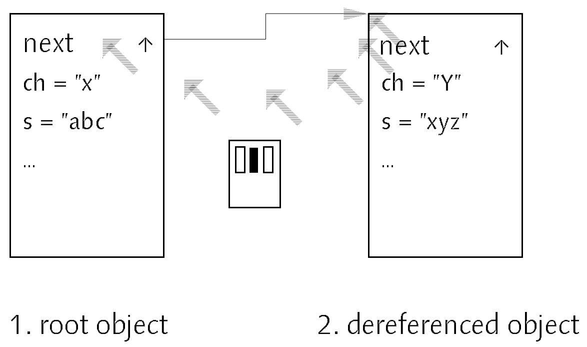


Fig. 6.4 – Dereferencing a pointer

ObjectView

Heap objects are visualized by instances of class *ObjectView*. The contents of the object is stored in a *text* which is produced by *GenericObjects.WriteObj*. When displayed, this text is surrounded by a rectangle. For handling of structured components, *level* specifies the number of expanded nesting levels. A reference to the underlying heap object is kept in field *o*. This field is necessary for detection of already displayed objects for instance in case of cyclic references. A visual feedback is given in such cases by highlighting the identified object instead of inserting a new one.

```
TYPE
   ObjectView = POINTER TO ObjectViewDesc;
   ObjectViewDesc = RECORD
      (KeplerFrames.ButtonDesc)
      text: Texts.Text;
      level: INTEGER;
      o: REFANY
   END;
```

Connection

References between heap objects are visualized by instances of class *Connection*. Connections are based on three handles, the top-left edge of the root object, an in-between handle and the top-left edge of the target object. This allows to display connections in an intuitive way for all relevant situations. A nice side-effect of the editing model used in Kepler is that if a view object is moved, all dependent connection objects are moved too. Connection objects have to know their vertical position in the root object for drawing themselves at the right place. This information is stored in field *yoff* as offset from the top boundary.

```
TYPE
  Connection = POINTER TO ConnectionDesc;
  ConnectionDesc = RECORD
    (KeplerGraphs.ConstellationDesc)
    yoff: INTEGER
  END ;
```

For convenience reasons, connections have a number of drawing heuristics built in. The routing of connections can be affected later on by moving handle 1. Drawing of connections always starts at coordinates (x0, y0 – yoff) and ends at coordinate (x2, y2). Connections are placed *behind* object views as shown in Fig. 6.5.

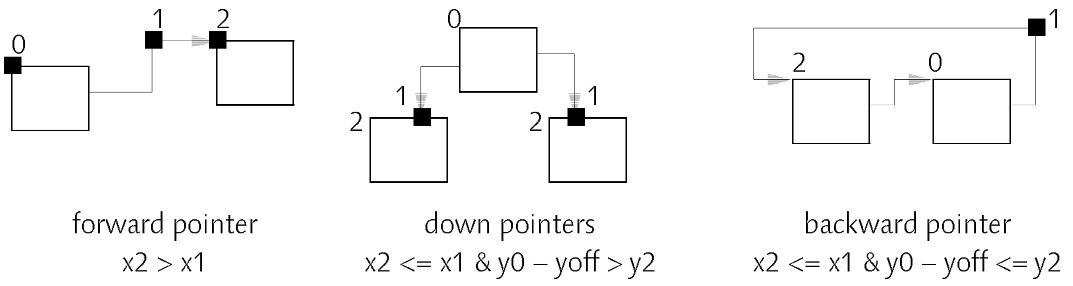


Fig. 6.5 – Display heuristics for Connections

It should be noted that it is also possible to provide for the automatic display of composite data structures such as sequential lists or binary trees. We implemented two commands *List* and *BinTree* for exactly this purpose. *List* takes a pointer field name as parameter and displays a sequential list rooted in the star marked object by placing the objects horizontally aligned. *BinTree* takes two parameters, the name of a left pointer field and the name of a right pointer field. It places the objects as a binary tree rooted in the star marked object such that no overlappings occur.

It is also possible to provide additional commands for application specific display of composite data. For the development of a compiler which is based on an abstract syntax tree, for example, it could be helpful to provide a command which displays (parts of) this tree. If application specific knowledge is available, the placement problem becomes very simple in most cases.

Compared with tools available in commercial programming systems (debuggers, browsers, inspectors), the presented approach is distinguished by its simplicity and extensibility. It has been implemented in a couple of days after the requirements had become clear. This shows that our facilities for generic access to data structures are useful for a number of practical examples and concludes the first part of this chapter. The second part deals with application of facilities for controlling procedure activations.

