Diss. ETH No 10650

Separate Compilation and Module Extension

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Régis Bernard Joseph Crelier
ing. informaticien dipl. EPFZ
born January 10, 1965
citizen of Bure, Jura

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

1994

Separate Compilation and Module Extension

Diss. ETH No 10650

Separate Compilation and Module Extension

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Régis Bernard Joseph Crelier
ing. informaticien dipl. EPFZ
born January 10, 1965
citizen of Bure, Jura

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

1994

A mes parents

Contents

Acknowledgements xi
Résumé xii
Abstract xiii

Chapter 1 **Introduction**
Declarations and Statements 2
Modules and Interfaces 3
Separate Compilation and Interface Checking 4
Program Linking and Loading 8

Chapter 2 **Symbol Files**
Symbol Table 11
Type Graph 11
Scope Graph 14
A Brief History 16
Symbol File Classification 16
Symbol File Linearization 17

Chapter 3 **System Consistency and Client Invalidation**
Consistency Checking 21
Granularity of the Checks 22
Compilation Dependence Graph 23
Using Keys for Consistency Checks 25
Restoring System Consistency 27
Trickle-Down Recompilations 27
Benefits of Tools 28
Background 30
Checksum Versus Timestamp 30
Enhancing the Granularity 31
Integrating Type Information into Object Files 32
Motivation 34

Chapter 4 **The Portable Oberon-2 Compiler OP2**
Architecture of the Compiler 37
Module Structure 38
Symbol Table 40
Syntax Tree 42
Compilation Phases 45
Lexical Analysis and Syntax Analysis 46
Storage Allocation 47
Code Generation 48

Chapter 5 **The Layer Model**
The Idea 51
A Stack of Extension Layers 52
Extensions Versus Modifications 53
Consistency Checking 55
The Implementation 58
Symbol File Format 58
Externalization, Fingerprinting, and Internalization 64
Storage Allocation and Reallocation 69
Front-End Modifications 76
Object File Format and Linker Modifications 76
Drawbacks and Limitations of the Model 77
Record Field Revelation 77
History of Development 78

Chapter 6 **The Object Model**
The Idea 81
A Fingerprint per Object 82
Fingerprinting Recursive Types 86
Breaking Cycles 89
Fingerprinting Signatures 93
Fingerprint Computation 95
Anonymous Types and Name Equivalence 98
A Fingerprint per Object Component 100
The Implementation 103
Symbol File Format 103
Fingerprinting Structures 106
Fingerprinting Objects 113
Consistency Checking at Compile Time 114
Comparing with the Old Symbol File 117

Object File Format 123
Linker and Run-Time Data Structures 127

Chapter 7 **Efficiency Considerations and Conclusions**
Implementation Costs and Measurements 131
Outlook 138
Automating Recompilations 138
Inserting New Type-Bound Procedures 139
Multiple Interfaces 140
Applying the Models to Other Languages 141
Conclusions 142

Appendix A: Layer Model File Formats 143
Appendix B: Object Model File Formats 146
References 149

Acknowledgements

Professor Niklaus Wirth, supervisor of this thesis, introduced me to the fascinating world of compiler construction. I learned more by studying both his Modula-2 and Oberon compilers than by reading any book on programming. By offering me the opportunity to work in his research group, he provided me with a sharper vision of what computer science really is, and should be.

Professor Hanspeter Mössenböck accepted promptly the coreference of this thesis. His numerous questions, comments, and valuable suggestions greatly helped to improve the comprehensibility of the text.

Joseph Templ was always ready to discuss or propose solutions to encountered problems. In particular, he had the idea to dissociate the fingerprint of a pointer type from the fingerprint of its base type.

Clemens Szyperski suggested that client modules not using the size of an imported record type should not be invalidated when this record is augmented by new hidden fields.

Jacques Supcik adopted the Object Model for his project of implementing the Oberon System on HP PA-RISC workstations. Thereby, he helped to test the model which was still in an early phase of its development.

Beat Heeb also applied the Object Model in a commercial Oberon programming environment based on the OP2 compiler. He found one bug.

Cuno Pfister and Stefan Ludwig carefully proofread parts of the manuscript and corrected many language mistakes and inelegances.

Hans Meier willingly commented on the page layout of this booklet. His expertise in typography is an inexhaustible source of wise counsel.

I express my deep gratitude to all the persons cited above for their guidance, advice, or contribution. I am also thankful to all my colleagues of the Institute for Computer Systems for the enjoyable working atmosphere and for the fruitful and exciting discussions during coffee breaks.

The implementation on DECstation has been supported by Digital Equipment's Systems Research Center in Palo Alto, California, USA, by their generous donation of a workstation.

Résumé

L'évolution incessante du matériel informatique permet la construction d'ordinateurs toujours plus performants. Parallèlement, de nouveaux concepts et de nouvelles techniques de programmation deviennent nécessaires afin de maîtriser la complexité grandissante des logiciels. L'une de ces techniques, la compilation séparée de modules, a fait ses preuves en Modula-2 et en Obéron, entre autres langages de programmation fortement typés.

Le module, qui est à la fois l'unité structurelle et l'unité de compilation des programmes, est interchangeable sans affecter les autres modules d'un système. Toutefois, si l'interface du nouveau module est différente de l'ancienne, les modules clients de cette interface modifiée doivent être recompilés afin de maintenir la cohérence du système. L'éditeur de liens vérifie habituellement que la clef réelle de chaque interface importée est identique à la clef attendue par le client, une différence indiquant une incohérence. Ce modèle simple et efficace n'est pas très souple, puisqu'une légère modification d'interface, telle l'insertion d'une nouvelle procédure, par exemple, peut provoquer de nombreuses recompilations inutiles.

Cette thèse présente deux nouveaux modèles de vérification fine de cohérence, ainsi que leur mise en oeuvre. Ces modèles permettent l'extension d'interfaces de modules compilés séparément sans qu'une recompilation des clients ne soit nécessaire, ce qui est particulièrement utile dans des systèmes avec chargement dynamique, où les clients d'une bibliothèque de modules ne sont pas forcément connus lors de la révision ou de l'extension de cette bibliothèque. L'édition d'interface sans conséquence ne se limite pas à des extensions, puisque la modification d'un article existant n'invalide pas les clients n'utilisant pas cet article. Et même s'ils l'utilisent d'une manière encore compatible après modification, une recompilation n'est pas plus indispensable.

Ces techniques ont été implantées dans le Système Obéron, mais elles ne sont spécifiques ni au Langage, ni au Système Obéron, et pourraient être appliquées à n'importe quel système de programmation modulaire pour en améliorer la sécurité, la flexibilité, ou tous les deux à la fois. De plus, ces techniques ne se présentent pas au programmeur sous forme d'outils dont l'utilisation reste facultative, mais elles sont entièrement intégrées au compilateur ainsi qu'au chargeur de modules. La sécurité ne doit pas être offerte en option.

Abstract

As continuous evolution in hardware results in more powerful computers, new programming techniques and concepts must be developed to master the consequently increasing software complexity. Separate compilation of modules is such a technique that has proven valuable in Modula-2 and in Oberon, among other strongly-typed programming languages.

The module is both the structural unit and the compilation unit of programs. Replacing a module by a new one does not affect the rest of the system, provided that the module interface has not changed. Otherwise, client modules of the modified interface have to be recompiled to maintain system consistency. The last opportunity to detect an inconsistency is when modules are linked to form an executable unit. The check usually consists in comparing, for each imported interface, the expected key of that interface, as known at compilation time of the client, with the key of the effectively supplied interface. A mismatch indicates an inconsistency. This model is simple and efficient, but not very flexible. Indeed, a minor modification of an interface, such as the insertion of a new procedure, can trigger many unnecessary recompilations.

This thesis presents two new models for fine-grained consistency checking and their implementation. These models allow the interface of separately compiled modules to be extended without requiring a recompilation of client modules. This is particularly valuable in systems with dynamic loading, where the clients of a library are not known when the library is revised or extended. Interface editing that does not require client recompilation is not restricted to extensions, since the modification of an existing item does not invalidate clients not using this particular item. Even if they use it in a way that is upward-compatible with the modification, they still do not need a recompilation.

These techniques have been implemented in the Oberon System, but they are neither specific to the Oberon Language nor to the Oberon System. They can be applied to any modular programming system in order to improve its safety, its flexibility, or both. Furthermore, these techniques are not available to the programmer as a separate tool whose use remains optional, but have been fully and transparently integrated into the compiler and module loader. Safety must not be optional.

Chapter 1

Introduction

As long as computers have existed, programming techniques have evolved. The first machines were programmed by setting a few dozen switches in adequate positions or by plugging some cables into the right connectors. Nowadays, software systems consist of thousands or even millions of bytes of code. Of course, the performance attained by old and current computers, as well as the kind of problems solved by them, cannot be compared.

In the computer pioneers' time, the whole difficulty resided in constructing the machine. Its programming was almost a trivial task that did not need special attention. The storage capacity was so low that only very simple algorithms and small amounts of data could fit into the program and data stores. The evolution of hardware made it necessary to develop software techniques. Since then, computer programming has become a science of its own.

Today, so much effort is invested in programming that software packages often survive several hardware generations. This does not mean that computer construction has become a trivial task. Computer architecture and computer programming are now largely two independent sciences evolving almost separately and are not intertwined as in the past. One does not need to know how a computer is built to be able to program it. Programming languages have created a level of abstraction between software and the underlying hardware. There is no longer a one-to-one correspondence between program statements and machine instructions.

Constant improvements in data storage capacity and processing speed make it possible to solve more and more complex problems. Consequently, more complex algorithms and data structures are needed. The role of the software engineer is to master this increasing complexity. He can model a problem by a simpler one, with a very similar behavior. He can also break the problem into several smaller problems easier to solve, and then put the results together ("divide et impera" technique). In practice, the structure of the program reflects this stepwise refinement at several levels: the program consists of *modules*, modules of *procedures*, and procedures of *statements*.

Declarations and Statements

A module defines and refers to different kinds of objects: *constants*, *variables*, *types* and *procedures*. A constant is a named value that cannot be changed during the execution of the module. A variable is a named container that can hold one value at a time. However, different values may be *assigned to* a variable during the same execution. A type is associated with every constant or variable. The type denotes the kind of value an object can have. It restricts the manipulation of the object to some applicable operations specific to that type. There are predefined types like *integer*, *character*, or *boolean*, and user-defined types like *pointers*, *arrays*, or *records*. Procedures describe sequences of operations applied to objects. They may take expressions of constants and variables as input parameters and yield result values. The operations, also called *statements*, are executed when the procedure is *called*. A procedure may also be assigned as a value to a variable. It is executed when the variable is referenced.

Before an object can be used, it has to be *declared*. The declaration provides the object with a name for subsequent references to the object, and with a type, which defines the nature of the object and its applicable manipulations. A local declaration appears in the block of a procedure, opening the *scope* of the declared object. The scope stretches from the point of the declaration to the end of the procedure, and corresponds to the domain where the object is valid and visible, and therefore accessible by statements or further declarations. A scope also extends to locally (internally) declared procedures. However, it may be masked by a declaration of an equally named object, in which case subsequent appearances of the name always refer to the object of the innermost scope.

Scopes extend to the inside, but not to the outside: an object declared in a local procedure cannot be accessed from within the enclosing procedure. Objects local to a procedure are instantiated when the procedure is entered and removed when the procedure returns. If they have to survive across several calls of the procedure, they must be declared at least in the same block where the procedure is declared or in some enclosing block.

Declarations appearing in the module block are *global*, in contrast to *local* ones found in procedures. Sometimes, the term *scope* is employed with a slightly different meaning: the *local scope* of a procedure designates the visibility range of local declarations in the procedure block. Global declarations follow the same scope rules as local declarations, and allow global objects to be visible in every procedure of the module and in the module body itself, but not outside of the module.

The *interface* of the module actually breaks these rules in a controlled way, making selected global objects visible and accessible to partner modules.

Modules and Interfaces

The concept of *module* or *package* is present in many modern programming languages like Ada [1], Mesa [2], Modula-2 [3], Modula-3 [4], and Oberon [5, 6]. The module is not just a container for a collection of objects; it has several purposes.

The enlarged picture of a car will not be very useful to understand its functioning. A detailed list of the parts the car is made of will not help either. It would be more appropriate to study the subsystems of the car separately, like the chassis, the engine, the electric system, and so on. It is then important to understand how these systems relate to each other. Their interactions define the global behavior of the car.

Similarly, software systems are sometimes very difficult to understand, and hence to implement, because they consist of many interacting components. The programmer has to identify these components and to specify the interactions between them by defining an *interface* for each of them. The module is then the *implementation* of the component's interface and accordingly the *structural unit* of the stepwise refinement process.

Interfaces play a central role in the design of a software system. They represent an abstraction by giving a simplified view of the module to the outside. Implementation details are not relevant to the clients of the interface. The driver does not need to know how a car engine is built in order to press on the accelerator. The module and its interface implement the concept of *data abstraction* and *code abstraction*.

It is sometimes desirable to explicitly hide information in order to protect this information. An ill-intentioned or ignorant client might invalidate program invariants by directly manipulating visible data structures. It is preferable to restrict access to sensitive information through a set of functions that can maintain invariants by rejecting unauthorized operations. Similarly, a security mechanism usually prevents the driver from engaging the reverse gear while the car is still moving, which may damage the gearbox otherwise. Modules assure data integrity and protection using *data encapsulation* and *information hiding*.

Interfaces also make it possible for a team to work on the same program. Each programmer is responsible for the implementation of one or several modules, the interfaces of which are defined at the beginning of the project. In this way, implementers can work independently of each other, using the

interface as a contract. Modules being in development can rely on interfaces of other modules not yet implemented. The car constructor does not have to wait for the chassis to be built before designing the engine. He just has to plan the exact position of the bolts and the nuts that will fix the engine onto the chassis. Therefore, interfaces serve as *protocol* between programmers and help to *coordinate* larger software projects.

One does not have to reinvent the wheel each time a new car is built. Different programs may use the same algorithms or the same look-and-feel features. Writing a user interface of an application often consists of calling the right routines from a graphical library in the right order. Libraries make their resources available to applications based on them, contrary to closed programs that can only be executed as they are, incapable of sharing parts of themselves with similar applications. Libraries are implemented as a collection of modules, where each module can be used by different clients. The module is the *unit of reusability*.

It is not necessary to buy a new car after a tire blows out. Changing the tire or the wheel usually solves the problem. One just has to be careful to replace it with a compatible model. Parts of software systems can also be changed without redesigning the whole system. Modules can be partially modified without affecting other modules in the system. They can also be completely replaced by a new implementation of the same interface. Modules are the *unit of compilation* and *unit of replacement*.

A module is described by a text that is comprehensible for a person, but that cannot be executed in that form by the computer. Processors expect executable machine code, which is hardly readable for a human. This is why several forms of the same module are necessary. The programmer writes a module in a high-level programming language, which is machine-independent, and then uses a *compiler* to translate it to a sequence of machine-dependent binary instructions. The textual form is named the *source text* (or *source file*), and the translated form is the *object code* (or *object file*). Since each module can be compiled independently of the others, one speaks of *separate* or *independent compilation*. The distinction between *separate* and *independent compilation* is explained below.

Separate Compilation and Interface Checking

The role of the compiler is not only to translate the source text into object code, but also to verify that the text is well-formed, in other words that it conforms to the syntax and semantics of the language. The context-independent

syntax can be checked as the text is read sequentially without context information, but checking the context-dependent syntax needs some additional information about declared objects such as their type and locality. For this purpose, the compiler manages an auxiliary data structure called the *symbol table*. Actually, the data structure is more complex than a table, since it reflects the hierarchy of nested scopes occurring in the program. Scopes themselves are not represented by tables either, but by sorted trees of identifiers that allow objects to be retrieved by their name.

The symbol table is constructed as declarations are parsed, and it is removed after the compilation of each module. Contrary to declarations, statements do not provide context information, and consequently do not contribute to the symbol table. They are instead directly compiled to machine code, to intermediate code, or to an abstract syntax tree for later processing.

The question coming naturally to one's mind now is: how can the compiler guarantee that objects visible across module boundaries are used in conformance with type compatibility rules? In other words: how can object declarations from one module be visible during the compilation of another module? Obviously, a mechanism providing a symbol table for external objects is necessary, a symbol table extract that is stored in order to be retrieved upon other compilations.

Some programming languages and assemblers avoid this problem by introducing external declarations informing the compiler that an object with the given name exists and is declared somewhere outside the currently compiled module. Usually, the external declaration mentions neither the exact origin of the object, nor its type, which makes interface checking impossible; and even if the type is provided, there is no guarantee that it is the right one. In that case, one speaks of *independent* compilation, in contrast to *separate* compilation, the latter performing full interface checking. Independent compilation will not be considered further here, since the techniques presented in this thesis improve the implementation of import-export mechanisms in modular programming languages like Modula-2 or Oberon, which already guarantee type safety across module boundaries.

This safety cannot be guaranteed by the compiler alone, as smart as it may be, without some support from the programming language in the form of clear concepts and adequate language constructs. Modula-2, for example, provides this support by both the *definition module* and the *import list*. A definition module is a separate text file specifying the interface of a module. It contains declarations of exported objects, making them available to partner modules, also called *clients* of the interface. In order to access the external object *X* exported by the interface of module *M*, a client *N* has to insert the name *M* in

its import list, thereby extending the scope of the interface of *M* to the module *N*. The qualified identifier *M.X* then refers to the imported object.

In Modula-2, the source text of a module consists of two files: the definition and the implementation part. The definition part consists only of declarations and does not contain any statements. Therefore, its compilation does not produce code, but only a symbol table, which is linearized to a file called *symbol file*. This symbol file is actually a compact representation of the interface that will be reused to compile client modules of this interface. The compiler restores the symbol table in memory from the symbol file each time the name of this interface appears in the import list of the compiled module. On the other hand, the compilation of the implementation part does not produce persistent information, except the machine code which is written to a file called *object file*.

Some implementations of Modula-2 [7, 8, 9], as well as Modula-3 [10], do without symbol files. The symbol table is reconstructed by recompiling the definition part each time a client imports the interface, which is less efficient, but has the advantage to eliminate the burden of managing a supplementary file for each module. A drawback of this method is that some interfaces may rely on several other interfaces by reexporting imported types. Importing such an interface forces the compiler to recompile many other interfaces, from which only a few type declarations are needed. Depending on the module hierarchy, a significant decrease in compiler performance may be observed. This problem cannot occur with self-consistent symbol files, since they duplicate type declarations imported from other modules (see next chapter).

In Oberon, definition and implementation parts are merged. Exported objects building the interface are marked in the text by an asterisk in their declaration. This approach has several advantages: the programmer has always the interface at hand, and works only on one document. This is especially practical during the development of a module, where the definition and the implementation must be held consistent after frequent modifications. For documentation purposes, the definition part can be extracted automatically. This simplification also releases the compiler from a nontrivial structural comparison between definition and implementation parts that is necessary to detect defined but possibly not implemented objects.

Here is the example of a module *M*, client of modules *A* and *B*, exporting a procedure *Do* and a variable *max*. First, the Modula-2 version:

```
DEFINITION MODULE M;
   IMPORT A;
   VAR max: A.Type;
   PROCEDURE Do(arg: A.Type);
END M.

IMPLEMENTATION MODULE M;
   IMPORT A, B;

   PROCEDURE Do(arg: A.Type);
      VAR temp: A.Type;
   BEGIN
      temp := B.Transform(arg);
      IF A.Greater(temp, max) THEN max := temp END
   END Do;

BEGIN
   max := A.Min
END M.
```

And then the Oberon version:

```
MODULE M;
   IMPORT A, B;
   VAR max*: A.Type;

   PROCEDURE Do*(arg: A.Type);
      VAR temp: A.Type;
   BEGIN
      temp := B.Transform(arg);
      IF A.Greater(temp, max) THEN max := temp END
   END Do;

BEGIN
   max := A.Min
END M.
```

In Modula-2, two compilation steps (denoted by *C* below) are necessary. First, the definition part *M.Def* is compiled producing a symbol file *M.Sym*. Then, the compilation of the implementation part yields the object file *M.Obj*:

M.Sym := C(M.Def, A.Sym); M.Obj := C(M.Mod, M.Sym, A.Sym, B.Sym)

Only one step suffices in Oberon:

(M.Sym, M.Obj) := C(M.Mod, A.Sym, B.Sym)

Actually, it is desirable that the compiler detects and announces a modification of the interface of *M*, if a previous version already exists. Therefore, the Oberon compilation also involves *M.Sym* if the file can be found:

(M.Sym, M.Obj) := C(M.Mod, M.Sym, A.Sym, B.Sym)

The symbol file *M.Sym* is necessary for compiling clients of *M* (or *M* itself), but it is not used for the execution of *M*, which only requires the object file *M.Obj* (besides the object files of imported modules).

Program Linking and Loading

The module concept along with separate compilation allows to partition a program into units that can be edited, documented, stored, distributed and compiled independently of each other. However, when a unit is being executed, it is not independent any longer: concrete interdependences exist between the module and its imports. For example, the address of imported procedures must be known in the calling module. The task of the *program linker* (or shortly *linker*) is to merge separate object files into an executable unit in which external references between modules are resolved. Depending on the programming environment and operating system, the executable unit is stored in a file and loaded later into memory by the *program loader* (or *loader*) for execution; or the linked unit remains in memory for immediate execution without being saved as a file. In the latter case, one speaks of a *linking loader*.

The linking and loading tasks in the Oberon System are somehow special, because there is no clear distinction between the operating system and an application. The Oberon System, which consists of a hierarchy of modules, is *open*, in the sense that the symbol files of these modules are available to the programmer. An application module can therefore import system modules; the module can be linked and loaded during execution, thereby dynamically extending the functionality of the base system. In fact, a distinction between system and user modules vanishes.

The organization of the run-time system influences the linker in its task. Depending on the way external procedures are called (indirection tables or absolute memory addresses) and global variables are accessed, the code may require more or less link editing work. Usually, local procedure calls use relative

addressing and accordingly do not need changes by the linker. Internal and external global variables are typically accessed through absolute memory addresses inserted in the code at link time.

Besides its linking function, the linker is also responsible for checking consistency of dependent modules. Indeed, it can happen that a module interface is modified in some incompatible way, and that clients having been compiled prior to the modification are not recompiled. Such an error must be detected before execution, since unforeseeable behavior might ruin the effort of the compiler in type checking.

This consistency check needs additional information in object files. This could be a timestamp indicating the last time the module was compiled. The linker could verify that the chronology of the timestamps corresponds to a topological order of the modules relatively to their import relations. Actually, this is not a good solution since implementation modules may be recompiled in any order, providing that the interfaces have not changed. A better solution is to consider the timestamp of the last *interface compilation*. The timestamps for the own interface and for each imported one are inserted in the object file. This timestamping mechanism is usually adopted in current implementations of modular programming systems.

The main drawback of this method is that a modification of an interface in a purely upward-compatible way cannot be detected and that clients sometimes need to be recompiled unnecessarily. The essence of the present work is to show how these recompilations can be avoided by using a fine-grained consistency check.

Chapter 2

Symbol Files

During module compilation, object declarations are recorded in the symbol table, which is an internal data structure of the compiler, or more precisely, a graph of linked records. The symbol table is discarded after every compilation, except for the part corresponding to exported declarations that must still be visible when client modules are compiled later on. Since this does not need to happen in the same compilation session, and since main memory is not unlimited, this part is *externalized* to a file called the symbol file, which is a linear representation of the symbol table of the module interface. Each time a client imports this interface, the compiler restores or *internalizes* the symbol table from the symbol file.

The exact contents of a symbol file, as well as the details of internalization and externalization mechanisms depend directly on the internal structure of the symbol table, which in turn depends on the compiled programming language. However, the global concepts and principles remain the same for all traditional strongly-typed, separately-compilable programming languages. In the following, Oberon-2 [11] is taken as example to describe symbol tables and symbol files.

Symbol Table

There are two main constituents of symbol tables: *objects* and *structures*. Objects are named entities like declared constants, type identifiers, variables, and procedures. The object's name is the only attribute by which the object is retrieved and identified in its scope. Each object has a reference to a structure node representing the object's *type*. Structures themselves are anonymous and therefore never accessed independently in their scope, but always in conjunction with an object they describe.

Type Graph

Structures may be shared by different objects having the same type, but objects are unique and appear only once in a scope. A name is attributed to a type in

order to declare objects of that type at scattered positions in a module text. The structure representing that type is associated with an identifying object containing the type name. In that case, a link exists from the structure to the naming object. *Type aliases* are objects defining an alternative name for a type, but the already existing reference from the structure to the original object is not modified by aliases. This original name is called the *canonical* name of the type. Here is a simple example of an Oberon declaration:

```
TYPE
   A = ARRAY 2 OF INTEGER;
   B = A;
VAR
   a: A;
   b: B;
   c: ARRAY 4 OF B;
```

A is the canonical name of the array type. *INTEGER* is a predefined type name. *B* is an alias for *A*: an object declared as of type *B* is actually of type *A*. Accordingly, the two variables *a* and *b* are of type *A* and the variable *c* is of an anonymous type, an array of *B* (hence of *A*). This declaration is compiled into the following type graph (figure 2.1), where an edge to a structure means "*is of type*" and an edge back to an object means "*whose canonical name is*".

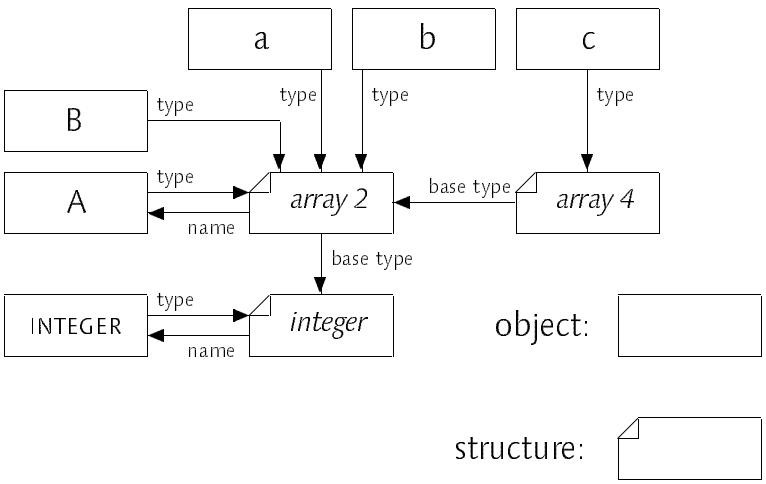

Figure 2.1 Example of a type graph

As shown in the previous example, structures may rely on further structures: an array consists of elements all of the same type. This element type can be a *basic* type, like the element type of *A*, which is an integer, or it can in turn be a *composite* type (also called *structured* type), like the element type of *c*, which is

an array of integers. A basic type is always a predefined and unstructured type, as opposed to a composite type, which is structured by components of further types.

If these components of a structured type are named, as in the case of record fields or parameters of a procedure type, the structure node also refers to named and typed objects. In contrast to what its name could suggest, the symbol table is a graph, and since recursive type definitions are possible, the graph may contain cycles. Example:

```
TYPE
   Ptr = POINTER TO Desc;
   Desc = RECORD
      do: PROCEDURE (VAR this: Desc; that: Ptr): BOOLEAN;
      list: Ptr
   END ;
VAR
   root: Ptr;
```

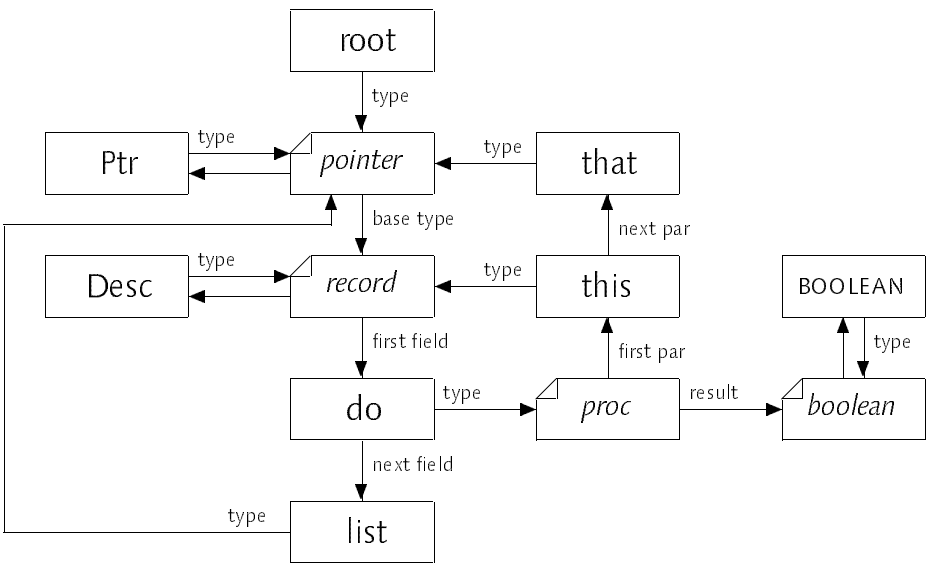

Figure 2.2 Example of a recursive type graph

The type graph describes the structure of user-defined types and objects, but does not give any information about the locality of declarations. Nevertheless, when an object is inserted into the symbol table, it must be checked that the object has not already been declared in the same scope; likewise, when an object is referenced, the object must exist in the current scope or in one of the

enclosing ones. This information about locality is also contained in the symbol table in the form of an additional and independent graph superimposed on the type graph.

Scope Graph

This second graph collects the objects of a scope and allows them to be retrieved by their name. Note that scope graphs consist of object nodes only and, unlike type graphs, do not involve structure nodes. Indeed, a structure is anonymous and does not appear alone in a scope, but always associated with a named object, as reflected by the type graph.

Usually, a scope graph is a binary tree of alphabetically sorted objects, but unsorted linear lists also yielded good results in the original Oberon compiler by N. Wirth [12]. Indeed, when the number of objects in a scope is rather small, which is usually the case, the gain due to the simple management of the list – only one pointer per object and no sorting – suffices to compensate for the loss of efficiency of the slower sequential search. Binary trees have a further drawback over unsorted linear lists: they do not keep track of the declaration order, which is important for formal parameters in procedure signatures and fields in records. For this reason, a third graph, a linear list this time, is superimposed on the binary tree for the objects whose declaration order must be known.

Therefore, the binary tree could be seen as an optional measure to optimize look-up operations in the symbol table only. Actually, it is also useful for a different reason: as explained later in this thesis, it is sometimes necessary to define a canonical order over a set of named objects, and this independently of the declaration order that may vary over several versions of the same module. The binary tree alphabetically sorts objects and hence simultaneously serves two purposes: efficiency and canonical ordering. Figure 2.3 shows the scope graphs for the same declarations as in figure 2.2:

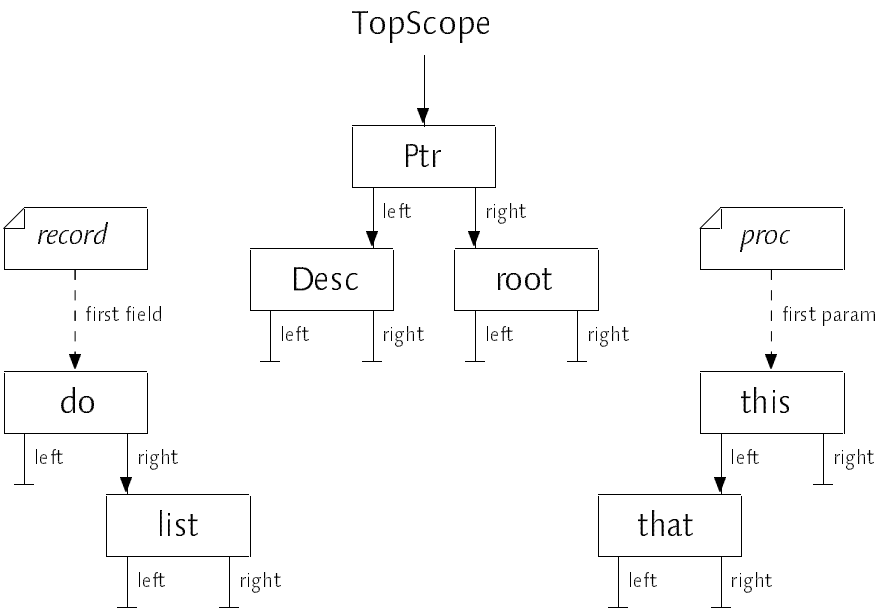

Figure 2.3 Example of scope graphs

The figure shows three scopes: a global one containing the objects *Ptr*, *Desc* and *root* of the outermost level, then a scope which is local to the record structure containing the fields *do* and *list*, and finally a scope local to the procedure structure containing the parameter *this* and *that*. The structure nodes as well as the dashed references actually belong to the type graph. Plain lines represent the tree of alphabetically sorted objects. Each scope is rooted at an object or at a structure of the enclosing scope. A global variable of the compiler named *TopScope* points to the first object of the current scope. This variable is the point where the whole symbol table is anchored in the table handler during compile time.

When the symbol table of an imported interface is internalized from a symbol file, a new module object is inserted in the global scope of the currently compiled module. The scope of the interface is then attached to this module object. Each time a designator of the form *M.A* is met during compilation, the object *M* is searched in the global scope; if M is found and is a module object, the object *A* is then searched in the scope attached to *M*.

Object and structure nodes are implemented as compile-time records linked by pointer fields (see chapter 4). The kind of object – constant, type, field, variable, procedure, parameter, and so on – is encoded in a field of the object, as well as the visibility of the object (internal, exported, exported as read-only). Similarly, the form of the structure – predefined type, pointer, record or array – is stored in a field of the structure, along with corresponding attributes like number of array elements or extension level in the record hierarchy.

The symbol table consisting of the information described above is machine-independent and reflects the textual declaration only, but other values are calculated by the compiler and inserted before and during code generation in the symbol table as well. One says that the symbol table is *augmented* by machine-dependent *attributes*. Among these attributes are the storage addresses of variables, allocation sizes and alignment factors of types, and entry points of procedures.

In order to produce a symbol file, the global scope graph of the symbol table is traversed, and each object that is marked as being exported is written to the file. The object alone does not suffice, but all related information that is necessary for using the object from a client module must also appear in the symbol file. In particular, the type of the object has to be externalized, too. More precisely, the complete type graph rooted in the object has to be linearized and written to the symbol file.

A Brief History

Symbol File Classification

Symbol files have not always been implemented the same way. A paper by J. Gutknecht [13] classifies the different file formats by using two criteria. The first one concerns the self-consistency of the file. If the symbol file of a module only describes objects declared in that module, objects being of some imported type are incompletely described. The missing information can only be completed by importing further symbol files. In that case, the first symbol file is not self-consistent and this method is said to be of class *A*. Note that this reexport of imported objects is only possible for types: constants, variables or procedures cannot be reexported. In contrast, if the symbol file flattens the module hierarchy by replicating parts of imported interfaces, the file is self-consistent and hence of class *B*. The module hierarchy is flattened in the sense that declarations stemming from modules at different levels in a module hierarchy are described together in the same symbol file. Thereby, clients do not have to internalize several symbol files in order to obtain the complete type information of an interface.

Using class-A symbol files in a system with a high hierarchy of modules may result in a serious degradation of compilation speed. Imagine for example an operating system whose resources are made available to applications through module interfaces, as it is the case in the Oberon System [14]. Some types declared in bottom modules of the hierarchy (e.g. Display.Frame in the Oberon

System) are imported and reexported by many interfaces. Each time a module importing such an interface would be compiled, the symbol file of the low-level interface declaring the type would have to be internalized. In the worst case, all interfaces of the operating system must be loaded to compile one application module.

The second classification criterion is the method used to encode the information in the symbol file. When the syntax used is very similar to the syntax of the source text, then the file is of class α. In the extreme, the symbol file is even replaced by the definition module itself, like proposed by Foster [7] and used in UCSD-Pascal [15] and Modula-3 [10]. The definition module being not self-consistent, the technique corresponds to the Aα class with the drawback explained above. That means that the definition module is recompiled each time it is directly or indirectly imported. When the symbol file is a compact representation of the symbol table, it is of class β. A scanner and a parser are necessary to build a symbol table from a class-α symbol file, whereas a very simple and efficient parser can directly load a class-β symbol file into a symbol table.

Symbol File Linearization

The first Modula-2 compiler developed at ETH [16] used a Bα technique, but the next generation of Modula-2 compilers [17] have used the more efficient Bβ method described by Gutknecht [13]. This method relies on a postorder traversal of the type graph, in the sense that all components of an object appear in the file before the object itself. For example, the type of an object is listed before the name and attributes of the object, record fields appear prior to their enclosing record type, and parameters prior to their procedure type. Since types may be shared by different objects, they are numbered as they appear in the file, and objects reference them with the corresponding number, therefore avoiding to list the same type several times. The first few reference numbers are reserved for predefined types. The postorder traversal of an acyclic type graph guarantees that a type is always listed before the number referencing it. This facilitates the reconstruction of the symbol table: a reference number can be replaced immediately by the type it represents as it is read from the symbol file, since the structure node for this type could already be allocated. Unfortunately, this nice property is not guaranteed any longer in case of cyclic type graphs. The following declaration illustrates the problem:

```
TYPE
   Ptr = POINTER TO Rec;
   Rec = RECORD next: Ptr END ;
```

According to the rule above, the pointer type *Ptr* cannot be listed before the record type *Rec*, since *Rec* is the base type of *Ptr*. On the other hand, *Rec* has a component *next* of type *Ptr*, so *Ptr* should be listed first. Obviously, this legal Modula-2 or Oberon type definition will require special treatment when exported. Note that this recursive type definition also needs special care from the compiler to be parsed correctly, since the identifier *Rec* is used prior to its declaration. Actually, this is the only exception to the Oberon rule stating that an identifier must be declared before being used. The solution to the problem is to break the cycle within the graph, to write the new acyclic graph to the file using the described method, but to leave a hint explaining how to close the cycle again when reading the file. More concretely, the graph of the example above is written to the file in the following acyclic form:

```
TYPE
   Ptr = #16 POINTER TO #0;
   Rec = #17 RECORD next: #16 END ;
FIX-UP
   #16 POINTER TO #17;
```

The first free reference number (#16) is implicitly assigned to the pointer type (reference numbers 0 to 15 are reserved for predefined types). This pointer type is temporarily undefined in order to point to a dummy predefined type with reference #0. The next unused reference number (#17) is implicitly assigned to the record type. The type of its field refers to the pointer type through the reference #16. The definition of the pointer type #16 can be corrected later to point to the record type #17, as indicated in the *FIX-UP* section.

In 1986, the first Oberon compiler written by N. Wirth [12] for the Ceres workstation [18] was derived from a Modula-2 compiler by the same author. Naturally, the existing technique for generating symbol files was adopted. At that time, definition modules still existed in Oberon. They were eliminated in 1990 because of the complexity they caused in the compiler (see next section), besides the advantages for the programmer not to have them, as explained in chapter 3. For compatibility reasons, the portable Oberon-2 compiler called *OP2* which I developed in 1990 [19] used the same symbol file format. OP2 is described in chapter 4.