6.6 Command Interpretation

A first example for controlling procedure activations is a command interpreter which activates arbitrary procedures. This is a generalization of standard Oberon command interpretation, where only parameterless procedures can be activated. We use a simplified Oberon procedure-call-syntax for commands and allow literals and designators as arguments. Both the command name and the designators must be qualified with a module name (i.e. the designators refer to global variables). As a simplification, we use the dot-notation of record field selection for array indexing as well.

```
command = qualident ["↑" | param {param} ].
param = designator | literal.
designator = qualident { "." (name | number) }.
literal = intconst | realconst | stringconst ... .
qualident = name "." name.
```

Examples of commands are:

```
Out.Open
Out.Int 42 2
Out.Int Oberon.curCol 3
Viewers.This 100 100
```

The implementation of the command interpreter consists of a scanner and a simple parser. The built-in Oberon text scanner has been used for scanning and the parser essentially consists of two procedures *Params* and *Desig*. In the following code fragments, the module names *GenericObjects* and *ActiveProcedures* have been abbreviated to *GO* nad *AP* respectively.

```
VAR res: INTEGER; S: Texts.Scanner; W: Texts.Writer;

PROCEDURE Exec;
   VAR mod, proc: ARRAY 64 OF CHAR; p: Libraries.Object;
      m: Modules.Module; ref: LONGINT; function: BOOLEAN;
      msg: AP.InvocationMsg; R: AP.ParamRider;
BEGIN open scanner S; Texts.Scan(S); res := 0;
   IF S.class = Texts.Name THEN
      Split(S.s, mod, proc); m := Modules.ThisMod(mod);
      IF m # NIL THEN
```

```
          m.GetRef(proc, ref); m.GetObj(ref, p);
          IF (p # NIL) & (p IS Modules.Procedure) THEN
             AP.GetParams(p(Modules.Procedure), msg.par);
             Texts.Scan(S); Params(msg.par, function);
             IF res = 0 THEN
                IF p.handle = NIL THEN AP.Eval(p(Modules.Procedure), msg.par)
                ELSE p.handle(p, msg)
                END ;
                IF function THEN AP.OpenRider(R, msg.par); GO.WriteItem(W, R)
                ELSE RETURN
                END
             ...
```

The top level procedure *Exec* sets up a text scanner, checks if the first token is a name and splits this name into its two components. Then it tries to get the corresponding procedure object, creates the parameter block, and parses the actual parameters with a one symbol lookahead technique. Eventually, it calls *Eval* to evaluate the procedure with the given parameters and displays a possible function result value in the log viewer.

```
   PROCEDURE Params(VAR par: AP.Parameters; VAR f: BOOLEAN);
      VAR R: AP.ParamRider; name: ARRAY 32 OF CHAR; vis: SHORTINT;
   BEGIN f := FALSE; AP.OpenRider(R, par);
      IF R.mode # GO.None THEN R.GetLocation(name, vis);
         IF name = "$" THEN f := TRUE; R.Skip END ;
         WHILE (R.mode # GO.None) & (res = 0) DO
            IF R.mode = GO.VarPar THEN Desig(R)
            ELSE CASE R.class OF
               | GO.Char:
                     IF S.class = Texts.Char THEN R.WriteChar(S.c); Texts.Scan(S)
                     ELIF S.class = Texts.String THEN R.WriteChar(S.s[0]); Texts.Scan(S)
                     ELSE Desig(R)
                     END
               | ...
```

The task of the parser is not only to check whether the actual parameters are well formed according to the grammar but also that they match with the structure of the formal parameters. Therefore, the parser is controlled by the formal parameter list which is represented by the parameter rider *R*. The parser is actually not a simple recursive decent parser but table driven, where the table is encoded in the parameter record. This mechanism can be seen easily in the outermost loop, which terminates if the end of the formal parameter list has been reached. It can also be seen in the handling of VAR-parameters, which require a designator as actual parameter, whereas for value parameters both literals and designators are allowed. As a side-task, procedure *Params* checks if the called procedure is a function by comparing the first element of the parameter record with the special name "$", which serves as the name of function result values.

```
PROCEDURE Desig(VAR R: GO.Rider);
   VAR mod, var, rest: ARRAY 64 OF CHAR;
      m: Modules.Module; R0: GO.RecordRider;
BEGIN
   IF S.class = Texts.Name THEN
      Split(S.s, mod, rest);
      m := Modules.ThisMod(mod);
      IF m # NIL THEN GO.OpenRider(R0, m.data);
         Split(rest, var, rest); set(R0, var); Sel(R, R0, rest);
      ELSE res := moduleNotFound
      END ;
      Texts.Scan(S)
   ELSE res := identExpected
   END
END Desig;
```

For the sake of simplicity, we assume that designators are written without white space in between, thus, they can be treated as a single scanner token. This token has to be split up into its components. Every component means either a field selection or an array indexing. Pointers are assumed to be dereferenced implicitly (as it is done in the Oberon language). Since the rider which holds the resulting actual parameter (RecordRider or ArrayRider) is not known in advance, we have to use a recursive selection mechanism and pass the actual parameter at the end of the recursion as it can be seen in procedure *Sel* below. An auxiliary procedure *Set* is assumed, which sets the rider position to the location with the specified name.

```
PROCEDURE Sel(VAR R, R0: GO.Rider; VAR rest: ARRAY OF CHAR);
   VAR var: ARRAY 32 OF CHAR; p: REFANY;
      R1: GO.RecordRider; R2: GO.ArrayRider;
BEGIN
   IF res = 0 THEN
      IF rest # "" THEN
         Split(rest, var, rest);
         IF R0.class = GO.Record THEN
            GO.ZoomRecord(R1, R0); Set(R1, var); Sel(R, R1, rest)
         ELSIF R0.class IN {GO.Array, GO.DynArr} THEN
            GO.ZoomArray(R2, R0); Set(R2, var); Sel(R, R2, rest)
         ELSIF R0.class = GO.Pointer THEN
            R0.ReadPtr(p); GO.OpenRider(R1, p); Set(R1, var); Sel(R, R1, rest)
         ELSE res := structuredTypeExpected
```

```
            END
          ELSE R.Pass(R0, res)
          END
        END
      END Sel;
```

A tool with the functionality of our generalized command interpreter is sometimes called a test frame generator (not to be confused with a test case generator). It enables the programmer to invoke (test) individual procedures without having to build an environment for providing the parameters and for displaying the results. Test frame generators are usually applied to a module definition and generate a program which acts as a test bed for the specified module. Compared to this sort of tool, our command interpreter is much easier to implement (more than a factor of 10 shorter if compared to a test frame generator that has been implemented by the author but never released) and at least as convenient to use. There are no temporary or hidden files involved, there is no need for a compiler and last but not least, it is a general purpose tool which covers the functionality of a test frame generator more or less by accident. The main task is the execution of possibly parameterized commands, which in Oberon serve as the atomic entities of interaction between the user and the computer.

6.7 A Tracing Utility

The next example shows the usage of procedure handlers. We implement a simple facility which allows to trace procedure calls by installing a handler in the corresponding procedure object. After every call of a traced procedure, its parameter record is appended in textual form to the log text. Value parameters are printed as they appear before the call and VAR-parameters and function results as they appear after the call. A pitfall is the recursive activation of the tracing facility while generating the trace output. To avoid tracing the tracer, a global *tracing* flag is used, which is initially set to FALSE.

```
PROCEDURE Tracer(p: Libraries.Object; VAR M: Libraries.ObjectMsg);
  VAR name: ARRAY 24 OF CHAR; R: AP.ParamRider;
BEGIN
  WITH M: AP.InvocationMsg DO
    AP.Eval(p(Modules.Procedure), M.par);
    IF ~tracing THEN tracing := TRUE;
      p.lib.GetName(p.ref, name);
      Texts.WriteString(W, "tracing "); Texts.WriteString(W, p.lib.name);
      Texts.Write(W, "."); Texts.WriteString(W, name); Texts.WriteLn(W);
```

```
            AP.OpenRider(R, M.par); GO.WriteObj(W, R, 0, 1);
            Texts.Append(Oberon.Log, W.buf);
            tracing := FALSE
        END
    ELSE
    END
END Tracer;
```

The tracing facility can be enabled on a per procedure basis, by an auxiliary command which callls AP.InstallHandle for a given procedure. For example, tracing Oberon's display broadcast mechanism would be enabled by a call of

```
AP.InstallHandle(This("Viewers.Broadcast"), Tracer)
```

where an auxiliary procedure *This* is assumed to map qualified identifiers to procedure pointers.

6.8 Notification

As an example for filter procedures, we assume the following situation. A graphical user interface tries to display all files as icons on a desktop. Whenever a change in the file directory is made, the desktop should be kept consistent. If the *Files* module does not know about desktops and does not provide any means for notification, there is no traditional way but to periodically check the directory for changes. This is obviously unsatisfying and can be solved easily by using special filters for the procedures which manipulate the file directory. In principle, this could also be done by using a procedure handler, but in this case, we know exactly which procedures we want to monitor. We shall look at the implementation of filters for two examples, deleting and renaming of files. The filters are essentially notifiers (they notify objects about events which they might be interested in) and are therefore called *NotifyDelete* and *NotifyRename* respectively. In the standard Oberon *Files* module, delete and rename operations are specified with the following signatures:

```
PROCEDURE Delete (name: ARRAY OF CHAR; VAR res: INTEGER);
PROCEDURE Rename (old, new: ARRAY OF CHAR; VAR res: INTEGER);
```

The notifiers are declared in a different module. In our example, they would most probably be implemented in one of the modules that implement the graphical desktop. In the same module, a new message type is introduced

(FileDirMsg). Messages of this type can be broadcast to the display space, where a desktop viewer can respond to them appropriately.

```
TYPE
   FileDirMsg = RECORD
      (Display.FrameMsg)
      id: INTEGER;
      name, new: ARRAY 32 OF CHAR;
   END ;

PROCEDURE NotifyDelete (name: ARRAY OF CHAR; VAR res: INTEGER);
   VAR M: FileDirMsg;
BEGIN
   Files.Delete (name, res);
   IF res = 0 THEN
      COPY(name, M.name); M.id := del; Viewers.Broadcast(M)
   END
END NotifyDelete;

PROCEDURE NotifyRename (old, new: ARRAY OF CHAR; VAR res: INTEGER);
   VAR M: FileDirMsg;
BEGIN
   Files.Rename (old, new, res);
   IF res = 0 THEN
      COPY(old, M.name); COPY(new, M.new); M.id := ren; Display.Broadcast(M)
   END
END NotifyRename;
```

The filters must be installed by *Push* operations which are typically performed in a module's body.

```
AP.Push(This("Files.Delete"), This("module.NotifyDelete"));
AP.Push(This("(Files.Rename"), This("module.NotifyRename"));
```

7 Summary and Conclusions

This last chapter summarizes what has been achieved, outlines areas for future research and tries to draw the conclusions of this project with respect to the programming language Oberon in particular and with respect to the meta-programming approach in general.