Unfortunately, this technique has one major drawback: it is not able to linearize cyclic graphs, except those where the cycle is created by a forward

pointer declaration, as shown in the example above. More general declarations with recursive type definitions – like the one in figure 2.2 for example – cannot be exported. The fix-up trick is easy to apply for a pointer type, since only one reference in the symbol table – the base type of the pointer – has to be patched. The general case requires the patch of several scattered references for each cycle.

R. Griesemer showed a more natural method of linearizing symbol tables [20]. The method relies on a preorder traversal of the type graph. The idea is that the linear description of an object in the file may open a parenthesis containing the complete description of a component of the object. The description of the object is resumed after closing the parenthesis. If the inserted description recursively refers to the enclosing object, a reference number is used instead, thereby breaking the cycle. This reference number is also used for further nonrecursive references to this object as in the postorder method. The rule stating that all components of an object have to appear in the file *before* the object itself is relaxed, since the components will be described in the same order as they appear in the declaration of the object. The previous example of a forward pointer declaration would be linearized in the following way:

```
TYPE
   Ptr = #16 POINTER TO (Rec = #17 RECORD next: #16 END) ;
```

The example in figure 2.2 would yield the following sequence, where #2 stands for the predefined type BOOLEAN:

```
TYPE
   Ptr = #16 POINTER TO
      (Desc = #17 RECORD
         do: #18 PROCEDURE (VAR this: #17; that: #16): #2;
         list: #16
      END) ;
VAR
   root: #16;
```

The externalization algorithm used in the project of this thesis and described in more details in further chapters uses a preorder traversal of the symbol table and produces a symbol file of the Bβ class. This technique simplifies the corresponding internalization algorithm and simultaneously eliminates the implementation restriction disallowing the export of recursive type declarations, which is not forbidden in the Oberon language.

Chapter 3

System Consistency and Client Invalidation

A system of separately compiled modules is said to be consistent if each object being imported by some client module is effectively supplied by its exporter module. Furthermore, the effective type of each imported object must comply with the type the object had when the client was compiled. In other words, the implementation of each external object must be compatible with the object definition used to compile each client module of that object. Note that "compatible" is weaker than "identical". Indeed, in some cases as shown below, a slight modification of an object does not have any effect on its clients and the system remains consistent. However, most programming environments do not implement consistency checks with such a fine granularity. They usually do not consider objects of an interface separately, but interfaces as a whole; they declare a system inconsistent if a module interface supplied at link time is not identical to the one seen at compile time by a client of it. This rule widely applied to detect inconsistencies is actually too strict and often requires unnecessary recompilations in order to maintain system consistency.

Consistency Checking

A strongly-typed, separately-compiled programming language without consistency checking at link time is comparable to a car equipped with airbags, but without brakes. In other words, it does not make much sense. All the effort spent for security at compile time can be ruined at link time, since inconsistent modules cannot be detected. Executing them can have unpredictable and destructive effects on the rest of the system. One cannot speak of a *safe* language implementation without automatic consistency checking at link time. There are many different ways of implementing these checks. An important criterion is that checks that can be made by the compiler should not be repeated by the linker, unless it is necessary, and that the former should assist in reducing the effort required by the latter.

Granularity of the Checks

A system of modules may remain consistent after the modification of an interface in a compatible way, even if the clients of this interface are not recompiled. Here is an example of such a modification. First, the original version:

```
MODULE A;
   TYPE
      Ptr* = POINTER TO Desc;
      Desc* = RECORD
         visible*, hidden: INTEGER;
      END ;
   VAR
      p*, q*: Ptr;
BEGIN
   NEW(p); NEW(q); ...
END A.

MODULE B;
   IMPORT A;
   VAR i: INTEGER;
BEGIN
   i := A.p.visible
END B.
```

Module *A* is compiled first, then module *B*. The name and the type of the internal field *hidden* in the record type *A.Desc* are then modified in the following way:

```
      Desc* = RECORD
         visible*: INTEGER;
         newhidden: LONGINT;
      END ;
```

Module *A* is recompiled. However, module *B* does not need to be recompiled since the modification of a hidden field is not visible for *B* and has no effect on the use *B* makes of *A.Desc*. The system of modules remains consistent, and *B* is not invalidated. The fact that an invalidation occurs does not depend only on the modification itself, but also on the way the imported object is used. The same modification of *A* has no effect on *B*, but invalidates the module *C* below:

```
MODULE C;
   IMPORT A;
BEGIN
   A.p↑ := A.q↑
END C.
```

If module *C* is not recompiled, the record assignment will not copy enough bytes, since the modification altered the size of *A.Desc*. On the other hand, if the number of bytes to be copied is determined at run time from the type descriptor of *A.Desc*, *C* does not need to be recompiled.

As shown by the examples above, system consistency can be defined at different granularity levels, from the entire interface down to object components, and it may also take into account implementation issues. The complexity of consistency checks can therefore strongly vary from one programming environment to the other, depending on the method used. Checks with coarse granularity are very simple and efficient to perform, but they negatively influence the productivity of the programming environment since recompilations are more often requested than actually necessary. On the other hand, the cost of very fine-grained checks is not always compensated by rarely avoided recompilations.

Compilation Dependence Graph

Independently of the checking technique, a recompilation of the complete system always makes it consistent, as long as the modules are recompiled in a correct order. Since an imported module has to be compiled before an importing module, the partial order defined by the import-export relation is also the compilation order. The system of modules forms a directed acyclic graph called the *compilation dependence graph*, where the vertices are compilation units and the edges export relations. In programming languages with separate definition and implementation parts, like in Modula-2, reciprocal imports (in implementation parts only) are allowed, which apparently introduces cycles in the compilation dependence graph making a compilation impossible, as shown in figure 3.1.

```
DEFINITION MODULE A;
   TYPE T = RECORD a: INTEGER END ;
END A.
```

```
IMPLEMENTATION MODULE A;
   IMPORT B;
   ...
   B.b.a := 0;
   ...
END A.

DEFINITION MODULE B;
   IMPORT A;
   VAR b: A.T;
END B.

IMPLEMENTATION MODULE B;
   IMPORT A;
   ...
END A.
```

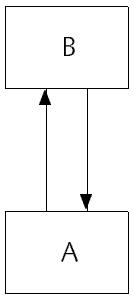

Figure 3.1 Cyclic import graph

In fact, each vertex of the cyclic import graph must be split in two subvertices yielding two separate acyclic graphs with edges from the definition graph to the implementation graph only, since an implementation part never exports anything (figure 3.2). It is therefore possible to first compile all definition modules in topological order, thereby producing the needed symbol files. The compilation of implementation modules may then occur in any order.

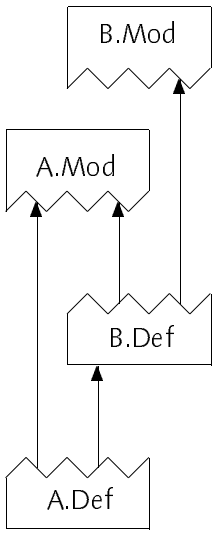

Figure 3.2 Corresponding acyclic compilation dependence graph

Reciprocal imports are not possible in Oberon, since definition and implementation parts are merged into one module. The import graph is therefore identical to the compilation dependence graph. Actually, cyclic import is also possible but not encouraged in Oberon, since it requires several editing/compiling sessions on the same module and surely does not reflect a clean programming style.

Using Keys for Consistency Checks

In most programming environments with separate compilation, consistency is guaranteed by a very simple rule stating that every client of an interface has to use exactly the same version of that interface. More practically, a unique symbol file for each module is used for all subsequent client compilations. In order to allow a check at link time, each symbol file contains the *timestamp* of its last compilation, which is a unique number derived from the date and time of the compilation. It may also be a number delivered by a random generator. The word *key* is often used in place of *timestamp*. If a module interface is modified and the module is recompiled, its symbol file receives a new key.

The key of a symbol file is copied to the object file of a compiled module using this symbol file. Note that self-consistent symbol files reexporting parts of further interfaces do not only contain their own key, but also the keys of these imported interfaces. So, each object file keeps track of the keys of all directly or indirectly imported interfaces, besides its own key. Example:

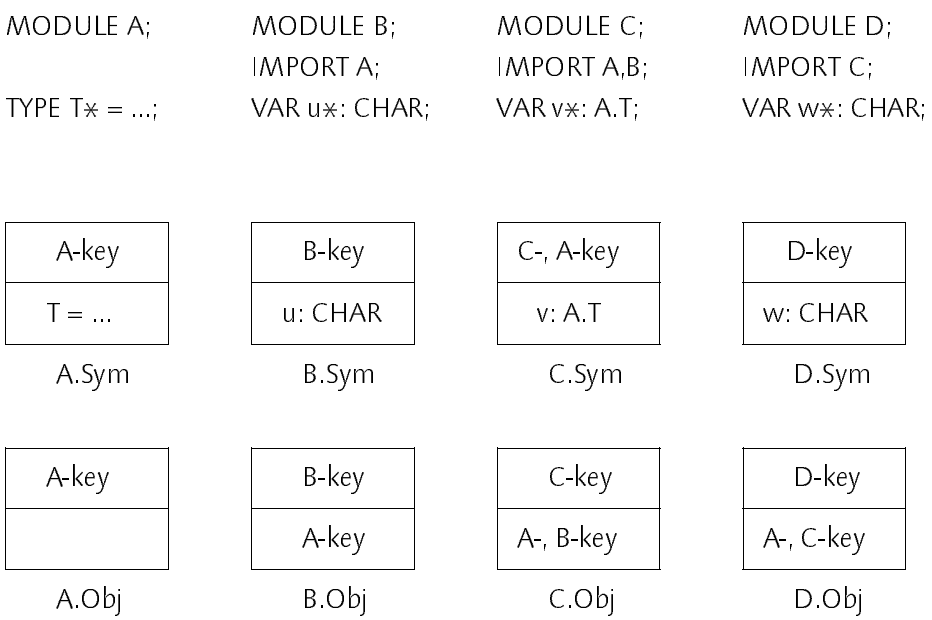

Figure 3.3 Key lists in symbol files and in object files

The linker compares the key lists in the imported object files of the modules being linked. An inconsistency is detected if the key of a module has different values in different object files. This occurs if a client of an interface is not recompiled when this interface has been recompiled and has received a new key. Consistency checking at link time is thereby reduced to the fast comparison of keys, one for each imported module.

Note that the keys of indirectly imported modules only would be sufficient for checking consistency at link time, but the keys of both directly and indirectly imported modules are listed in object files for simplicity reasons.

In implementations of Modula-2 using symbol file keys, the compilation of a definition part always results in a new symbol file with a new key, thereby invalidating clients of this interface, whereas compilation of an implementation part only produces a new object file without side effect. This is different in Oberon: remember that the compilation of an Oberon module produces both a symbol file and an object file. The original Oberon compiler [12] needs to compare the new symbol file with the old one in order to decide whether a new key is required. It would not be acceptable that the compiler generates a new key each time an implementation is slightly modified and recompiled. So, the newly generated symbol file is only definitively stored on disk if it is really different. Symbol files are written in a canonical form to allow an efficient byte-stream comparison.

In its very first version [21], the Oberon language had separate definition and implementation modules compiled in the same way as Modula-2 modules, but

with one major exception: the compilation of an implementation module could also generate a new symbol file and, as a consequence, invalidate clients. The problem was that the definition module was only defining visible fields of records, whereas the implementation part had to reprocess each record definition and possibly complete it with hidden fields. The compiler could therefore not always determine the size of an exported record by compiling the definition part only, so that a new corrected symbol file was generated after compiling the implementation part. Merging the two parts in one module elegantly solved the problem and eliminated the burden of checking the two declarations of the same record type for inconsistency, which involved a nontrivial recursive structural comparison.

Restoring System Consistency

Once a client has been invalidated, the system of modules becomes inconsistent and cannot be loaded before consistency has been restored. Client invalidation always occurs after the interface of a supplier module has been modified and recompiled, which generates a new symbol file for that module. Just recompiling the invalidated client does not always restore system consistency, since the interface of that client might reexport parts of the previously modified interface. In that case, recompiling the invalidated client results in a new symbol file for the client, which in turn invalidates further clients of it.

Trickle-Down Recompilations

This is particularly annoying if the modification concerns a *basic* interface, an interface at a low level in the system. Indeed, a basic interface often exports types and routines widely used in the system; its modification can cause *trickle-down recompilations* (recompilations triggering further recompilations) affecting a large portion of the system and thereby annihilates the benefits of separate compilation. Unfortunately, this happens quite often, because a basic interface is defined very early in a project and the abstraction it represents is not always well understood at that stage. As more clients of it are implemented, refinements of the interface are necessary, causing trickle-down recompilations.

Sometimes, recompilation is not sufficient or, in fact, not possible at all. This occurs, for example, when the component of an interface has been deleted but is still referenced by a client module, or when the formal parameter list of an exported procedure has been extended. In that case, the client needs editing

before being recompiled. This differentiates the *source invalidation* from the *object invalidation*; the latter class requires only a recompilation of the unchanged source text. Obviously, object invalidation is preferable to source invalidation, since consistency cannot be restored automatically without the intervention of the programmer in case of source invalidation.

Benefits of Tools

Unfortunately, the kind of invalidation depends intrinsically on the nature of the modification and cannot be controlled by tools, as smart as they might be. However, tools can help predicting the effect of a planned modification, or finding the modules affected after the modification is accomplished. They can also limit the recompilation to those modules that have really been invalidated.

A programmer, manually restoring the consistency of a module set, would probably follow the compilation dependence graph, starting from the first invalid module up to the top level modules of the hierarchy, while recompiling all visited modules. In spite of the danger of forgetting some of them, this is the right thing to do in the case of a coarse-grained consistency check on the interface level. However, if the linker uses fine-grained checks, it would be a waste of time to proceed in this way, because many modules would be recompiled unnecessarily. Indeed, the trickle-down recompilations may stop before reaching the top level modules of the graph, since the propagation of recompilations is dependent on the granularity of the consistency checks. Example:

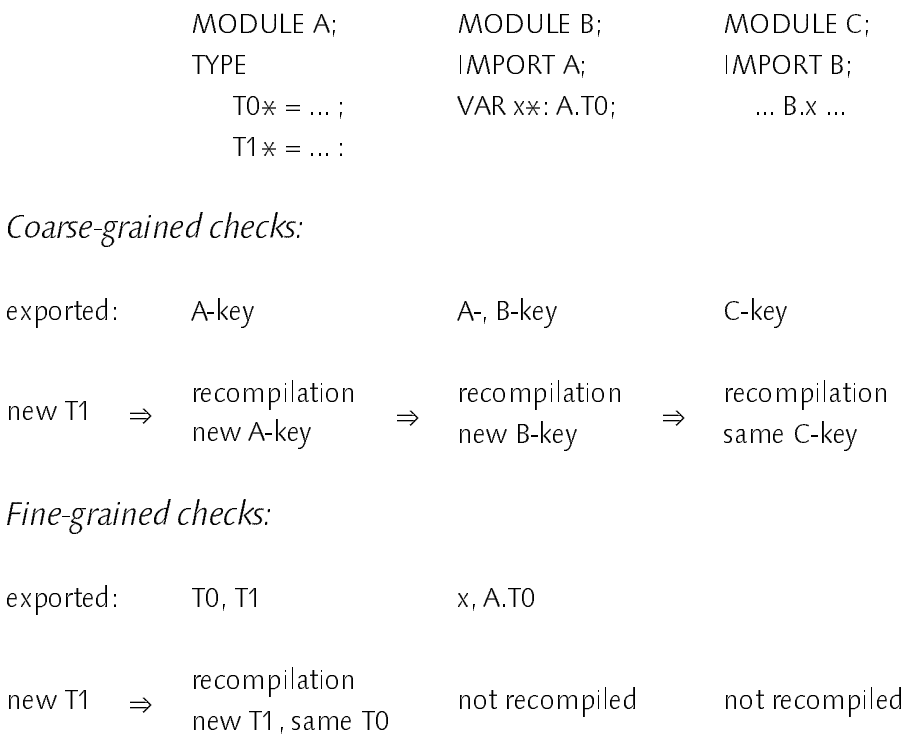

Figure 3.4 Granularity of the checks and number of recompilations

Using consistency check at the interface level may require more recompilations than when using consistency checks at the object level, as shown in figure 3.4; with coarse-grained checks, the modification of type *A.T1* and the recompilation of its module *A* result in a new key for *A*, which requires the recompilations of modules *B* and *C*, although they do not use *A.T1*. In contrast, with fine-grained checks, the clients of *A* do not need to be recompiled since they do not use the modified item.

Considering the number of interdependences, the programmer cannot decide by himself whether a client module needs recompilation, and cannot even tell whether a software system is consistent, unless he tries to link it. Clearly, the use of tools is compulsory to obtain the maximal gain out of a fine-grained consistency check method, especially if the first goal is to avoid redundant recompilations.

Previous work has sometimes concentrated too much on such tools while keeping old and unsafe module linkers unchanged. This is a questionable approach, since it is up to the programmer to use these tools performing the consistency checks. If he does not, the safety of the system is not ensured.

Background

There is not much literature to be found on the subject of link-time type checking of strongly-typed, separately compiled languages. Most work has been done between 1984 and 1989, with a peak in 1986. Still, even if research seems to have stopped by then, no entirely satisfying solutions have been presented yet.

Checksum Versus Timestamp

In 1984, as C. Bron, E. J. Dijkstra, and T. J. Rossingh [22] were about to implement link-time checking in their programming environment using a simple timestamping mechanism, they realized that this would only work in a universal module space. With the envisaged method, the symbol file (called *specification file* in Modular Pascal) of each interface would have been timestamped so that the linker could have checked that the code of each client is younger than the interface it relies on. But consider the situation where two different projects, *A* and *B*, are under development, each one in a different module space (in a different subdirectory of the same hierarchical file system, for example), and both of them using the same name *M* for two unrelated modules. If both modules *M* have been incidentally compiled at about the same time, and for some obscure reason, the object file *M.Obj* of *A* space is accidentally moved to *B* space, thereby replacing the correct *M.Obj* of *B* space, the linker will not be able to detect the inconsistency, because the timestamp of the erroneous *M.Obj* will be in a valid range.

The above observation convinced the authors to implement a different technique. For every interface a checksum is calculated, which is supposed to change with every change in the interface. The object file of every client module of this interface contains the interface checksum. At link time, the checksum of the effectively supplied interface must be identical to the one expected by the client.

Mesa, Modula-2, and Oberon use the same checking mechanism with the difference that the checksum is *unique*, in that it is derived from the date and time the interface is compiled. Actually, this is not a checksum – it is called *key* – since it does not depend on the contents of the interface. This has the advantage to work well in different module spaces, to be simpler and more efficient (no checksum to be computed), but the disadvantage that an interface can never be restored to a previous stage. For example, an accidentally deleted

symbol file cannot be recovered by recompiling the corresponding interface without receiving a new key, and thereby invalidating clients.

At about the same time, M. Rain proposed a solution involving both timestamps and checksums [23]. Symbol files, here called *unit dictionary,* are given a timestamp at creation time. Before each recompilation, the old unit dictionary is loaded into the *cache dictionary*, some kind of global symbol table, if it is not already present. During this operation, the compiler computes a checksum of the dictionary being loaded. As the compiler produces the new dictionary, it also computes a checksum in the same way. The dictionaries are considered the same if the checksums are equal. In that case, the old timestamp is reused for the new compiled unit.