7.1 Summary

We have introduced a generalized notion of persistent objects and object libraries which also covers components of programs such as types, procedures and modules. In our approach, a module is considered to be a persistent collection of type and procedure objects, thus it is a special kind of object library. Since there may be a potentially unlimited number of different library kinds, we have introduced an extensible mechanism for library handling which includes dynamic library loading and unloading. Module loading in this framework is just a special instance of a more general task. We have applied this framework to the design of a meta-level architecture which introduces types, procedures and modules as first-class objects.

Based on this architecture, a metaprogramming protocol has been defined which consists of two main parts. The first part provides generic access to arbitrary data structures by introducing the concept of hierarchical riders, which can be used to iterate over objects allocated on the heap, global data of modules, procedure activation records and parameter lists. Riders can be used to zoom into structured components of objects down to an arbitrary nesting level. The second part allows controlling procedure activations by means of reifying parameter lists and providing explicit access to the evaluation of procedures. By treating procedures as objects, which in general have behavior also, we have introduced the concept of active procedures. An active procedure has a message handler and can respond to messages such as invocation requests in its own, possibly application-dependent, way.

We have shown a simple and efficient implementation for safe library loading and unloading and we have discussed the implications with respect to garbage collection. A practical approach for the handling of large libraries and for solving the finalization problem has been introduced. We have also shown that an implementation of the introduced metaprogramming protocol is possible without introducing undue complexity or inefficiency.

Some of our results can be applied in isolation and have already found their way into standard Oberon implementations. Safe library loading, for instance, is now implemented in most ETH-Oberon systems. The finalization mechanism and the handling of subobjects could also be implemented independently of the metaprogramming facilities.

7.2 Future Work

At the time when our work began, the notion of metaprogramming was a rather exotic concept for the general-purpose programming community. Since then, the interest in this approach has increased significantly and new application areas have become apparent. Metaprogramming is now considered to be a promising approach for all kinds of interfacing tasks including interfacing to graphical user interface toolkits, interfacing to data base systems, and interfacing between multiple processes or computers with remote procedure calls or by sending objects across a network.

For the latter, our approach of unifying persistent object libraries and modules seems to be relevant. An object server could for instance be built that would allow accessing remote objects stored in a library. This library could have a number of imports including the modules needed to implement the exported objects. Recall that the type tag of an object and procedure variables are nothing but references to possibly imported types and procedure objects. Thus, a simple protocol could be defined that not only allows transferring an object's local and imported data, but also transferring the implementation of an object, i.e. its code. This transparent handling of behavior in addition to contents would of course also provide new challenges to the designers of such systems. The most obvious problem is expected to be security. The behavior of an object must be trusted, since an object can get control without explicit user interaction. Another problem is the machine dependence of today's compiled modules. This could be solved by using a portable module distribution format and delaying the machine-specific code generation until installation time or – more traditionally – by providing multiple object file versions on the server.

If the idea of object servers is put into a commercial context, the flat name space for libraries could become an obstacle. In fact, any vendor could pollute the library name space with names being used by other vendors for different purposes and thereby could cause incompatibilities. It will probably be necessary to introduce one additional level, which provides locality for libraries supplied by different vendors. Let us introduce the term *package* for such a group of libraries. Packages would export library names and import other packages. A package name would probably be subject to trademark, hence it would be world-wide unique. Packaging will probably not be introduced at the language level, but at the level of the library loading framework, i.e. at the system level.

A different topic is the application of metaprogramming facilities for implementing or interfacing to object-oriented data bases. It is expected that for supporting this kind of applications, dereferencing pointers must be under explicit programmer control. This can either be done by using the protection mechanism of memory management units and page fault handlers or – similar to our implementation of active procedures – by intercepting pointer access by special in-lined code. Since object-oriented data bases are a rather new research topic and it is not clear yet what an object-oriented data base really is, we have not further inquired into this topic.

7.3 Implications for Programming Language Design

During the design and implementation of our meta-level architecture for Oberon, some minor weaknesses of the language Oberon-2 have been detected. Some of the problems have to be seen in connection with the particular requirements of metaprogramming, others are more general. Most points refer to the type system, which is not surprising since the type system is at the very heart of a strongly typed programming language such as Oberon. The following discusses these points and outlines proposals for possible improvements. In order to avoid a wrong picture, we want to emphasize that our experience with the language Oberon-2 was very positive in general and that we do not suggest an immediate language update. The following points might be taken as ideas for discussion and in case other projects uncover similar problems might lead to extending the language slightly.

Untyped Pointers

We have used the type REFANY in some places in our metaprogramming protocol. This type is actually defined as *TYPE REFANY = SYSTEM.PTR*, which means that it is not expressed within proper Oberon-2 but escapes to pseudo module SYSTEM. This module is intended for doing system-level programming where it is necessary to circumvent type checking when operating directly on untyped memory. However, we wanted to use type REFANY not to operate on untyped but on arbitrary typed memory. What we want is fully dynamic type checking as opposed to type casting. The language Modula-3 [Nel91] introduced type REFANY for exactly this purpose and inspired the use of REFANY in this work. In Modula-3, any pointer variable can be assigned to variables of type REFANY, the reverse assignment needs a type guard, though. In addition, we would also like that arbitrary procedures and procedure variables can be assigned to REFANY. In principle it would be possible to live with SYSTEM.PTR if only type guards and type tests were allowed on variables of this type. In the portable Oberon-2 front-end this is currently not possible.

Abstract Methods

Oberon-2 does not provide any means for expressing the difference between an abstract (not implemented) and a concrete (implemented) method. Abstract methods have to be expressed as concrete methods with a HALT statement in their body. This is not a severe restriction but concerning the importance of abstract methods for implementors of subclasses an unpleasant shift from static to dynamic checking. Neither for the programmer nor for the compiler is it visible from the definition of a type whether a procedure bound to this type must be overridden or not. A possible solution for this problem may be to denote abstract methods within record types and concrete ones outside as it is done now. One advantage of this notation is that it would not invalidate any existing programs.

```
TYPE
  Library = POINTER TO LibraryDesc;
  LibraryDesc = RECORD
    ...
    PROCEDURE (L: Library) GetImport (n: INTEGER; VAR imp: Library);
  END;

  Module = POINTER TO ModuleDesc;
  ModuleDesc = RECORD (LibraryDesc)
    ...
  END;

  PROCEDURE (M: Module) GetImport (n: INTEGER; VAR imp: Library);
```

Another more pragmatic solution, which avoids any syntax change at all, would be to employ Oberon's forward-declaration mechanism and to regard unresolved forward methods as being abstract.

```
PROCEDURE ↑ (L: Library) GetImport (n: INTEGER; VAR imp: Library);
```

Comparison of Procedure Variables

Oberon does not allow to compare a procedure variable with a procedure constant. It is necessary to assign the procedure constant to an auxiliary procedure variable first and to compare the two variables. The restriction of using procedure constants only for assignments seems to be rather accidental and leads to the usage of unnecessary auxiliary variables.

7.4 Conclusions

The overall conclusion which can be drawn from this project is that metaprogramming in a statically typed and compiled programming language such as Oberon is indeed feasible and useful. With the only exception of dynamically typed pointers as mentioned above, the type system was never in the way of our goals but helped to structure and document the system. For the demanding tasks of future software systems, metaprogramming seems to be a promising technique which should be part of the standard repertoire of every software engineer. The presented applications show that programs using meta-level facilities are as simple as programs that don't. In fact, metaprogramming does not introduce meta-level problems but is just like normal programming but with other programs and data structures as its domain.