Actually, M. Rain does not explain why he is using both timestamps and checksums. The checksum could replace the timestamp in the symbol file. Incidentally, note that the test at link time is very different for timestamps or checksums. On the one hand, each object file has only its own timestamp and the linker checks that this timestamp is *younger* than the one of every imported object file. On the other hand, each object file contains its own checksum and a list of checksums of imported interfaces, and the linker checks that all effectively supplied checksums are equal to the expected ones. So, the use of checksums is slightly more expensive but eliminates the problem of multiple module spaces.

Enhancing the Granularity

The main drawback of the above methods is the very coarse granularity of the corresponding link-time and compile-time checking. Changing a single line in an interface may trigger massive recompilations. In 1986, W. T. Tichy presented his *smart recompilation* algorithm to solve this problem [24]. The algorithm analyzes the effect of a modification in a module by computing a *change set* for the definition part of this module and a *reference set* for each dependent implementation part. The change set is computed each time an interface is recompiled by comparing the old and new symbol files, and consists of those objects that were either added, changed, or deleted. The reference set records the objects being imported by an implementation part. If the intersection of these two sets is empty, then a recompilation of the considered implementation part is not necessary.

The cost of computing the change sets is not negligible and is reported to be, on average, less than a third of the cost of a compilation. The technique is

not fully integrated in the compiler, but requires two separate tools in conjunction with the *Make* utility of *UNIX* [25].

R. Hood, K. Kennedy, and H. A. Müller extended the *smart recompilation* algorithm in order to determine the effects of a change not only on the direct clients of an interface, but also on its indirect ones [26]. The improved algorithm, called *global interface analysis algorithm*, propagates the change set along the compilation dependence graph, starting from the vertex of the modified interface. The change set is *filtered* through the contents of each traversed vertex, before being propagated to children vertices. More precisely, the change set is first filtered through the list of imported objects (i.e. the intersection of the change set with the set of imported objects is calculated). A nonempty result indicates that the module needs to be recompiled. The Tichy algorithm is then used to analyze the effect of this recompilation. The new change set is then filtered through the list of exported objects and propagated to the children.

The global interface analysis algorithm does not work directly on the object files, but on a private data structure. This requires some preprocessing on a well-defined set of modules, like constructing the compilation dependence graph or computing the filters for each vertex. The problem with this approach is that the auxiliary data structure has to be kept consistent with the system of modules under development. If a new module is added, for example, the data structure has to be recomputed. The paper does not specify how this is achieved. It neither mentions any safety aspect. The algorithm is (to be) embodied in an editor which interactively displays a list of modules affected by incremental changes. It is not clear what happens if a module is edited using another editor and then recompiled. The loader probably does not detect inconsistencies.

Integrating Type Information into Object Files

H. Eidnes, S. O. Hallsteinsen, and D. H. Wanvik implemented a smart separate compilation facility that has been used since 1983 in the CHIPSY programming environment for the CHILL language [27]. In contrast to the global interface analysis algorithm described above, the method integrates the information needed for the checks into the object files. This eliminates the burden of keeping a separate data structure up-to-date. Furthermore, the exported symbol table information is also stored in object files and not in separate symbol files. Like Oberon, CHILL has no textually separate interfaces.

Each object file in CHIPSY contains an array of exported objects (the *export interface*), an array of imported objects (the *import interface*), and a timestamp which is given to the object file when the module is compiled for the first time. With each exported object is associated a version counter and symbol table information. An imported object has a version counter and a cross-link to the exporter. The cross-link consists of a file name, a timestamp, and an array index, uniquely identifying the imported object. When a module is recompiled, the old timestamp is reused, and the new export interface is compared to the old one. If an exported object is modified, its version counter is incremented, but the object retains its position in the array. If an exported object is deleted, it leaves an empty slot in the array. Global type consistency is guaranteed at link-time if all cross-links to the same exporter have the same and correct timestamp, and if in each pair of imported and exported objects, the array index, as well as the version counter, have the same values.

This technique has some drawbacks caused by the fact that the history of development of a module is contained in its object file. If, for example, an object is removed from an interface and reinserted later, it is considered as a new object and it receives a different array index, thereby invalidating all the clients of this object. Furthermore, the problem in conjunction with multiple module spaces is not solved. Indeed, a module and its copy may be edited and recompiled with their clients in two different module spaces. The same object modified in two different ways sees its version counter being incremented by one in both spaces. If the duplicated module is copied back to the original space, recompiled clients become inconsistent and the linker cannot detect the problem.

The Modula-2 compiler from Digital's Western Research Laboratory for VAX and MIPS DECstations [9] implements link-time type checking by a tool called *intermodule checker* that is run separately from the linker. The compiler writes the type information of each exported and each imported object in the object file. The tool verifies by structural comparison that the same type information is used throughout the system of modules to be linked.

The latest version of the Modula-3 compiler from Digital's System Research Center [10] seems to use a similar technique as the one presented in chapter 6 of this thesis. [It should be stated that I presented the idea of this thesis during an internship at DEC SRC in summer 1990.]

Motivation

In 1986, W. Tichy noted in a paper [24] the consequences of using slow compilers: "With modern high-level languages, redundant compilations are a serious obstacle. The processing cost of making a minor change or adding a few declarations to a large system may be so great that it retards the growth and evolution of the system. At the very least, it imposes hours of idle time on development teams while everything is periodically recompiled from scratch. High compilation costs also tend to convolute system structure, because they force programmers to incorporate changes in ways that minimize the number of recompilations, rather than preserve well-structuredness."

Knowing that the Oberon-2 compiler [19] for MIPS processors [28] consisting of 9 modules recompiles itself on a *Silicon Graphics' Indigo*^{2} (MIPS R4400, 150 MHz) in about 1.5 second user time, one could think that the statement above is not true any longer, and that all the care taken to avoid recompilations is not justified any longer, since compilers have become so fast now (which is partly due to faster modern workstations).

Unfortunately, there will always be slow compilers and inefficient programming environments, but the principal reason why this statement, especially its second part, is still true today comes from a completely different consideration. Indeed, the style of programming and the architecture of software systems have radically changed in the last few years. In the past, programs consisting of a set of modules used to be considered as a complete whole. These programs had to be used as they were, any modification or accommodation to the context being impossible. As shared libraries and dynamic loading become more widespread, a different kind of software systems seems to emerge. These systems consist of libraries of modules and serve as resource and base for future applications that are still unknown when the system is compiled. An application developed later can dynamically link and load modules of the library.

As developer and manager of a library, it is important to be careful not to invalidate clients unless absolutely necessary. Indeed, it is always unpleasant to have to explain to a client that his software will need a complete recompilation if he uses the new version of a library. In the extreme case, some clients are even unable to recompile their programs, because they simply lost the sources.

Hence, a slightly different goal is pursued today: instead of reducing the number of recompilations, which are cheap now, one tries to minimize the number of invalidations, which may have severe consequences when a library is used world-wide by many unknown clients. Even if the goal has changed, the

means to reach it are approximately the same, because avoiding an invalidation also avoids a recompilation. As shown above, using fine-grained checks is always better in terms of number of recompilations. However, the costs of a consistency check vary depending on the granularity of the checks, as determined by the definition of consistency. Finer granularity causes higher costs. A trade-off between flexibility in use and complexity of implementation must be chosen.

The role played by tools has also changed. In the past, research in the area aimed at producing tools capable of automatically restoring consistency after an invalidation in a system of well-defined modules. Today, it is impossible to restore the consistency since the invalidated clients are not known. Tools, whose use remains optional, have a secondary role now. In this work, more attention is given to the compiler and the associated linker. These have to collaborate in order to maintain integrity.

Chapter 4

The Portable Oberon-2 Compiler OP2

Two new models for compile-time and link-time consistency checking of separately compiled modules are presented in the following chapters. Both models have been implemented in the portable Oberon-2 compiler OP2 [19] described in this chapter. OP2 was developed to port Oberon onto commercially available platforms. Like the original Oberon compiler [12], from which it was derived, OP2 is itself written in Oberon and its first version produced code for the Ceres workstation [18]. Since then, OP2 has been modified slightly to accept Oberon-2 programs [11] and different code generators have been written to cover today's most commonly encountered processor architectures [29].

Architecture of the Compiler

In contrast to other programs written in a high-level programming language, a compiler cannot be merely ported. The program has to be modified to produce different code for the new machine the language is to be ported to. Therefore, it is worthwhile paying attention to portability *before* constructing the compiler. The time invested in designing a well-structured compiler, separating machine-independent from machine-dependent parts, is rewarded many times when porting it. However, many compilers developed with the goal of being portable have turned out to be inefficient in terms of compilation speed and quality of compiled code. Portability and efficiency have been given equal importance in OP2. Therefore, automated retargetable code generation has not been considered. More conventional and faster techniques have been chosen instead.

In a single-pass recursive-descent compiler, all the tasks of the compilation are executed "simultaneously": for example, actions of syntax analysis, code generation and type checking are interleaved. Because all required attributes are passed on the procedure stack, no intermediate representation of the source text is needed between the different tasks. This makes the compiler compact and efficient, but not easily portable. Indeed, since machine-dependent and machine-independent tasks are closely coupled, it is difficult to modify the compiler for a new machine. One solution to the problem is to clearly separate the compilation tasks into two groups: a *front-end* consisting of the machine-

independent tasks (lexical analysis and syntax analysis, type checking) and a *back-end* consisting of the machine-dependent tasks (storage allocation, code generation).

Effectively, compilation thereby becomes a two-pass process, although the source text is processed only once. The interface between front-end and back-end is a complex data structure in memory instead of a sequential file, taking advantage of large stores. Only the back-end needs to be modified when the compiler is ported. The front-end enters declarations into a symbol table and builds an abstract syntax tree representing the program statements. If no errors are found, control is passed to the back-end, which generates code from this syntax tree. Since this structure is guaranteed to be free of structural errors and type inconsistencies, type checking and error recovery are not part of the back-end, which is a noteworthy advantage. Only implementation restrictions must be checked for.

Another advantage of an intermediate representation is that additional passes may be inserted to improve code quality. Such an optimization phase cannot be embedded easily in a conventional single-pass compiler, if at all. Also, an intermediate representation reduces the effort of porting several programming languages to a new target architecture, since a new back-end can be used with different existing front-ends.

Module Structure

The front-end and the back-end are implemented separately as a set of nine modules, all written in Oberon. The lowest module of this hierarchy is OP*M*, where *M* stands for *m*achine. One must distinguish between the host machine on which the compiler is running, and the target machine for which the compiler is generating code. Most of the time, the two machines are the same, except when using a cross-compiler. OPM defines and exports several constants used to parametrize the front-end. Some of these constants reflect target machine characteristics or implementation restrictions. For example, these values are used in the front-end to detect overflow conditions in the evaluation of constant expressions. OPM has a second function. It works as the interface between the compiler and the host machine. This interface includes procedures to read the text to be compiled, to read and write data in symbol files, and to display text (error messages, for example) on the screen. All these input and output operations are strongly dependent on the operating system. The compiler is structured in such a way that it can easily be ported to environments

other than the Oberon System. If the compiler resides in the Oberon System environment, the host-dependent part of OPM is based on the standard modules *Texts* and *Files*.

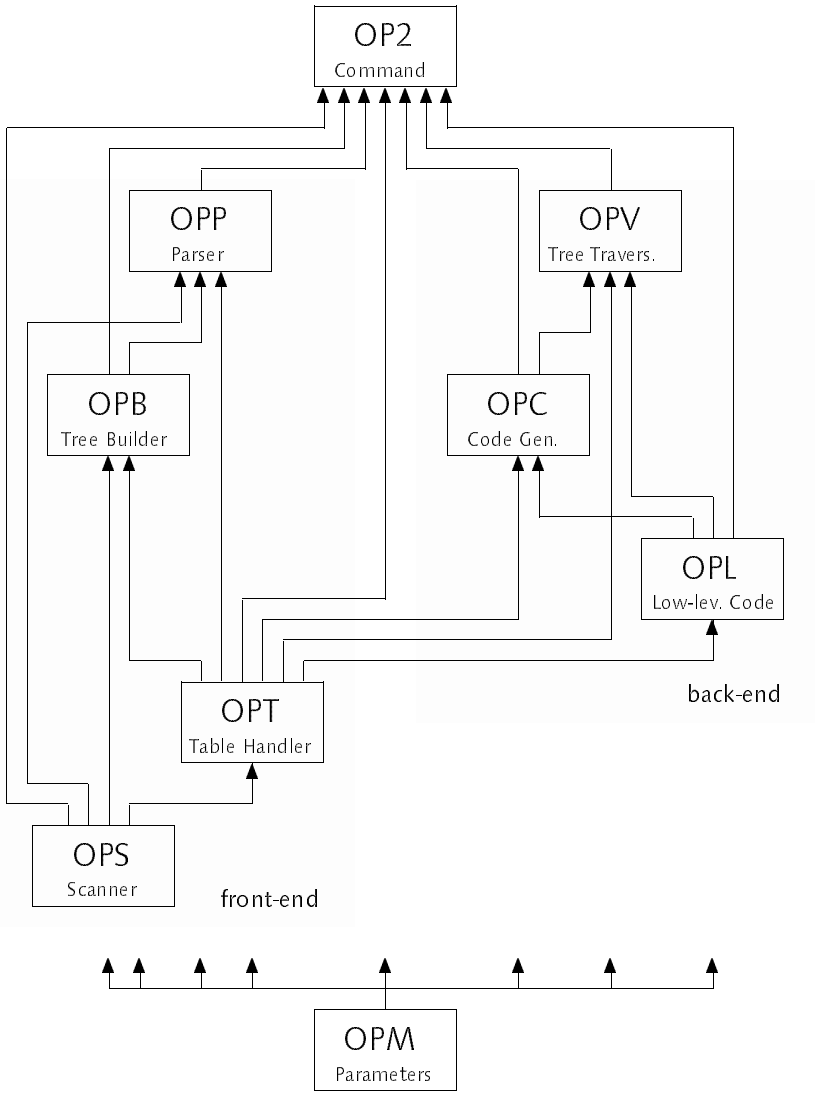

Figure 4.1 OP2 Module import graph

The topmost module OP2 is very short. It is the interface to the user, and therefore host machine-dependent, since input parameters like module file names and compiling options have to be read through host-dependent routines. Like the host-dependent part of OPM, this module remains unchanged when the compiler is used in the Oberon System environment. It first calls the front-end with the source text to be compiled as a parameter. If no error is detected, it then calls the back-end, passing the symbol table and the syntax tree that were generated by the front-end as parameters.

Between the highest and the lowest module, one finds the front-end and the back-end, which consist of four and three modules, respectively. During compilation, there is no interaction between these two sets of modules. The symbol table and the syntax tree are defined in module OPT and are accessed both by the front-end and the back-end. This explains the presence of import arrows from OPT to back-end modules visible in the import graph of figure 4.1. However, there is no transfer of control, such as procedure calls (some will be introduced later by the new consistency-check models).

The front-end is controlled by module OPP, a recursive-descent parser. Its main task is to check the syntax and to call procedures constructing the symbol table and the syntax tree. The parser requests lexical symbols from the scanner (OPS) and calls procedures of OPT, the symbol table handler, and of OPB, the syntax tree builder. OPB also checks for type compatibility.

The back-end is controlled by OPV, the tree traverser. It first augments the symbol table with machine-dependent data (using OPM constants), such as the size of types, the address of variables, or the offset of record fields. It then traverses the syntax tree and calls procedures of OPC, the code generator, which in turn synthesizes machine instructions using procedures of OPL, the low-level code emitter.

This module structure results in a fully portable front-end, as well as a host-machine independent back-end.

Symbol Table

The symbol table contains information about declared constants, variables, types, and procedures, as explained in chapter 2. It is built by the front-end. The front-end uses it to check the context conditions of the language and the back-end retrieves type information from it. In OP2, the symbol table is a dynamically allocated data structure with three different component types:

```
TYPE
   Object = POINTER TO ObjDesc;
   Struct = POINTER TO StrDesc;
   Const = POINTER TO ConstDesc;
```

An *Object* is a record (more precisely a pointer to a record), which represents a declared object, such as a variable, a named constant, a procedure, or a named type. The object declaration in the compiler is the following:

```
ObjDesc = RECORD
   left, right, link, scope: Object;
   name: OPS.Name;   (* identifier *)
   leaf: BOOLEAN;   (* procedure: leaf; variable: candidate to be allocated in register *)
   mode: SHORTINT;   (* constant, type, variable, procedure, or module *)
   mnolev: SHORTINT;   (* negative module no if imported, level no if local *)
   vis: SHORTINT;   (* visibility: not exported, exported, read-only exported *)
   typ: Struct;   (* object type *)
   conval: Const;   (* numeric attributes *)
   adr, linkadr: LONGINT   (* storage allocation *)
END ;
```

The name of the object stored in the object itself (field *name*) is used to retrieve the object in its scope. Each scope is organized as a sorted binary tree of objects (fields *left* and *right*) and is anchored to the owner procedure (field *scope*), which in turn belongs as an object to the enclosing scope. Parameters of the same procedure, fields of the same record and variables of the same scope are additionally linked sequentially (field *link*) to maintain the declaration order. Procedures that do not call any further procedures (leaf procedures) are marked by the front-end (flag *leaf*), as are variables whose addresses are never needed, and which therefore can be allocated in registers. The back-end may use this information for improving code quality. Note that this information would not be available in a single-pass compiler. An object always has a type, described by a *StrDesc* record, pointed to by a field *typ* in the object:

```
StrDesc = RECORD
   form, comp: SHORTINT;   (* basic or composite type, type class *)
   mno: SHORTINT;   (* imported from module no mno *)
   extlev: SHORTINT;   (* record extension level *)
   ref, sysflag: INTEGER;   (* export reference, system flag *)
   n, size: LONGINT;   (* number of elements and allocation size *)
   tdadr, align: LONGINT;   (* address of type descriptor, alignment factor *)
   txtpos: LONGINT;   (* text position *)
   BaseTyp: Struct;   (* base record type or array element type *)
   link: Object;   (* record fields or formal parameters of procedure type *)
   strobj: Object   (* named declaration of this type *)
END ;
```

There are several classes of types: basic types such as character, integer, or set, and composite types like array, open array, or record (fields *form* and *comp*). The field *form* denotes the exact class of a basic type or indicates a composite type, whereas the field *comp* denotes the exact class of a composite type or indicates a basic type. This separation allows to distinguish basic from composite types by an efficient integer comparison. It also permits a 16-bit imple-

mentation to use sets for efficient tests; this would not be possible with a single field, since more than 16 different classes exist.