Appendix: Module Definitions

```
DEFINITION Libraries;

 CONST
  Done = 0; TypeNotFound = 1; LibNotFound = 2; InvalidLib = 3;
  InvalidKey = 4; OutOfMemory = 5; ClientsExist = 7;

 TYPE
  Name = ARRAY 20 OF CHAR;
  Library = POINTER TO LibraryDesc;
  Object = POINTER TO ObjectDesc;
  LibraryDesc = RECORD
   name–: Name;
   nofClients–, nofImports, nofObjects: LONGINT;
   init, fini: PROCEDURE (L: Library);
   PROCEDURE (L: Library) GetImport (n: LONGINT; VAR imp: Library);
   PROCEDURE (L: Library) GetName (ref: LONGINT; VAR name: ARRAY OF CHAR);
   PROCEDURE (L: Library) GetObj (ref: LONGINT; VAR o: Object);
   PROCEDURE (L: Library) GetRef (name: ARRAY OF CHAR; VAR ref: LONGINT);
   PROCEDURE (L: Library) GetVersion (ref: LONGINT; VAR key: LONGINT);
   PROCEDURE (L: Library) Unmark (all: BOOLEAN);
  END ;

  ObjectMsg = RECORD END ;
  Handler = PROCEDURE (O: Object; VAR M: ObjectMsg);
  ThisProc = PROCEDURE (name: ARRAY OF CHAR): Library;
  EnumProc = PROCEDURE (L: Library; VAR cont: BOOLEAN);

  ObjectDesc = RECORD
   lib: Library;
   ref: LONGINT;
   handle: Handler;
  END ;

 VAR
  imported, importing: Name;
  res: INTEGER;

 PROCEDURE This (name: ARRAY OF CHAR): Library;
 PROCEDURE Free (L: Library);
 PROCEDURE Install (ext: ARRAY OF CHAR; this: ThisProc);
 PROCEDURE Enumerate (do: EnumProc);
 PROCEDURE Lookup (name: ARRAY OF CHAR; VAR L: Library);

END Libraries.
```

```
DEFINITION Modules;

 IMPORT Libraries;

 TYPE
  Module = POINTER TO ModuleDesc;
  ModuleDesc = RECORD (Libraries.LibraryDesc)
   nofcoms–, nofrecs–, nofrefs–, nofActiveProcs: INTEGER;
   entries–, names–, SB–, code–, refs–: LONGINT;
   data–: Data;
   imp–: ARRAY 32 OF Module;
   PROCEDURE (M: Module) GetImport (n: LONGINT; VAR imp: Libraries.Library);
   PROCEDURE (M: Module) GetName (ref: LONGINT; VAR name: ARRAY OF CHAR);
   PROCEDURE (M: Module) GetObj (ref: LONGINT; VAR o: Libraries.Object);
   PROCEDURE (M: Module) GetRef (name: ARRAY OF CHAR; VAR ref: LONGINT);
   PROCEDURE (M: Module) GetVersion (ref: LONGINT; VAR key: LONGINT);
   PROCEDURE (M: Module) Unmark (all: BOOLEAN);
  END ;

  Procedure = POINTER TO ProcedureDesc;
  ProcedureDesc = RECORD (Libraries.ObjectDesc) END ;

  Type = POINTER TO TypeDesc;
  TypeDesc = RECORD (Libraries.ObjectDesc)
   info–: LONGINT;
   level–: INTEGER;
   base–: ARRAY 8 OF Type;
  END ;
  Command = PROCEDURE;

 PROCEDURE ThisCommand (mod: Module; name: ARRAY OF CHAR): Command;
 PROCEDURE ThisMod (name: ARRAY OF CHAR): Module;

END Modules.
```

```
DEFINITION GenericObjects;

 IMPORT SYSTEM, Modules, Texts;

 CONST
  (* type classes *)
  Byte = 1; Bool = 2; Char = 3; SInt = 4; Int = 5; LInt = 6;
  Real = 7; LReal = 8; Set = 9; Pointer = 13; Procedure = 14;
  Array = 15; Record = 16; DynArr = 17;

  (* object modes *)
  None* = 0; Var* = 1; VarPar* = 2; Elem* = 3; Fld* = 4;

  (* visibility *)
  Private = 0; Exported = 1; ReadOnly = 2;

 TYPE
  REFANY = SYSTEM.PTR;
  Rider = RECORD
   mode–, class–: SHORTINT;
   PROCEDURE (VAR R: Rider) GetLocation (VAR n: ARRAY OF CHAR; VAR vis: SHORTINT);
   PROCEDURE (VAR R: Rider) Pass (VAR Rs: Rider; VAR res: INTEGER);
   PROCEDURE (VAR R: Rider) ReadBool (VAR x: BOOLEAN);
   PROCEDURE (VAR R: Rider) ReadChar (VAR x: CHAR);
   PROCEDURE (VAR R: Rider) ReadInt (VAR x: INTEGER);
   PROCEDURE (VAR R: Rider) ReadLInt (VAR x: LONGINT);
   PROCEDURE (VAR R: Rider) ReadLReal (VAR x: LONGREAL);
   PROCEDURE (VAR R: Rider) ReadProc (VAR x: REFANY);
   PROCEDURE (VAR R: Rider) ReadPtr (VAR x: REFANY);
   PROCEDURE (VAR R: Rider) ReadReal (VAR x: REAL);
   PROCEDURE (VAR R: Rider) ReadSInt (VAR x: SHORTINT);
   PROCEDURE (VAR R: Rider) ReadSet (VAR x: SET);
   PROCEDURE (VAR R: Rider) ReadString (VAR x: ARRAY OF CHAR);
   PROCEDURE (VAR R: Rider) Skip;
   PROCEDURE (VAR R: Rider) WriteBool (x: BOOLEAN);
   PROCEDURE (VAR R: Rider) WriteChar (x: CHAR);
   PROCEDURE (VAR R: Rider) WriteInt (x: INTEGER);
   PROCEDURE (VAR R: Rider) WriteLInt (x: LONGINT);
   PROCEDURE (VAR R: Rider) WriteLReal (x: LONGREAL);
   PROCEDURE (VAR R: Rider) WriteProc (x: REFANY);
   PROCEDURE (VAR R: Rider) WritePtr (x: REFANY);
   PROCEDURE (VAR R: Rider) WriteReal (x: REAL);
   PROCEDURE (VAR R: Rider) WriteSInt (x: SHORTINT);
   PROCEDURE (VAR R: Rider) WriteSet (x: SET);
   PROCEDURE (VAR R: Rider) WriteString (VAR x: ARRAY OF CHAR);
  END ;
  ActivationRider = RECORD (Rider)
   module–, proc–: ARRAY 64 OF CHAR;
   retpc–, relpc–, dlink–: LONGINT;
   PROCEDURE (VAR R:ActivationRider) GetLocation(VAR n:ARRAY OF CHAR; VAR vis:SHORTINT);
  END ;

  ArrayRider = RECORD (Rider)
   len–, index–: LONGINT;
   PROCEDURE (VAR R: ArrayRider) GetLocation (VAR n: ARRAY OF CHAR; VAR vis: SHORTINT);
   PROCEDURE (VAR R: ArrayRider) ReadProc (VAR x: REFANY);
   PROCEDURE (VAR R: ArrayRider) ReadPtr (VAR x: REFANY);
   PROCEDURE (VAR R: ArrayRider) ReadString (VAR x: ARRAY OF CHAR);
   PROCEDURE (VAR R: ArrayRider) Skip;
   PROCEDURE (VAR R: ArrayRider) WriteProc (x: REFANY);
   PROCEDURE (VAR R: ArrayRider) WritePtr (x: REFANY);
   PROCEDURE (VAR R: ArrayRider) WriteString (VAR x: ARRAY OF CHAR);
  END ;

  RecordRider = RECORD (Rider)
   level–: SHORTINT;
   PROCEDURE (VAR R: RecordRider) GetLocation (VAR n: ARRAY OF CHAR; VAR vis: SHORTINT);
  END ;

  RefElem = POINTER TO RefElemDesc;
  RefElemDesc = RECORD (Texts.ElemDesc)
   proc: BOOLEAN;
   p: REFANY;
  END ;

 PROCEDURE OpenRider (VAR R: RecordRider; o: REFANY);
 PROCEDURE OpenFrame (VAR R: ActivationRider; sp, pc: LONGINT);
 PROCEDURE SetLevel (VAR R: RecordRider; level: INTEGER);
 PROCEDURE SetIndex (VAR R: ArrayRider; idx: LONGINT);
 PROCEDURE ZoomRecord (VAR R: RecordRider; VAR base: Rider);
 PROCEDURE ZoomArray (VAR R: ArrayRider; VAR base: Rider);

 PROCEDURE WriteItem (VAR W: Texts.Writer; VAR R: Rider);
 PROCEDURE WriteObj (VAR W: Texts.Writer; VAR R: Rider; expand, indent: INTEGER);

 PROCEDURE Type (o: REFANY): Modules.Type;
 PROCEDURE New (VAR o: REFANY; t: Modules.Type);

 PROCEDURE AllocRef;
 PROCEDURE NewRef (proc: BOOLEAN; p: REFANY): RefElem;
 PROCEDURE HandleRef (E: Texts.Elem; VAR M: Texts.ElemMsg);

END GenericObjects.
```

```
DEFINITION ActiveProcedures;

  IMPORT Libraries, GenericObjects, Modules;

  TYPE
    Parameters = RECORD END ;

    InvocationMsg = RECORD (Libraries.ObjectMsg)
      par: Parameters;
    END ;

    ParamRider = RECORD (GenericObjects.Rider)
      PROCEDURE (VAR R: ParamRider) GetLocation (VAR n: ARRAY OF CHAR; VAR vis: SHORTINT);
      PROCEDURE (VAR Rd: ParamRider) Pass (VAR Rs: GenericObjects.Rider; VAR res: INTEGER);
    END ;

  PROCEDURE GetParams (p: Modules.Procedure; VAR par: Parameters);
  PROCEDURE OpenRider (VAR R: ParamRider; VAR par: Parameters);
  PROCEDURE Eval (P: Modules.Procedure; VAR par: Parameters);
  PROCEDURE SetProcHandle (p: Modules.Procedure; handle: Libraries.Handler);
  PROCEDURE FilterHandle (P: Libraries.Object; VAR M: Libraries.ObjectMsg);
  PROCEDURE Push (P, F: Modules.Procedure);
  PROCEDURE Pop (P: Modules.Procedure): Modules.Procedure;

END ActiveProcedures.
```

Bibliography

[ABBFG89] G. Attardi, C. Bonini, M. R. Boscotrecase, T. Flagella, M. Gaspari, "Metalevel Programming in CLOS". ECOOP'89 Proceedings, Cambridge University Press, 1989.

[AS85] H. Abelson, G. J. Sussman, "Structure and Interpretation of Computer Programs". The MIT Press, Cambridge, Massachusetts, 1985

[BCFT92] M. Brandis, R. Crelier, M. Franz, J. Templ, "The Oberon System Family". Computersysteme ETH Zürich, Report No. 174, April 1992.

[BKPS92] F. Buschmann, K. Kiefer, F. Paulisch, M. Stal, "The Meta-Information-Protocol: Run-Time Type Information for C++". Proceedings of the IMSA '92 Workshop, Tokyo, Japan.

[Bra61] H. Bratman, "An Alternate Form of the UNCOL Diagram". Comm. ACM 4, 1961, page 142.

[Cha93] C. Chambers, "The Cecil Language, Specification and Rationale". Department of Computer Science and Engineering, FR-35 University of Washington, Seattle, Washington 98195 USA Technical Report 93-03-05, March 1993

[Coh91] N. H. Cohen. "Type-Extension Type Tests Can Be Performed In Constant Time". ACM Trans. Program. Lang. Syst. 13, 4, (October 1991), 626–629.

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

[Cre94] R. Crelier, "Separate Compilation and Module Extension". forthcoming PhD Dissertation. ETH Zurich 1994.

[Ebe87] H. Eberle, "Development and Analysis of a Workstation Computer". ETH Zürich, Dissertation No. 8431 (1987).

[Fra93] M. Franz, "Emulating an Operating System on Top of Another". Software–Practice and Experience, Vol. 23(6), 677–692 (June 1993).

[Fra94] M. Franz, "Code-Generation On-The-Fly: A Key to Portable Software". PhD Dissertation. ETH Zurich 1994.

[GPHT91] R. Griesemer, C. Pfister, B. Heeb, J. Templ, "Oberon Technical Notes". Computersysteme ETH Zürich, Report No. 156, March 1991.

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

[Gut85] J. Gutknecht, "Compilation of Data Structures: A New Approach to Efficient Modula-2 Symbol Files". Computersysteme ETH Zürich, Report No. 64, July 1985.

[Gut93] J. Gutknecht, "Oberon System 3: Vision of a Future Software Technolo Software Concepts & Tools, Vol. 15(1), 1994.