The third element type of the symbol table is *ConstDesc*. This record contains numeric attributes of objects, like values of declared or anonymous constants:

```
ConstDesc = RECORD
   ext: ConstExt;   (* extension for string constant *)
   intval: LONGINT;
   intval2: LONGINT;
   setval: SET;
   realval: LONGREAL
END ;
```

An example involving the three different kinds of components is shown in figure 4.2 below:

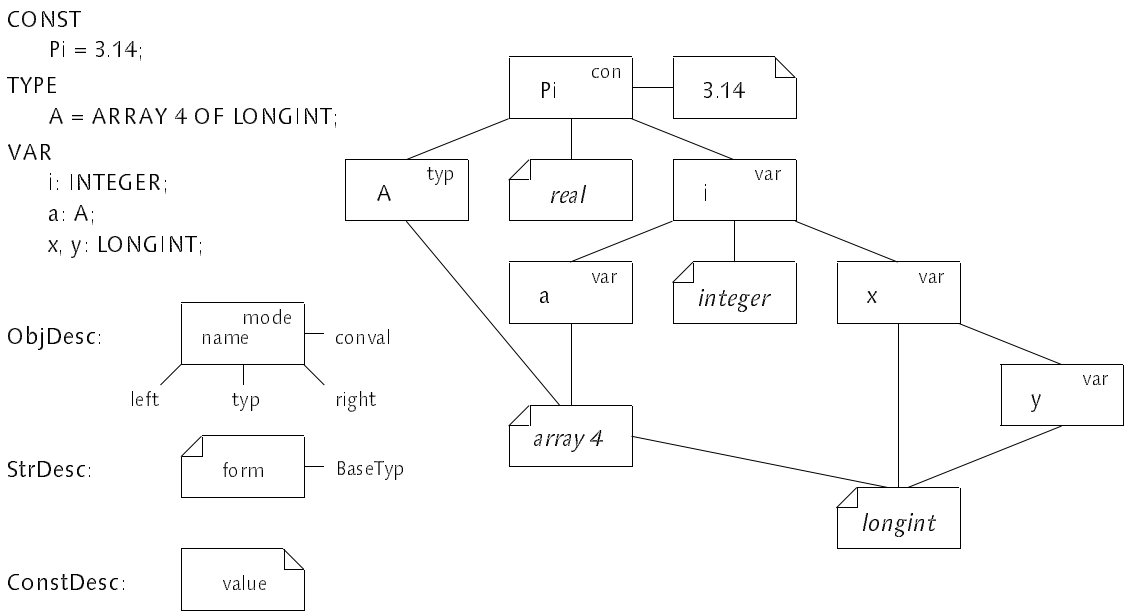

Figure 4.2 Declarations and corresponding symbol table

Syntax Tree

The front-end builds an abstract syntax tree representing all statements of the program being compiled. The Oberon syntax is mapped onto a tree of elements called *NodeDesc*:

```
Node = POINTER TO NodeDesc;

NodeDesc = RECORD
   left, right, link: Node;
   class, subcl: SHORTINT;   (* kind of node: construct, operation, expression *)
   readonly: BOOLEAN;   (* whether this expression is read-only *)
   typ: Struct;   (* type of the expression represented by this node *)
   obj: Object;   (* named object represented by this node *)
   conval: Const   (* constant value represented by this node *)
END ;
```

Each Oberon construct can be decomposed into a root element identifying the construct and a maximum of two subtrees representing its components: an assignment has a left and a right side, a While statement has a condition and a sequence of statements, and so on. Some Oberon constructs are organized sequentially: for example, lists of actual parameters in procedure calls and sequences of statements in structured statements. Auxiliary nodes might have been inserted to link these subtrees, yet an additional link field in the node is more space-efficient.

Each node has a class, and possibly a subclass, identifying the Oberon construct represented. It also has a type, which is a pointer to a *StrDesc* of the symbol table. Similarly, a leaf node representing a declared object contains a pointer (field *obj*) to the corresponding *ObjDesc* of the symbol table. A *ConstDesc* may be attached (field *conval*) to a node to describe a numeric attribute, such as the value of an anonymous constant. The position in the source text is stored in the root node of each statement. This facilitates locating compilation errors reported by the back-end. Figure 4.3 shows the representation of two statements involving variables declared in figure 4.2.

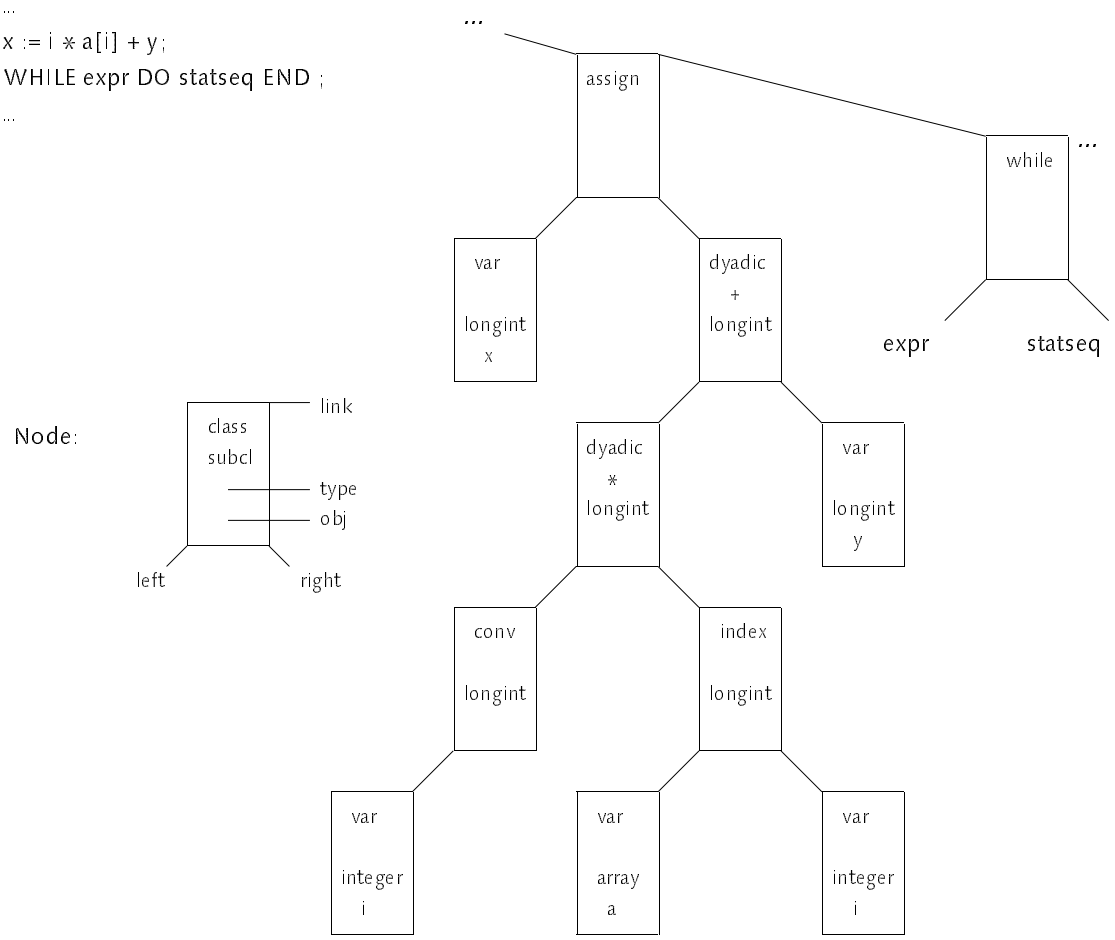

Figure 4.3 Statements and corresponding syntax tree

While generating code for a node, one typically has to evaluate left and right subtrees recursively, then the node itself, and finally the linked successors. A traversal of the tree looks like this:

```
Traverse(node: Node):
   WHILE node # NIL DO
      Traverse(node↑.left);
      Traverse(node↑.right);
      Do something with node;
      node := node↑.link
   END
```

The intermediate representation might have been a stream of instructions for a virtual machine, but an abstract syntax tree has been preferred for various reasons. Using an instruction stream may have some advantages in regard to register allocation, for example. The virtual machine may be defined as having a

infinite number of registers, which are manipulated by the instructions; some optimizations on register level are then possible by reordering or modifying the instructions. The infinite register set is then mapped in the back-end to a real register set.

An instruction set for a virtual machine would have been defined without any knowledge of future target machines. Perhaps the mapping of this instruction set to a real instruction set would not be easy, the virtual and real machines being very different (RISC and CISC, for example). Furthermore, generating these pseudoinstructions requires a code generator already, whereas building the syntax tree is a trivial recursive task easily embedded in a recursive-descent parser.

Trying to solve these problems by maintaining the instruction set for the virtual machine on a high abstraction level – using a stack instead of a register set, for example – does not help either, since most of the advantages of an instruction stream disappear.

Since the tree is a natural mapping of the Oberon syntax, each procedure of the parser returns as parameter the root of the subtree corresponding to the construct just parsed. Furthermore, a tree keeps the program structure intact, so that control-flow dependent optimizations can be integrated easily. Without a tree, an expensive control-flow analysis would be required, since basic blocks would have been dissolved in the linear code. The reordering of program pieces is easier to perform in a tree than in an instruction stream. For example, by first generating the statement sequence of a While statement (right subtree) and then evaluating the condition (left subtree), one branch instruction can be removed from the loop and replaced by an unconditional branch instruction executed only once.

Compilation Phases

As explained at the beginning of this chapter, the task of compilation can be divided into several more or less independent phases. Each phase works on some data produced by previous phases and provides input data for the following phases. The input data of the first phase is the source text of the module to be compiled. The output of the last phase is the object file containing the generated code. Additional input and output data in the form of symbol files are necessary in a context of separate compilation (figure 4.4).

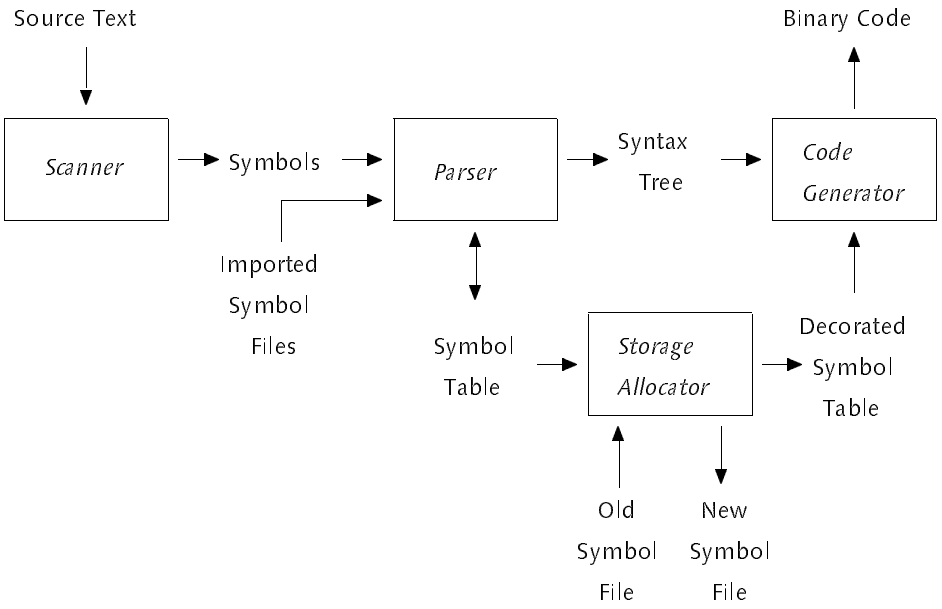

Figure 4.4 The different compilation phases

Lexical Analysis and Syntax Analysis

The scanner implements lexical analysis and the parser implements syntax analysis. Although these two tasks are implemented by separate modules, they execute in parallel and form the first compilation phase. Thereby, the sequence of symbols provided by the scanner does not have to be stored for later processing. Instead, the symbols are continuously passed, one after the other, as required by the recursive-descent parser. This is possible because the Oberon syntax can be parsed with one symbol lookahead. Similarly, the scanner needs only one character lookahead to decompose the character stream into a sequence of terminal symbols. In other words, lexical analysis and syntax analysis are done in a single pass on the source text.

The output of this first compilation phase performed by the front-end is the symbol table on the one hand, and the syntax tree on the other hand. From the front-end's point of view, the syntax tree is a write-only structure, but the symbol table is a read-and-write structure accessed for checking the context-dependent syntax. Context information for external objects is obtained from the symbol files (as explained in chapter 2). When the parser recognizes an import declaration, the corresponding symbol file is loaded into a separate scope of the symbol table.

Storage Allocation

In contrast to the previous phase, which is machine-independent, the next one, storage allocation, has to be parametrized depending on the target architecture. Storage allocation adds target-machine specific attributes to the objects of the symbol table. These attributes depend on the kind of object being considered:

- Every type gets a size attribute. For record types, a type descriptor is allocated and its address is associated with the type. Type descriptors are used at run time to perform type tests and garbage collection.
- Record fields get an offset specifying their position in the enclosing record.
- Variables get an address, or a register number if they are permanently allocated in a register.
- For procedures, a frame size sufficient to keep all locally defined variables is calculated. Exported procedures additionally need an identifying number – called *entry number* – so that they can be referenced from a client module. Oberon-2 type-bound procedures (methods) get a number that is used as an index into a method table.

Note that some attributes listed above, as well as the different allocation steps described below are specific to the original version of OP2. The storage allocation phase has been completely rewritten in the two models presented in the following chapters. Indeed, storage allocation is tightly coupled with module linking, which in turn depends on the method used for link-time consistency checking.

As shown in figure 4.4, the syntax tree is not affected by storage allocation. Some information inferred from the structure of the syntax tree – like whether or not a procedure is leaf, or whether a variable can be allocated in a register – is nevertheless useful for storage allocation. This information is already collected by the front-end as the tree is built and stored in the concerned objects. Thereby, the syntax tree does not have to be traversed during storage allocation.

Since external objects are imported together with their allocation information, they are not affected by this phase. It is therefore not necessary to traverse scopes of imported modules, but only the scope of locally declared objects. Storage allocation is done in the three following steps:

1. Exported types and procedures are visited in alphabetical order. Alphabetical order is preferable to declaration order: this allows the programmer to swap type declarations or procedure declarations in an already com-

piled source text. A recompilation of the modified text has then no influence on entry numbers or type descriptor addresses, which are stored in the symbol file. Different attribute values would enforce the creation of a new symbol file, thereby invalidating clients of the interface. Similarly, a modification of nonexported objects should be invisible to clients. This is why internal objects are visited *after* exported ones.

2. The list of (global) variables is traversed in the order of their declaration. Alphabetical order would be less appropriate, since allocation could not take advantage of a programmer's tendency to declare variables of the same size in clusters. This would result in a higher memory fragmentation caused by alignment requirements. If addresses of exported variables are not visible over module boundaries, but entry numbers are exported instead, entry numbers can be distributed in alphabetical order of the exported variables in step 1, whereas addresses are still distributed in declaration order of the variables in step 2.

3. The last step treats all remaining nonexported (global) types and procedures, as well as the local scopes of all procedures. Step 2 is applied for each list of local variables and step 3 is recursively called on each local scope. Traversal order is not important here, since objects decorated in this step never appear in an interface, and therefore have no influence on system consistency.

At the end of this phase, the symbol table is traversed and each exported object is linearized together with its allocation information into a symbol file. The original version of OP2 reads the old symbol file, compares it in a byte-wise fashion with the new one, and stores the new file on disk if it is different. Otherwise, the old file is retained with its old key (see chapter 2). When possible, the order in which the objects appear in the symbol file should remain the same among subsequent compilations, even if some declarations have been swapped. Here again, processing the objects in alphabetical order helps to avoid unnecessary invalidations.

Code Generation

The last compilation phase is code generation. The symbol table now contains sufficient information to emit code. Some allocation information about imported objects may still be missing (like the entry point addresses of imported

procedures). This information is inserted in the code by the linker at load time. The code accessing internally defined objects does not need to be patched at load time, since the relative addresses of these objects are known. However, depending on the processor architecture, absolute addressing is sometimes preferred to relative addressing. Therefore, the linker may also insert absolute addresses of internal objects in the code.

The task of code generation can be thought of as mapping every node of the syntax tree into a semantically equivalent code sequence, and storing these code pieces linearly into an array. Every code piece consists of zero or more machine instructions and, in general, depends on code generated for other nodes. These dependences should be kept as small as possible to allow for an efficient and systematic code generation process. A simple rule to enforce a high degree of locality in the code generator is to postulate that dependences exist between adjacent nodes only. In this case, dependences can be modeled as attributes flowing along the edges of the syntax tree. These attributes are recorded in an *item* that is passed as parameter (on the stack) instead of being stored in the syntax tree. The contents of an item are highly dependent on the target machine architecture. An excerpt of the module OPV is listed below. It gives an overview of the syntax tree traversal:

```
PROCEDURE design(n: OPT.Node; VAR x: OPL.Item);
(* generate code for the designator n *)
   VAR y: OPL.Item;
BEGIN
   CASE n↑.class OF
   ...
   | index:
      design(n↑.left, x);
      expr(n↑.right, y);
      OPC.Index(x, y)    (* x := x[y] *)
   ...
   END ;
   x.typ := n↑.typ
END design;

PROCEDURE expr(n: OPT.Node; VAR x: OPL.Item);
(* generate code for the expression n *)
   VAR y: OPL.Item;
BEGIN
   CASE n↑.class OF
   ...
   | dyadic:
      expr(n↑.left, x); ...
      expr(n↑.right, y);
```

```
        CASE n↑.subcl OF
        ...
        | plus:
            OPC.Add(x, y)    (* x := x + y *)
        ...
        END ;
    ...
    END
    x.typ := n↑.typ
END expr;

PROCEDURE stat(n: OPT.Node);
(* generate code for the statement sequence n *)
    VAR x: OPL.Item; L0, L1: OPL.Label;
BEGIN
    WHILE n # NIL DO
        CASE n↑.class OF
        ...
        | while:
            L0 := OPL.pc;    (* remember start address of loop *)
            expr(n↑.left, x);    (* evaluate conditional expression into x *)
            OPC.CFJ(x, L1);    (* if not x then jump to L1 *)
            stat(n↑.right);    (* do statement sequence *)
            OPC.BJ(L0);    (* backward jump to L0 *)
            OPL.FixLink(L1)    (* fix-up L1 with current pc *)
        ...
        END ;
        n := n↑.link
    END
END stat;
```

Code generation is not the subject of this thesis and will not be discussed further here. For more details about code generation and retargeting of OP2, see the technical report *The Oberon System Family* [29].

Chapter 5

The Layer Model

As observed in previous chapters, being able to modify module interfaces without invalidating their clients would be very convenient. The original OP2 compiler does not permit this, since any modification to an exported object always results in a new symbol file with a new key. Obviously, as smart and sophisticated the technique might be, deleting or modifying an object in an interface will always affect clients importing that object. On the other hand, extending an interface with a new object should be a harmless operation, since previously compiled clients do not see the newly inserted object.

This chapter presents a new model for interface extension and its realization in OP2. Note that the new model has been conceived with one special requirement in mind: the existing implementation of OP2 and of the linking module loader should need only few modifications in order to switch to the new model. Especially, the compiler back-end and the object file format should not be affected too much by the modifications, so that the new model can be adopted without programming effort onto the many different platforms running the Oberon System and OP2. However, the compiler front-end and the symbol file format can be entirely replaced since they are machine-independent and hence identical on all platforms.

The Idea

Clients of an exported object need some allocation information about the object to access it. These clients are invalidated when this information is modified. Extending an interface by some new objects should leave the allocation information for already existing objects unchanged, so that clients of these old objects do not need a recompilation. The part of the existing symbol file, which provides this allocation information, should remain unchanged by an interface extension. This can be the case if the additional symbol table information corresponding to inserted objects is *appended* at the end of the file. Conceptually, each time an interface is extended, a new *extension layer* appears on top of the existing interface.

A Stack of Extension Layers

Repeated extensions of an interface result in a multilayered interface. The compilation of an extended interface must yield a new symbol file containing the preceding symbol file as a prefix. In order to build such a stack of chronologically ordered layers, the routine generating the symbol file must first externalize exported objects belonging to the oldest layer, continue with objects of intermediate layers, and finish with newly inserted objects (if any). Obviously, it is not possible to determine the layer an object is belonging to, by only looking at its declaration in the source text. This information must be taken from the old symbol file.