[HDKP92] H. Bretthauer, H. Davis, J. Kopp, K. Playford, "Balancing the EuLisp Metaobject Protocol". Proceedings of the IMSA '92 Workshop, Tokyo, Japan.

[Kic92] G. Kiczales, "Towards a New Model of Abstraction in Software Engineering". Proceedings of the IMSA '92 Workshop, Tokyo, Japan.

[KRB91] G. Kiczales, J. des Rivieres, D. Bobrow, "The Art of the Metaobject Protocol". The MIT Press, Cambridge, Massachusetts, 1991

[Maes87] P. Maes, "Computational Reflection", PhD Thesis, Vrije Universiteit Brussel 1987 also published as technical report 87-2, AI-Lab Vrije Universiteit Brussel.

[McCar60] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine". Comm. ACM 3(4): 184–195, 1960.

[Hau93] M. Hausner, "Documentation of Text Elements". Part of SPARC-Oberon distribution (file Elem.Guide.Text), internet ftp: neptune.inf.ethz.ch:~ftp/Oberon/SPARC.

[Min74] M. Minsky, "A Framework for Representing Knowledge". MIT, AI–Lab, AI–MEMO 306. Cambridge, Massachusetts, 1974.

[MMN92] O. L. Madsen, B. Moller-Pedersen, K. Nygard,
"Object-oriented programming in the BETA programming language".
Draft, August 1992

[Moe93] H. Mössenböck, "Object-oriented Programming in Oberon-2".
Springer Verlag, 1993.

[Nel91] G. Nelson, Ed. "Systems Programming with Modula-3".
Prentice Hall, 1991

[Neu45] J. von Neumann, "First Draft of a Report on the EDVAC".
Papers of John von Neumann on computing and computer
theory. The MIT Press, Cambridge, Massachusetts, 1987

[PA90] A. Paepcke, "PCLOS: Stress Testing CLOS".
In ECOOP/OOPSLA '90 Proceedings,
ACM SIGPLAN Notices, 25 (10), Oct 1990

[RC93] J. E. Richardson, M. J. Carey, D.T. Schuh,
"The Design of the E Programing Language".
ACM Trans. On Prog. Lang. and Syst. 15 (3), July 1993.

[Rei91] M. Reiser,
"The Oberon System – User Guide and Programmer's Reference".
Addison–Wesley, 1991.

[RS84] J. des Rivieris, B. C. Smith, "Interim 3-LISP Reference Manual".
XEROX Intelligent Systems Laboratory ISL-1. Xerox Parc,
Palo Alto, California, 1984.

[RW92] M. Reiser, N. Wirth, "Programming in Oberon".
Addison–Wesley, 1992.

[Sim77] C. Simonyi, Meta Programming: A Software Method
PHD thesis, Stanford University, 1977.

[SC92] L. Smarr, C. E. Catlett, "Metacomputing",
Communications of the ACM, Vol. 35, No. 6, June 1992

[Smi82] B. C. Smith, "Reflection and Semantics in a Procedural Language",
PhD Thesis 1982, MIT Laboratory for Computer Science,
also published as MIT technical report 272, 1982.

[Ste90] G. L. Steele Jr., "Common Lisp, The Language".
Digital Press, Bedford, Massachusetts, 1990.

[Stein87]	L. A. Stein, "Delegation is Inheritance". In OOPSLA '87 Conference Proceedings, Orlando, FL, October 1987, Published in ACM SIGPLAN Notices 22(12), December 1987.
[Str91]	B. Stroustroup, "The C++ Programming Language". 2nd Edition, 1991.
[Szy92]	C. A. Szyperski, "Insight ETHOS: On Object-Orientation in Operating Systems". ETH Zürich, Dissertation No. 9884 (1992).
[Te91]	J. Templ, "Design and Implementation of SPARC–Oberon". Structured Programming, (1991) 12: 197–205, Springer Verlag, New York.
[Te93]	J. Templ, "Kepler". Part of SPARC-Oberon distribution (file Kepler.Text), internet ftp: neptune.inf.ethz.ch:~ftp/Oberon/SPARC.
[UCCH91]	D. Ungar, C. Chambers, B. Chang, U. Hölzle, "Organizing Programs Without Classes". Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, June, 1991.
[Ung84]	D. Ungar, "Generation Scavenging: A Non-disruptive, High-performance Storage Reclamation Algorithm". Proceedings of the ACM Symposium on Practical Software Development Environments, Pittsburg, PA. Published in ACM SIGPLAN Notices, 19, 5, 157–167 (April 1984).
[US87]	D. Ungar, R. B. Smith. "SELF: The Power of Simplicity". In OOPSLA '87 Conference Proceedings, Orlando, FL, October 1987, Published in ACM SIGPLAN Notices 22, 12, (December 1987).
[WG92]	N. Wirth, J. Gutknecht, "Project Oberon, the Design of an Operating System and Compiler". Addison–Wesley, 1992.
[Wil92]	P. R. Wilson. "Uniprocessor Garbage Collection Techniques". Proceedings of the 1992 International Workshop on Memory Management, St. Malo, Lecture Notes in Computer Science. Springer-Verlag, Sept. 1992.