The first compilation phase therefore reads the objects of the old symbol file into a separate scope. This scope is used during parsing to assign a layer number to a declared name. Note that it would be sufficient to keep in memory a table associating a layer number with each name present in the symbol file, instead of loading the complete symbol file into a scope. But for convenience, the same routine is used as for decoding and loading symbol files of imported modules.

The layer number of each object is determined by the position of the object in the symbol file, but is not explicitly stored in the file. Layers are separated in the file by a stopper. The first layer contains the objects with layer number 0.

The compilation consists of the following steps:

1. read own symbol file (*SF*);
   n := number of layers in SF;
   each object *old* from SF recalls its layer number: 0 <= old.layer < n;

2. parse source text, build symbol table and abstract syntax tree;
   for each declaration of a global object *obj*:
      if obj is exported and was present in SF: obj.layer := old.layer (0..n–1)
      if obj is exported and was not present in SF: obj.layer := n
      if obj is not exported: obj.layer := NotExported

3. allocate storage;

4. generate new symbol file:
   FOR i := 0 TO n DO externalize objects with obj.layer = i END

5. generate code;

The variable *n* indicates the number of layers present in the old symbol file. If the symbol file does not exist yet or does not contain any exported object, *n* is simply set to 0. In the second phase, the source text is parsed. When a global

exported object is declared, the scope of the old symbol file is searched for an object with equal name. If such an object is found, this means that the object was already exported in a previous version. In that case, the old layer number is copied to the new object (field *layer*), otherwise the object is new and is assigned layer number *n*. A nonexported object has no layer number (field *layer* set to *NotExported*, a large constant). After storage allocation, which distributes addresses and entry numbers to local and exported objects, the new symbol file is generated, layer after layer. The code generation phase completes the compilation.

Extensions Versus Modifications

The algorithm above does not take into account the case where an object previously belonging to the layer number *i* is not exported any longer, is modified, or is simply deleted. In that case, the stack of extension layers should collapse from the height *n* to the height *i*, all the objects with a layer number between *i* and *n–1* forming a new layer with number *i*. Indeed, modifying an object in a layer has consequences in storage allocation for objects in the same layer and layers above. The example in figure 5.1 shows the effect of editing changes to the number of layers in a symbol file originally consisting of 2 layers:

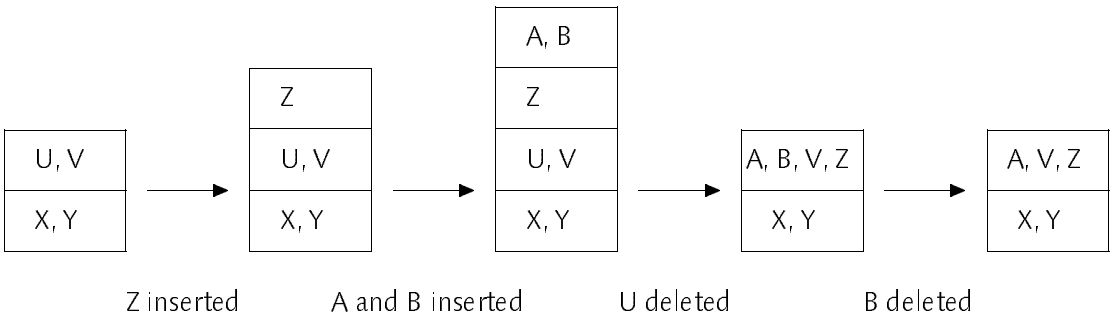

Figure 5.1 Growing and shrinking of extension layers

The algorithm has to detect differences between the old and new layers, and shrink the stack of layers if necessary. Such a difference indicates an interface modification that is more destructive than a simple extension: clients of the modified layer have to be invalidated.

Verifying that each previously exported object is still present in the new source text is not sufficient, but every exported attribute of it, like its type or its

allocation information, must also remain unchanged, otherwise clients have to be invalidated.

A structural comparison between the old and the new version of each object of the same layer is expensive. A global comparison at the layer level is preferable. Therefore, all objects of a same layer must be externalized in some canonical order, thereby allowing a byte-stream comparison of the new and old layers.

In order to avoid unnecessary invalidations caused by the insertion of an exported object anywhere in the source text, which is a legal interface extension, the allocation of a new object should not influence the allocation of objects in lower layers. Storage allocation has therefore to be done in the order of the layers, starting with the oldest exported objects, and finishing with nonexported objects. Furthermore, objects of the same layer must be allocated in a canonical order, so that swapping two object declarations in the text has no effect on storage allocation.

Consequently, both storage allocation and symbol file generation process the objects in the same order: canonical order for objects of the same layer (alphabetical order for types and procedures, declaration order for variables, see previous chapter) and chronological order of the layers. It is therefore possible to combine these two tasks into the same phase, thereby reducing the number of symbol table traversals. Here is a more elaborate version of the algorithm executing the different compilation phases:

```
1. read own symbol file (SF);
   n := number of layers in SF;
   each object old from SF recalls its layer number: 0 <= old.layer < n;

2. parse source text, build symbol table and abstract syntax tree;
   for each declaration of a global object obj:
      if obj is exported and was present in SF: obj.layer := old.layer  (0..n–1)
      if obj is exported and was not present in SF: obj.layer := n
      if obj is not exported: obj.layer := NotExported

3. allocate storage and generate new symbol file:
   i := 0; match := TRUE;
   WHILE (i < n) & match DO
      save allocation state;
      allocate and export types and procedures with obj.layer = i, in alphabetical order;
      allocate and export variables with obj.layer = i, in declaration order;
      match := new layer = old layer;
      INC(i)
   END ;
   IF ~match THEN discard last layer; restore allocation state END ;
```

allocate and export types and procedures with i <= obj.layer <= n, in alphabetical order;
allocate and export variables with i <= obj.layer <= n, in declaration order;
allocate remaining nonexported objects;

4. generate code;

The comparison between old and new versions of a layer takes place immediately after the allocation and generation of the new layer. If a mismatch is detected, this last layer is discarded and a new layer containing all remaining exported objects is created. This makes storage allocation necessary for all remaining objects, including the objects of the faulty layer, which have already been allocated once. The storage allocation for these objects is nevertheless repeated, since the set of objects to be allocated in canonical order may have changed.

Storage allocation reserves some global resources like entry numbers or memory space for allocated objects. Resources attributed to objects of the discarded layer must be released prior to the reallocation. The allocation state is saved before each allocation of a layer, and restored after a possible mismatch.

Consistency Checking

At compile time, the client of a module *sees* a stack of layers describing the exported interface. If an identical stack is provided at link time by the exporter, the client is consistent and can be safely linked to the module. Also, if the interface is extended in the meantime, consistency is guaranteed, because the stack of layers required by the client is present as a prefix of the effectively provided stack. However, if the stack is smaller, some required layer may not be present, and the client may have to be invalidated.

Comparing the height of the stack seen by the client at compile time with the height of the stack provided by the exporter at link time is not sufficient for detecting inconsistencies: the stack may be higher and yet not conform, because it may shrink and grow again in an incompatible way; it may also be smaller and nevertheless conform, because a client may not import objects from every layer it sees; a prefix of the visible stack may contain all required objects. The interface part following the required prefix may be modified or even deleted without invalidating that particular client.

For each imported module, the client specifies in its object file the number of layers it needs as well as a checksum, here called *fingerprint*, reflecting the contents of these layers. Note that a single fingerprint for all required layers is sufficient, because the client always imports a set of contiguous layers starting

from the lowest layer. Contrary to the client, the exporter has to list one fingerprint at each layer level, since a client may import any number of layers. When a client is being linked to an exporter, the linker first checks that the exporter supplies at least as many layers as required by the client. Then, it compares the effectively listed fingerprint for the highest required layer to the fingerprint expected by the client. Verifying one fingerprint at some layer level implicitly verifies all fingerprints below that level as well, because each fingerprint does not only depend on the layer contents but also on the fingerprint of the preceding layer. The advantages of using a fingerprint instead of a conventional timestamp will become obvious in the next section describing the implementation.

Figure 5.2 shows an exporter *M* with one layer and its client *A*. The export section of the object file *M.Obj* mentions the number of exported layers, one in the example, and lists the corresponding fingerprints, here *fp0*. The import section of the object file *A.Obj* indicates that the first layer of module *M* with fingerprint *fp0* is required.

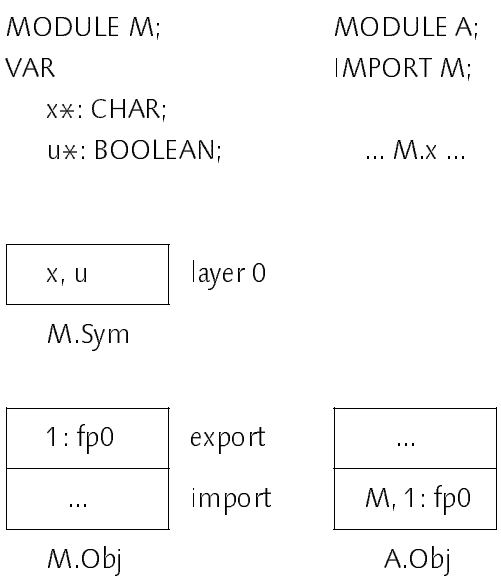

Figure 5.2 An interface *M* and its client *A*

Figure 5.3 shows an extension of module *M* by two exported variables, as well as two new clients, *B* and *C*. Note that client *A* is still consistent after the extension of *M* and does not need to be recompiled.

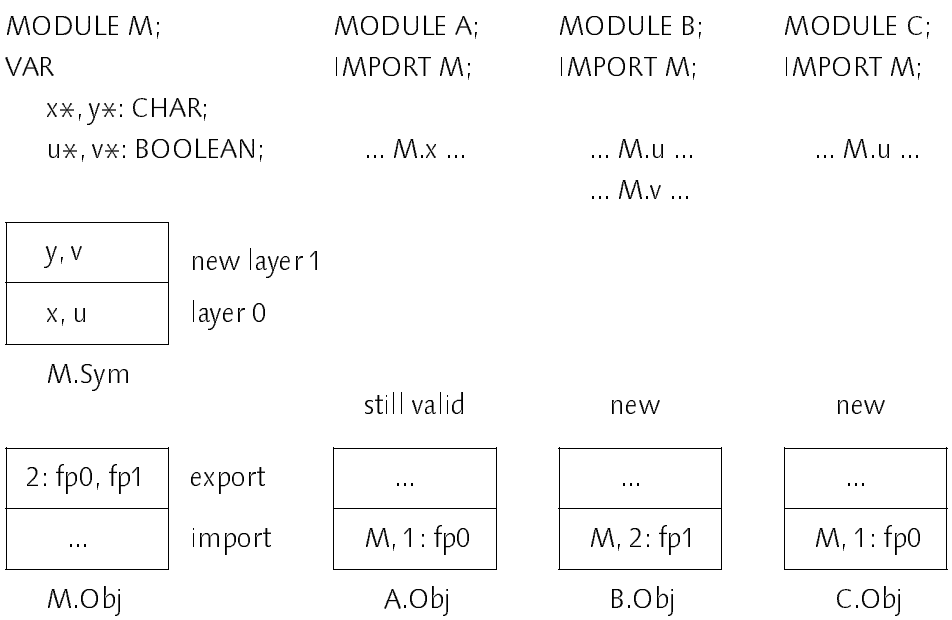

Figure 5.3 Interface extension without client invalidation

Although client *B* uses two layers from *M*, only one fingerprint corresponding to the topmost layer is listed in its object file. Client *C* sees two layers from *M*, but uses only one. Consequently, the second layer of *M* may be modified without invalidating *C*, and neither *A*. However, *B* is invalidated if any object of the second layer is modified, as shown in figure 5.4, where the variable *y* is dropped from *M*.

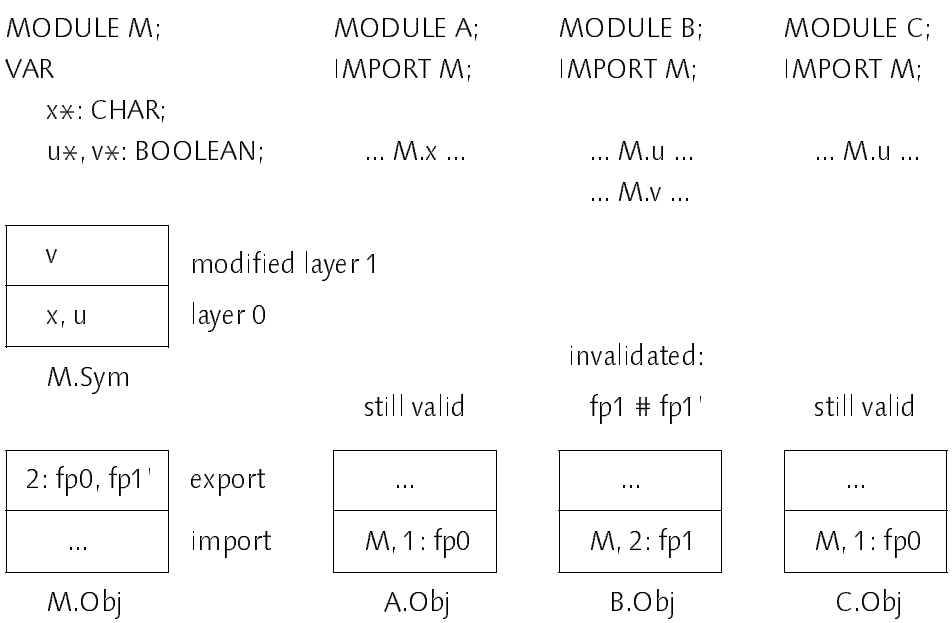

Figure 5.4 Interface modification resulting in client invalidation

Client *B* is invalidated: although it does not use the deleted variable *y*, it uses the layer the variable was removed from. Note that the variable *v* could also be deleted without affecting either *A* or *C*. In this case, the layer stack would shrink to the initial height, causing no problem to client C, which would nevertheless see a smaller stack at link time than at compile time.

The Implementation

The Layer Model has been implemented in the portable Oberon-2 Compiler OP2 described in its original version in the preceding chapter. OP2 was nearly left unchanged except for the modules OPT and OPV. A new symbol file format reflecting the stack structure required new import and export routines in OPT, and the OPV part performing storage allocation was rewritten. The part traversing the abstract syntax tree was left unchanged.

Symbol File Format

Because it was not possible to maintain compatibility between symbol files in the original model and in the layer model, the opportunity has been used to completely revise the symbol file format. The new format is still of the Bβ class, but corresponds now to a preorder traversal of the type graph, thereby allowing recursive types to be exported (see chapter 2). The following productions in EBNF syntax describes the structure of the symbol file:

```
SymFile     =  0FAX Module {{Object} FPrint}.
```

After the one-byte file tag and the module name, lists of objects separated by stoppers (*FPrint*) forms the different layers. The stopper consists of a one-byte tag (*FPRINT*) followed by the fingerprint value of the layer:

```
FPrint      =  FPRINT value.
```

> Note: identifiers in capital letters represent one-byte tags, identifiers starting with a capital letter are production names, other identifiers stand for numbers, except *name*, which represents a 0X-terminated list of characters. See the complete format in the appendix.

It would not be necessary to store the fingerprint of each layer in the symbol file, since it could be recomputed when the symbol file is loaded, but it is done for efficiency reasons. The additional disk space is insignificant, considering the relative small size of a fingerprint (LONGINT) compared to the size of a layer.

Each object is either a constant, a type, a variable, or a procedure:

```
Constant    =    CHAR value:1
               |  BOOL (FALSE | TRUE)
               |  (SINT | INT | LINT | SET) value
               |  REAL value:4
               |  LREAL value:8
               |  STRING name
               |  NIL.

Object      =    Constant name
               |  TYPE Struct
               |  ALIAS Struct name
               |  (RVAR | VAR) Struct offset name
               |  (XPRO | IPRO) Signature entryno name
               |  CPRO Signature len {code:1} name.
```

Note that the name of a type object (*TYPE*) is not listed, because it appears as canonical name of the object's structure (production *Struct*). This is explained in more details in the following. A type may appear in a symbol file without being marked as exported by an asterisk. This occurs for example when the type of a variable is not exported, but the variable is. On the one hand, the identifier of such a type should not appear in the symbol file, because clients are not allowed to use it. On the other hand, the compiler has to recognize types indirectly imported from different modules, in order to guarantee type equivalence. Now, Oberon uses name equivalence. Undoubtedly, the name has to be listed in the symbol file. This is also convenient for the browser which would have notation problems with anonymous types otherwise.

A clear distinction has to be made between the *structure* and the *object* of the type: the structure of a type always appears in the symbol file if any exported object of that type exists, but the object of the type only appears if the type itself is marked as exported. The question now is whether the name belongs to the structure or to the object. Obviously, the considerations above attribute the name to the structure, which is correct. Note that data structures in OP2 do not provide a *name* field for structures (record *StrDesc*, see previous chapter), but a pointer (field *strobj*) to a named object. The pointer avoids redundancy and contributes to efficiency.

The name in a type declaration is the canonical name of the type (see chapter 2). It is an attribute of the structure of the type that has to be listed with the structure in the symbol file. When a type is marked as exported, both its structure and its object are listed in the symbol file. Since the name is an attribute of the structure, it is not necessary to repeat it with the type object. Clients can only use the name if the corresponding object is exported. Type aliases introduce alternative names for the same type, or, in other words, new objects for the same structure. In that case, type alias objects are listed with their name, which is different from the already listed canonical name.

A previous version of the symbol file format bound the name to the object rather than to the structure. The object had therefore always to be exported with the structure. Exported and nonexported types had different tags. This did not cause any problems in the original model, but it would do so in the layer model. Indeed, a type not explicitly exported, but nevertheless appearing as hidden type in a symbol file layer, could not be made visible later on, without modifying the layer, since a new tag would be necessary. The new technique allows the structure to be listed alone in a layer, and the corresponding object in a subsequent layer. Marking the type identifier with an asterisk becomes a true extension.

The tag *RVAR* specifies a read-only variable, *VAR* a read-write variable, *XPRO* an external procedure, *IPRO* an interrupt procedure (which may have different calling conventions), and *CPRO* a code procedure (inline procedure which is used for hardware interfacing purposes). Storage allocation information has the form of an offset for variables and an entry number for procedures. Entry numbers could replace offsets for variables, which would allow variables of the same layer to be reordered in the source text without invalidating clients. Note that with either of the two methods, a new variable can be inserted at any position into the text without client invalidation, since the new variable is allocated in a new layer.

Procedures have a signature, whereas variables and types have a structure:

```
Struct      =     negref
                | STRUCT Module name [SYS value]
                  (  PTR Struct
                  |  ARR Struct nofElem
                  |  DARR Struct
                  |  REC Struct size align descAdr nofMeth {Field} {Method} END
                  |  PRO Signature).
```

The same structure may be referenced several times by different objects. The (positive) tag *STRUCT* introduces the first occurrence of a structure. Subsequent occurrences are replaced by a (negative) reference number. The first user-defined structure has reference number 16, since smaller reference numbers are reserved for predefined basic types. Indeed, the structure of standard types is implicitly known in every module and does not need to be described in symbol files. Every occurrence of a standard type is therefore replaced by a small (negative) predefined reference number (see appendix).

A structure may be a pointer referencing a further structure, a fixed or dynamic array of elements, a record – possibly extending some base record – with fields and methods, or a procedure type specified by a signature. Allocation information for record types is the allocation size, an alignment factor, the type descriptor address, and the total number of methods.

Fields, methods (also called *type-bound procedures*), and signatures are described by the following productions:

```
Field     = ((RFLD | FLD) Struct name | (HDPTR | HDPRO)) offset.

Method    = (TPRO Signature name | HDTPRO) methno entryno.

Signature = Struct {(VALPAR | VARPAR) Struct offset name} END.
```

Visible record fields may be read-only (*RFLD*) or read-write (*FLD*). Hidden fields represent nonexported fields, which are of a pointer (*HDPTR*) or procedure (*HDPRO*) type. The mark-and-sweep garbage collector needs to know the position of pointer fields in records. For this reason, type descriptors contain a table of pointer fields offsets. Information on hidden pointers may be used by the compiler to build type descriptors for record types extending an imported record type. This information is not necessary if type descriptors are built at load time. In that case, similar information must be stored in the object file, so that hidden fields can be found at load time in nested records as well as in global variables. OP2 prefers the first variant which is more convenient.

A safe implementation of module unloading may need the exact location of hidden procedure fields in imported records. Information about nonexported methods (*HDTPRO*) is necessary to build method tables at compile time. Three boolean constants exported from module OPM can disable the generation of information in the symbol file on hidden pointer fields, hidden procedure fields, and hidden type-bound procedures.

Signatures consist of a function result type – which can be the predefined type *NoTyp* in case of a procedure – and a list of value (*VALPAR*) or variable (*VARPAR*) formal parameters. Parameter names are present for documentation

purposes and are used by the browser only. The presence of parameter offsets avoids recomputing them when compiling caller modules. Formal parameter offsets may not be necessary depending on the calling conventions. Other allocation information is record field offsets, method table indexes, and method entry numbers.

A structure always has a name – which can be empty if the structure is anonymous – a module specifier (*Module*), and an optional system flag (*SYS*). The meaning of the flag is implementation-dependent and is interpreted by the back-end only. It can be set in the source text of low-level interfaces, for example, to mark redeclared types of the host operating system, which may have different allocation requirements.

Contrary to objects, structures may be reexported. As a consequence, they need a module specifier indicating their origin, so that the compiler can recognize the same structure indirectly imported via different symbol files. The module specifier indicates the module name and layer number the structure stems from:

```
Module      =  0 | ((negmno layerno | MNAME name) {FPrint} END).
```

The module whose symbol file is currently read has a module specifier equal to zero, because both the module name and layer number are implicitly known. Otherwise, a similar numbering scheme as for structures is used for module specifiers: a module name is only spelled out at its first occurrence (after the positive tag *MNAME*), subsequent occurrences are replaced by a (negative) number.

If an imported type is reexported, a module specifier lists the fingerprint value of the original layer declaring the type. Indeed, such a reexported type may be imported several times through different symbol files, and a possible inconsistency has to be detected at compile time. Therefore, the compiler verifies that the layer declaring the type has the same fingerprint in all symbol files.

The consistency check is more flexible at link time, since the fingerprints of the used layers only are listed in the object file and checked by the linker. The model thereby tolerates inconsistent fingerprints of unused layers, which is not a mistake, but a contribution to extension flexibility.

Each fingerprint is only listed once in the symbol file. The number of listed fingerprint determines the number of used layers at the first occurrence of a module (*MNAME*). The number of layers (*layerno*) is explicitly listed in subsequent occurrences of a module specifier. If this number is larger than in the first occurrence, missing fingerprints only are listed.

The example in figure 5.5 shows a module M and its client N. Module N reexports the types T0, T1, and T2, declared in the layers no 1, 0, and 3 of module M.

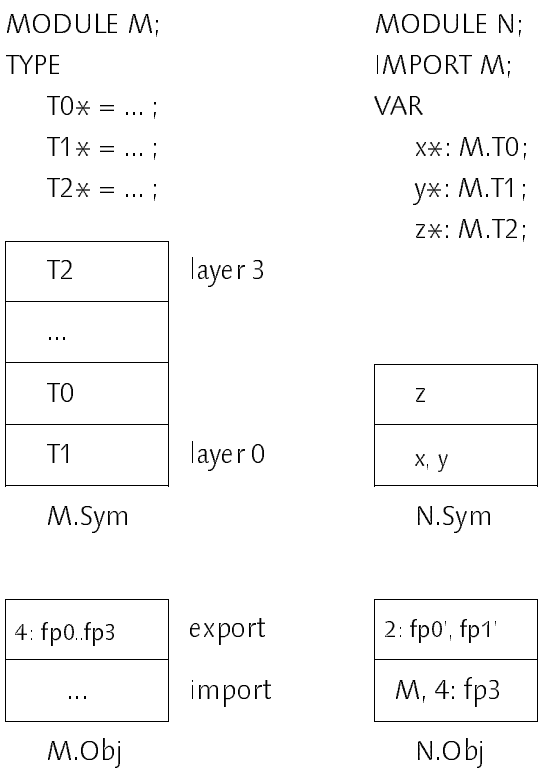

Figure 5.5 Type reexport in symbol files

Here are excerpts from the symbol file of N, which consists of two layers, and which reexports the types declared in M:

N's layer no 0:

... STRUCT MNAME M fp0 fp1 END T0 ...
fp0 and fp1 are listed ⇒ T0 belongs to M's layer no 1
... STRUCT –1 0 END T1 ...
–1 means module M, 0 means T1 belongs to M's layer 0 ⇒ fp0 is not repeated

N's layer no 1:
... STRUCT –1 3 fp2 fp3 END T2 ...
T2 belongs to M's layer 3 ⇒ fp0 and fp1 are not repeated

A fifth layer in M could be modified without invalidating the already compiled client N. Also, a client P importing both N and the new M would not notice that M has been extended after the compilation of N, because N does not use the new layers of M.

Consistency checking at compile time is performed when a layer is imported several times through different symbol files. The compiler verifies that such a layer has the same fingerprint in all symbol files.

The new symbol file format is machine-independent in that the same module OPT of the compiler is able to read and write a symbol file on different platforms without modifications, provided that the boolean constants controlling generation of hidden information are correctly set in OPM. However, the contents of the symbol file are not machine-independent, because some attributes, like record field offsets for example, may take different values on different machines. It would be possible to recalculate these offsets when loading the symbol file, but this would require the presence of every exported and nonexported field along with its type in the symbol file. This would not only take more space but also dramatically complicate the consistency check or introduce superfluous dependences for clients. Furthermore, the symbol file contents would not make any advantages of being machine-independent, since the symbol file can be reproduced at any time from the textual interface with no danger of invalidation for any clients.

Externalization, Fingerprinting, and Internalization

The algorithm for storage allocation presented in the previous section is implemented in a procedure of module OPV. This procedure traverses the scope graph of globally declared objects, allocates objects in the required order, and calls a routine *OutObj* in OPT for each object to be exported. *OutObj* initiates a recursive depth-first traversal of the object's type graph to serialize each node into a byte sequence as defined by the symbol file format. To each production of the format corresponds a serializing procedure:

```
PROCEDURE OutObj*(obj: Object);
BEGIN
  CASE obj↑.mode OF
  | Con:
    OutConstant(obj); OutName(obj↑.name)
  ...
  | XProc:
    OPM.SymWInt(Sxpro, expCtxt.fprint); OutSign(obj↑.typ, obj↑.link);
    OPM.SymWInt(obj↑.adr, expCtxt.fprint); OutName(obj↑.name)
  ...
  | Typ:
    IF obj↑.typ↑.strobj = obj THEN
      OPM.SymWInt(Stype, expCtxt.fprint); OutStr(obj↑.typ)
```

```
      ELSE
        OPM.SymWInt(Salias, expCtxt.fprint); OutStr(obj↑.typ); OutName(obj↑.name)
      END
    END
  END OutObj;

  PROCEDURE OutStr(typ: Struct);
  BEGIN
    IF typ↑.ref < expCtxt.ref THEN OPM.SymWInt(–typ↑.ref, expCtxt.fprint)
    ELSE
      OPM.SymWInt(Sstruct, expCtxt.fprint);
      typ↑.ref := expCtxt.ref; INC(expCtxt.ref);
      IF expCtxt.ref > maxStruct THEN err(228) END ;
      OutMod(typ);
      IF typ↑.strobj # NIL THEN OutName(typ↑.strobj↑.name)
      ELSE OPM.SymWCh(0X, expCtxt.fprint)
      END ;
      IF typ↑.sysflag # 0 THEN ... END ;
      CASE typ↑.form OF
      | Pointer:
        OPM.SymWInt(Sptr, expCtxt.fprint); OutStr(typ↑.BaseTyp)
      | ProcTyp:
        OPM.SymWInt(Spro, expCtxt.fprint); OutSign(typ↑.BaseTyp, typ↑.link)
      ...
      END
    END
  END OutStr;

  PROCEDURE OutSign(result: Struct; par: Object);
  BEGIN
    OutStr(result);
    WHILE par # NIL DO
      IF par↑.mode = Var THEN OPM.SymWInt(Svalpar, expCtxt.fprint)
      ELSE OPM.SymWInt(Svarpar, expCtxt.fprint)
      END ;
      OutStr(par↑.typ);
      OPM.SymWInt(par↑.adr, expCtxt.fprint);
      OutName(par↑.name); par := par↑.link
    END ;
    OPM.SymWInt(Send, expCtxt.fprint)
  END OutSign;
```

Terminal symbols of the file syntax are generated by procedures of OPM, like *SymWInt* for an integer or *SymWCh* for a character. These procedures also update the current fingerprint passed as second parameter. The fingerprint is

the result of a hash function applied to the sequence of values making up the layer. The fingerprint FP of a sequence of n+1 values V_0 to V_n is defined as follows:

$FP(V_0, V_1, \ldots V_n) := (FP(V_0, V_1, \ldots V_{n-1})\ XOR\ V_n)\ ROT\ 1$
$FP(\text{empty sequence}) := 1$

where *XOR* denotes a 32-bit exclusive or, and *ROT 1* a 32-bit rotate left by one bit. This recursive definition allows the procedure *FPrint* to compute fingerprints in an incremental fashion, like checksums. *FPrint* is called each time a procedure like *SymWCh* or *SymWInt* writes a value to the symbol file:

```
PROCEDURE FPrint*(VAR fp: LONGINT; val: LONGINT);
BEGIN fp := S.ROT(S.VAL(LONGINT, S.VAL(SET, fp) / S.VAL(SET, val)), 1)
END FPrint;

PROCEDURE SymWCh*(ch: CHAR; VAR fp: LONGINT);
BEGIN Files.Write(newSF, ch); FPrint(fp, ORD(ch))
END SymWCh;

PROCEDURE SymWInt*(i: LONGINT; VAR fp: LONGINT);
BEGIN Files.WriteNum(newSF, i); FPrint(fp, i)
END SymWInt;
```

Using fingerprints for consistency checking has several advantages over using unique keys derived from the compilation time and date. First, a fingerprint can be recomputed, because it is a hash function of the layer contents. Therefore, the same layer always results in the same fingerprint, independently of its compilation time. This is convenient when the same source text is compiled on different machines. Another advantage is that a newly generated layer can be compared to its older version by comparing their fingerprints, which is more efficient than reading the old symbol file a second time. This also avoids reading the newly generated symbol file.

The fingerprinting function should guarantee that any change in a layer is reflected in the fingerprint. Unfortunately, this is impossible because the fingerprint is of a finite length (32-bit integer here) and there can be more different layers than possible fingerprint values. Therefore, two different layers may have the same fingerprint, which is called a *fingerprint collision*. Taking the layer contents as fingerprint value would avoid collisions, but this is both impractical and inefficient in terms of memory usage and comparison time.

In practice, a fingerprint is nevertheless as safe as a timestamp, because a collision is not more probable than a breakdown of the computer real-time

clock (dead backup battery, for example). Even in the case of a collision, the consistency of the system of modules would most probably not be endangered. The only fatal collision is the one occurring between the original and the modified version of a same layer. In that case, the linker will not notice that nonrecompiled clients of that layer have been invalidated.

The probability of a collision can be reduced by simply increasing the fingerprint length. More sophisticated fingerprinting algorithms, like the MD5 Message-Digest Algorithm presented by R. Rivest [30] and intended for digital signature applications, could be employed here. However, these methods fulfill special requirements, like function irreversibility, which are not necessary in the layer model. More important here are the incremental aspect of computation, low probability of collision, and efficiency. In order to find a good fingerprinting method, one should take into account the influence of typical interface modifications on the probability of a collision. Undoubtedly, several PhD theses could be written on the subject. However, it is not manifest why the very simple fingerprinting function presented above should give poorer results than a more complicated one. Complexity is often superfluous. It is nevertheless possible to use another fingerprinting function by simply replacing the *FPrint* procedure in module OPM with no effect on the model.

The global record variable *expCtxt* contains the export context information, which is used and updated during the externalization phase. Fields of this record are for example the current fingerprint being computed (*fprint*) and the reference counter for exported structures (*ref*).

Similarly, a global record variable *impCtxt* manages import context information, which is used and updated during the internalization phase. Among other fields, tables are necessary to associate with a reference number an already internalized structure or module name. Predefined structures have predefined reference numbers smaller than the constant *FirstRef*.

The internalizing routines are the counterpart to the externalizing routines presented above:

```
PROCEDURE InSign(mno: SHORTINT; layer: INTEGER; VAR res: Struct; VAR par: Object);
   VAR last, new: Object; tag: LONGINT;
BEGIN
   InStruct(res);
   tag := OPM.SymRInt(); last := NIL;
   WHILE tag # Send DO
      new := NewObj(); new↑.mnolev := –mno; new↑.layer := layer;
      IF last = NIL THEN par := new ELSE last↑.link := new END ;
      IF tag = Svalpar THEN new↑.mode := Var ELSE new↑.mode := VarPar END ;
       InStruct(new↑.typ); new↑.adr := OPM.SymRInt(); InName(new↑.name);
      last := new; tag := OPM.SymRInt()
```

```
    END
  END InSign;

  PROCEDURE InStruct(VAR typ: Struct);
    VAR mno: SHORTINT; layer: INTEGER;
      tag: LONGINT; name: OPS.Name; obj, last, old: Object;
  BEGIN
    tag := OPM.SymRInt();
    IF tag # Sstruct THEN typ := impCtxt.ref[-tag]
    ELSE
      typ := NewStr(...);
      InMod(mno, layer); typ↑.mno := mno; typ↑.layer := layer;
      InName(name);
      IF name # "" THEN obj := NewObj(); obj↑.name := name;
        InsertImport(obj, GlbMod[mno].head↑.right, old);
        IF old # NIL THEN typ := old↑.typ
        ELSE
          obj↑.mode := Typ; obj↑.typ := typ; typ↑.strobj := obj;
          obj↑.mnolev := -mno    (* obj↑.layer = NotExported, name not visible here *)
        END
      END ;
      impCtxt.ref[impCtxt.nofr] := typ; INC(impCtxt.nofr);
      tag := OPM.SymRInt();
      IF tag = Ssys THEN ... END ;
      CASE tag OF
      | Sptr:
        typ↑.form := Pointer; ... InStruct(typ↑.BaseTyp)
      | Spro:
        typ↑.form := ProcTyp; ... InSign(mno, layer, typ↑.BaseTyp, typ↑.link)
      ...
      END
    END
  END InStruct;

  PROCEDURE InObj*(VAR obj: Object);    (* first number in impCtxt.nextTag *)
    VAR mno: SHORTINT; typ: Struct; tag: LONGINT;
  BEGIN
    tag := impCtxt.nextTag;
    IF tag = Stype THEN    (* type name visible now *)
      InStruct(typ); obj := typ↑.strobj; obj↑.layer := impCtxt.layer
    ELSE
      mno := ...;
      obj := NewObj(); obj↑.mnolev := -mno; obj↑.layer := impCtxt.layer;
      IF tag <= Pointer THEN    (* Constant *)
        obj↑.mode := Con; obj↑.typ := impCtxt.ref[tag];
        obj↑.conval := NewConst(); InConstant(tag, obj↑.conval)
      ELSIF tag >= Sxpro THEN
        ...
```

```
            InSign(mno, impCtxt.layer, obj↑.typ, obj↑.link);
            CASE tag OF
            | Sxpro: obj↑.mode := XProc; obj↑.adr := OPM.SymRInt()
            ...
            END
         ELSIF tag = Salias THEN
            obj↑.mode := Typ; InStruct(obj↑.typ)
         ...
         END ;
         InName(obj↑.name)
      END
   END InObj;
```

These routines do not have to recompute fingerprints, since each layer is followed by its fingerprint in the symbol file. Reading a symbol file consists of repeated calls to the procedure *InObj*, which returns an object to be inserted in the scope of the corresponding module:

```
InMod(mno, dummy);
impCtxt.nextTag := OPM.SymRInt();
WHILE ~OPM.eofSF() DO
   WHILE impCtxt.nextTag # Sfprint DO
      InObj(obj);
      InsertImport(obj, GlbMod[mno].head↑.right, old);
      impCtxt.nextTag := OPM.SymRInt()
   END ;
   InFPrint(mno);
   INC(impCtxt.layer);
   impCtxt.nextTag := OPM.SymRInt()
END ;
```

Writing a symbol file is a little bit more complicated than reading it, since storage allocation is performed at the same time. Furthermore, the last written layer may have to be discarded and storage allocation repeated in case of a layer mismatch.

Storage Allocation and Reallocation

During storage allocation, objects and structures are supplied with both context-independent attributes, like type size or field offsets, and context-dependent attributes, like address or entry number. The order in which objects are processed has no effect on the value of context-independent attributes, but influences context-dependent attributes. When an object is reallocated after a

layer mismatch, only its context-dependent attributes need to be recomputed. It is therefore advantageous to perform storage allocation in two distinct steps and only repeat the second step after a mismatch.

Moreover, the size of a type may be required already in the front-end by the Oberon standard function SIZE for constant folding. Since the source text is not entirely parsed at this stage, a complete allocation is not possible yet. However, the first allocation step, which includes the computation of the size, can be computed, since almost all context-independent attributes are defined, except for one. Indeed, since type-bound procedures may be declared outside the context of their record type, it is not possible to count them before the text is completely parsed. So, the number of methods is determined in the second step, although this number is context-independent.

Objects have no context-independent attributes, but context-dependent attributes only. Therefore, the first step is done on structures only, by a procedure in OPV called *TypeSize*. As indicated by its name, this procedure computes the size of the type passed as parameter, by first determining recursively the size, alignment, and offsets of the type components:

```
PROCEDURE TypeSize*(typ: OPT.Struct);
   VAR f, c: INTEGER; offset, size: LONGINT; align, falign: LONGINT;
      fld: OPT.Object; btyp, ftyp: OPT.Struct;
BEGIN
   IF typ = OPT.undftyp THEN OPM.err(58)
   ELSIF typ↑.size = –1 THEN    (* not yet computed *)
      f := typ↑.form; c := typ↑.comp; btyp := typ↑.BaseTyp;
      IF c = Record THEN
         IF btyp = NIL THEN offset := 0; align := 1
         ELSE TypeSize(btyp); offset := btyp↑.size; align := btyp↑.align
         END ;
         fld := typ↑.link;
         WHILE (fld # NIL) & (fld↑.mode = Fld) DO
            ftyp := fld↑.typ; TypeSize(ftyp);
            size := ftyp↑.size; falign := Base(ftyp); Align(offset, falign);
            fld↑.adr := offset; INC(offset, size);
            IF falign > align THEN align := falign END ;
            fld := fld↑.link
         END ;
         typ↑.align := align;
         Align(offset, Base(typ)); typ↑.size := offset;
         typ↑.n := –1    (* methods not counted yet *)
      ELSIF c = Array THEN
         TypeSize(btyp);
         typ↑.size := typ↑.n * btyp↑.size
      ELSIF f = Pointer THEN
```

```
            typ↑.size := OPM.PointerSize
         ELSIF f = ProcTyp THEN
            typ↑.size := OPM.ProcSize
         ELSE (* c = DynArr *)
            TypeSize(btyp);
            IF btyp↑.comp = DynArr THEN typ↑.size := btyp↑.size + 4
            ELSE typ↑.size := 8
            END
         END
      END
   END TypeSize;
```

The first line of the procedure catches erroneous type definitions that recursively use SIZE, like this one:

```
TYPE A = ARRAY SIZE(A) OF CHAR;
```

When a named type is being defined, the structure associated with the named object is still undefined (*undftyp*). This allows *TypeSize* to detect such illegal definitions. Other recursive type definitions resulting in cyclic type graphs are either correct, or already caught by the front-end. Cycles cannot cause *TypeSize* to loop forever, because there is always a pointer type or a procedure type in a cycle; the constant size of a pointer or of a procedure (*OPM.PointerSize* and *OPM.ProcSize*, usually one word) breaks the cycle. The initial value of -1 indicates that the size has not been calculated yet. The comparison with -1 avoids a second traversal of the type graph.

The field *align* in a record structure reflects the alignment constraint in number of bytes of the record. Usual values are 1, 2, 4 or 8 bytes. For example, a long real field may have to be aligned on a double word boundary (8 bytes). The procedure *Base* returns the alignment factor of the type passed as parameter. A record type takes on the strongest alignment constraint of its fields, so that the alignment is respected when the record is allocated as field in another record.

The procedure *TypeAlloc* performs the second allocation step. Contrary to *TypeSize*, *TypeAlloc* may reach the same node several times in a recursive type graph. It is therefore necessary to mark visited nodes to avoid infinite loops. An integer value is used to mark nodes, because a binary value, "visited" or "not visited", would not allow a type to be revisited during reallocation after a layer mismatch. Note that only the types belonging to the last layer need a reallocation; types already allocated in lower layers do not. Therefore, each type has to recall the layer number in which it was visited for the first time. Moreover, objects in local scopes are allocated after exported objects; since local scopes

are rooted in global objects or global structures, these global nodes have to be revisited. In addition, type graphs of exported objects are also traversed during export. Figure 5.6 shows the different states of a structure node during allocation and export:

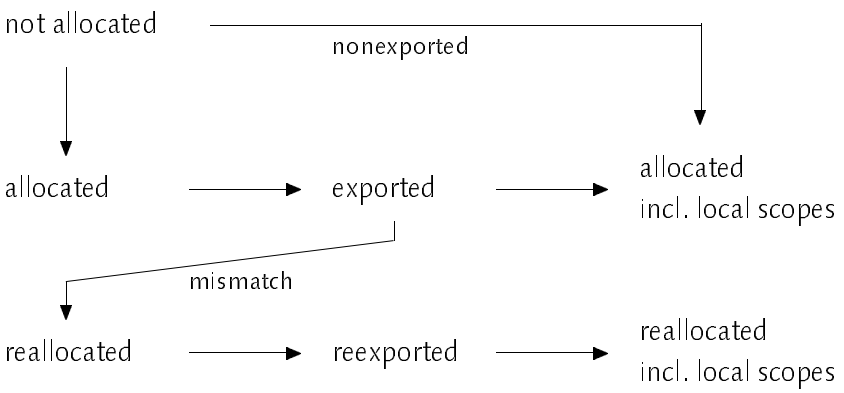

Figure 5.6 States of a user-defined type during allocation and export

The procedure *TypeAlloc* may reach type nodes being in any of these states. The procedure has to identify the state before deciding whether to (re)allocate the node or not: a node being already in the next state is part of a cycle and is not reallocated. When the first symbol file layer is built, the type nodes of the layer number 0 change from the "not allocated" state to the "allocated" state, and then to the "exported" state when export routines are called during the same traversal. All other type nodes remain in the "not allocated" state.

A second traversal of the scope brings all the type nodes belonging to the layer number 1 into the "allocated" and then "exported" state. This continues until all symbol file layers are generated. At this stage, all exported type nodes are in the "exported" state (or in the "reexported" state if a mismatch occurred) and all nonexported type nodes are still in the "not allocated" state. A last traversal revisits all nodes in order to find and allocate nonexported objects and types in local scopes and to bring remaining type nodes into the "allocated incl. local scopes" state.

The new field *stamp* in *StrDesc* encodes the state and layer number of each node in such a way that a simple comparison indicates whether the node has to be (re)traversed. The field *ref* in *StrDesc* indicates the reference number of the structure in the symbol file. This field is reset during a reallocation, so that the node can be correctly reexported. Note that reference numbers are valid in all symbol file layers, since an object in a higher layer may be of a type already defined in a lower layer. The following table shows the value of the fields *stamp* and *ref* for each state:

Table 5.1 State encoding during allocation

State	Field *stamp*	Field *ref*
not allocated	*NotAllocated*	= *NotExported*
allocated	4∗layer + 2	= *NotExported*
exported	4∗layer + 2	< *NotExported*
alloc. incl. local scopes	4∗layer + 3	≤ *NotExported*
reallocated	4∗layer	= *NotExported*
reexported	4∗layer	< *NotExported*
realloc. incl. local scopes	4∗layer + 1	< *NotExported*

An odd stamp value in a type node indicates that the type and the objects in the local scope of the type have been allocated. In each traversal, *TypeAlloc* decides whether to traverse (and stamp) a node or not, by comparing the node's stamp to a global stamp (*expCtxt.stamp*), which is updated for each new layer. The value of *expCtxt.stamp* divided by 4 always corresponds to the number of the layer being allocated. The first value of the global stamp is 2; it is then incremented by 4 after each layer, except after a mismatch, in which case it is decremented by 2. A node is (re)traversed if its stamp value is greater than the global stamp value. Hence, decrementing the global stamp by 2 after a mismatch forces nodes to be reallocated in the next traversal.

NotAllocated and *NotExported* are large constants whose values are chosen to simplify the stamp comparison (*NotAllocated* is even). Predefined types have a predefined *ref* field and do not need to be allocated in *TypeAlloc*. External types, imported from other modules, are not allocated, but since they may be reexported, they need a field *ref*. This field must be reset after a layer mismatch if the type was reexported in the discarded layer. For this reason, the procedure *TypeAlloc* also stamps external types, exactly like internal types.

```
PROCEDURE TypeAlloc(typ: OPT.Struct);
BEGIN
   IF typ↑.mno # 0 THEN    (∗ imported type, its size is already computed ∗)
      IF OPT.expCtxt.exported & (typ↑.stamp > OPT.expCtxt.stamp) THEN
         typ↑.stamp := OPT.expCtxt.stamp; typ↑.ref := OPT.NotExported;
         ... stamp all reachable external types and reset their ref ...
      END
   ELSIF typ↑.ref >= OPT.FirstRef THEN    (∗ not a predefined type ∗)
      stamp := typ↑.stamp; TypeSize(typ);
      IF OPT.expCtxt.exported THEN
         IF stamp > OPT.expCtxt.stamp THEN
```

```
            typ↑.stamp := OPT.expCtxt.stamp; typ↑.ref := OPT.NotExported;
            ... allocate all reachable types and reset their ref ...
          END
        ELSE
          IF ~ODD(stamp) THEN    (* not yet traversed with OPT.expCtxt.exported = FALSE *)
            IF stamp > OPT.expCtxt.stamp THEN    (* not traversed at all *)
              typ↑.stamp := OPT.expCtxt.stamp + 1;
              ... allocate all reachable types and local scopes ...
            ELSE    (* already traversed with OPT.expCtxt.exported = TRUE *)
              INC(typ↑.stamp);
              ... allocate all reachable local scopes ...
          END
        END
      END
    END TypeAlloc;
```

The field *exported* of the export context *expCtxt* indicates whether the current traversal allocates exported objects only. The fields *of expCtxt* are updated by the procedure *OutLayer,* each time a new layer is written to the symbol file. The fields *expCtxt.from* and *expCtxt.to* specify a range of layer numbers; objects with a field *layer* in that range are selected by the following two procedures to be allocated and possibly written to the symbol file:

```
PROCEDURE Variables(var: OPT.Object);
  VAR adr: LONGINT; layer: INTEGER; typ: OPT.Struct;
BEGIN ...
  WHILE var # NIL DO
    layer := var↑.layer;
    IF (layer >= OPT.expCtxt.from) & (layer <= OPT.expCtxt.to) THEN
      typ := var↑.typ;
      TypeAlloc(typ);
      IF var↑.stamp > OPT.expCtxt.stamp THEN
        var↑.stamp := OPT.expCtxt.stamp;
        NegAlign(adr, Base(typ));
        DEC(adr, typ↑.size); var↑.adr := adr; var↑.linkadr := adr;
        IF OPT.expCtxt.exported THEN OPT.OutObj(var) END
      END ;
    END ;
    var := var↑.link
  END ;
  ...
END Variables;

PROCEDURE Objects(obj: OPT.Object);
  VAR layer: INTEGER;
BEGIN
```

```
    IF obj # NIL THEN
      Objects(obj↑.left);
      layer := obj↑.layer;
      IF (obj↑.mode IN {Con, Typ, ... (* not Var *) }) &
          (layer >= OPT.expCtxt.from) & (layer <= OPT.expCtxt.to) THEN ...
        TypeAlloc(obj↑.typ);
        IF OPT.expCtxt.exported THEN OPT.OutObj(obj) END
      END ;
      Objects(obj↑.right);
    END
  END Objects;
```

The range specified by *expCtxt.from* and *expCtxt.to* contains only one layer, until a mismatch occurs. The range then includes the remaining layers to be exported. A last traversal with *expCtxt.exported* set to FALSE completes the allocation phase. In this last traversal, the range is set to include all exported and nonexported objects, so that every local scope can be reached and allocated. The following statements from the procedure *AllocAndExport* in OPV show the global structure of the allocation phase:

```
OPT.InitExport;
WHILE OPT.expCtxt.exported & OPM.noerr DO
  OPL.GetAllocState(allocState);
  Variables( first variable );
  Objects( first object );
  OPT.OutLayer;
  IF OPT.expCtxt.mismatch THEN OPL.SetAllocState(allocState) END
END ;
...
Variables( first variable );
Objects( first object )
...
```

The procedure *OPT.OutLayer* detects a possible layer mismatch and sets the different fields of the *expCtxt* variable. After a layer mismatch, the allocation state is restored and the variable *expCtxt.exported* remains TRUE for one more traversal. The allocation state indicates the amount of memory allocated for global variables and constants, and of the number of entries attributed to exported procedures and methods. The procedure *OutLayer* also resets the writing position of the symbol file rider to the beginning of the discarded layer.

Front-End Modifications

The implementation of the layer model requires only few modifications in OP2. Besides new OPT procedures that are described above, existing procedures of the front-end are not modified, except for two in the table handler:

- The procedure retrieving imported objects in external scopes recalls for each imported module the number of the highest accessed layer. This layer number is recorded together with the corresponding fingerprint in the object file and will be used by the module linker for consistency checking.

- After each declaration of an exported object, the table handler searches in the scope of the old symbol file for an object with the same name, in order to reuse the same layer number for the declared object.

The data structures in OP2 are left unchanged, except for the records *ObjDesc* and *StrDesc* in OPT, which both need two new integer fields, *stamp* and *layer.* The field *stamp* is used to mark nodes visited during allocation as shown above. The field *layer* recalls the layer number from which the object or the structure was imported. In addition, module OPT declares several new data structures, like import and export contexts, that are used locally by the new procedures, but have no influence on the rest of the compiler.

Object File Format and Linker Modifications

The back-end remains unchanged, except for the procedure generating the object file and the allocation procedures in OPV described above. Indeed, a slight modification of the object file format (see the complete format in the appendix) is necessary to list the number and fingerprints of exported layers:

```
HeaderBlk   =  ... noflayer:2 {fprint:4} modname.
```

The old object file format used to list a key only. Another modification concerns the import block where the number and fingerprint of the highest used layer from each imported module replace the key of that imported module:

```
ImpBlk      =  85X {noflayer:2 [fprint:4] name}.
```

Note that a module may be imported without being used; in that case, the number of used layers is 0 and no fingerprint is present. Here also, the old object file format provided a key for each imported module. Other aspects of the format remain unchanged.

According to the modified format, the linker performs a different consistency check. Instead of comparing the key of each imported module with the expected one, it now verifies that each imported module supplies at least the same number of layers as required, and then it compares the corresponding fingerprint with the expected value:

```
IF (LEN(m.fprints↑) < e) OR (e > 0) & (m.fprints[e–1] # fprint) THEN mismatch END
```

Where *m* is the imported module, *m.fprints* the array of fingerprints of the supplied layers, *e* the required number of layers, and *fprint* the required fingerprint. This test is performed for each imported module.

Drawbacks and Limitations of the Model

The layer model attains the objective stated at the beginning of this chapter: it is now possible to extend module interfaces without invalidating their clients. Updating an existing implementation of OP2 to the layer model is easy, because only machine-independent portions of the compiler need to be replaced. The minor modifications required in the back-end and in the module linker are trivial.

A new portable module browser, which is necessary due to the new symbol file format, is available and replaces the old one. Furthermore, the use of fingerprints instead of unique keys has several advantages. A fingerprint comparison is more efficient than a byte-stream comparison, for example. Also, the recompilation of an older interface version does not yield a new and incompatible key, but the original fingerprint. Similarly, an interface compiled in different module spaces gets the same fingerprint in all spaces.

Record Field Revelation

The model has nevertheless some limitations. One could expect that exporting a hidden record field, which is called *revelation*, is treated as an extension and does not cause clients of the record to be recompiled; but this is not the case. Indeed, a field made visible in a base record might cause a name collision with

a field of a record extending this base record. Slightly modifying the Oberon scope rules for record fields could solve this problem. Applying the concept of locality to the scopes of extending records would allow fields in extending and extended records to be declared with identical names. This would not be comparable to method overriding, since the name would always designate the same field independently of the dynamic type of the record. This could confuse the programmer and discourage good programming style.

The implementation of field revelation would also pose difficulties. The layer exporting the record with the hidden field should not be modified, otherwise clients would be invalidated. So, a kind of fix-up in a higher layer should reveal the field. That means that the declaration of a record would be spread over several layers. This fix-up mechanism would introduce complexity in the export routines: each type object might belong to a range of layers instead of to a single layer, and it might have to be traversed several times to be exported. A layer number would be assigned to each record field. Furthermore, several versions of the same record might be indirectly imported from different modules, which would require an additional consistency check. The reexport of a revealed field without client invalidation would be difficult to implement. Fix-ups would have to appear in a higher layer of the symbol file of the reexporting module.

For these reasons, field revelations, as well as method revelations, are not considered as extensions, but as invalidating modifications. Similarly, changing a read-only variable or field into a read-write variable or field would require fix-ups causing the same problems.

Extending a record with a new hidden field usually modifies the size of the record, and therefore the fingerprint of the layer. Also, a new hidden method results in a new fingerprint because the number of methods is modified in the symbol file. In both cases, clients of the record are invalidated.

History of Development

The main drawback of the model is that the history of the module's development is included in the symbol file. It is necessary to read the old symbol file to assign a layer number to an object. This number represents the age of the object and is not present in the source text. If the old symbol file is not available for some reason when the module is recompiled, the history of development is lost, and all exported objects then form a single layer. This may invalidate clients expecting a different stack of layers.

Also, deleting an obsolete object may invalidate clients, even if these do not use the object. Indeed, the stack of layers above the layer previously containing the object will collapse.

A solution to the problem would be to store history of development in the source text. A layer number could follow each export mark, for example. This would probably create other consistency problems, even more difficult to solve.

A better idea is to get rid of the history of development. History is only necessary to associate a layer number with each exported object, and to order the different layers. So, if each layer would only contain one object, and if the relative order of the layers would be irrelevant, then history of development would not be necessary any longer. This naturally leads to the *object model* presented in the next chapter.

Chapter 6

The Object Model

The OP2 version implementing the layer model avoids client invalidation by using the history of development that is stored in the symbol file of each interface. If the symbol file gets lost, the compiler cannot guarantee compatibility between the recompiled interface and clients of it, even if the interface is not modified. The original compiler also stores history into the symbol file in the form of a unique key derived from date and time of the first interface compilation. The recompilation of a module whose symbol file is lost may have disastrous consequences on system consistency.

This chapter proposes a second model for separate compilation and module extension that does not require the history of development. Applying this new model, the compiler can recompile modules without altering the system consistency, even if the old symbol files are not available. Symbol files do not keep track of history of development as in the layer model. Although old symbol files are not indispensable for a noninvalidating recompilation, the compiler nevertheless reads them to warn the programmer when an interface is modified in a way that may invalidate clients.

The Idea

System consistency is lost when some symbol table information stored in the symbol file of a module and expected by clients is modified after the recompilation of this module. Without history of development, it is not possible to allocate newly inserted objects after existing ones in order to keep the symbol table information for older objects unchanged. The problem is that older and newer objects cannot be differentiated without history. Therefore, objects of the same age cannot be grouped in the same layer and layers cannot be chronologically sorted. A single layer containing all objects and labeled by a single fingerprint would be a step back to the original model: any interface modification would always result in a new fingerprint and hence in a client invalidation. As a unique alternative, the new model chooses the other extreme; instead of packing all objects into a single layer, the model provides a layer and an individual fingerprint for each object.

A Fingerprint per Object

Conceptually, this model is a special case of the layer model with a finer granularity for consistency checking. There is nevertheless an important difference to the layer model: since these layers containing one object each cannot be sorted without history of development, a client cannot express its requirements by just specifying a layer number and a fingerprint, but it has to explicitly list every needed object with its fingerprint. Every module exports objects to clients and makes use of objects from imported modules. The new model discards the concept of layers and is called the *object model*.

Names and fingerprints of all exported objects are listed in the export section of an object file, but only the names and fingerprints of effectively used objects are listed in the import sections for imported modules. An external object is marked to be listed in the import section if its name is used in the source text being compiled.

The example in figure 6.1 shows an exporter *M* and its client *A*. The export section of *M* lists every exported object with its fingerprint, and the import section for *M* in *A* lists every used object from *M* with its fingerprint.

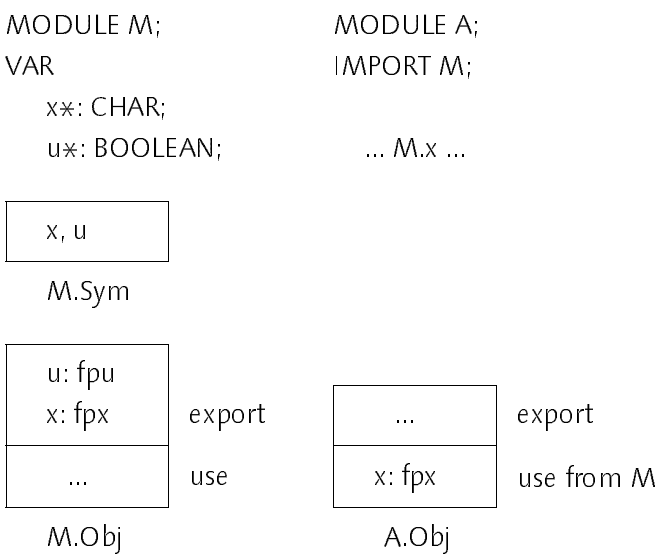

Figure 6.1 An interface *M* and its client *A*

Figure 6.2 shows an extension of module *M* by two exported variables, as well as two new clients, *B* and *C*. Note that client *A* is still consistent after the extension of *M* and does not need to be recompiled.

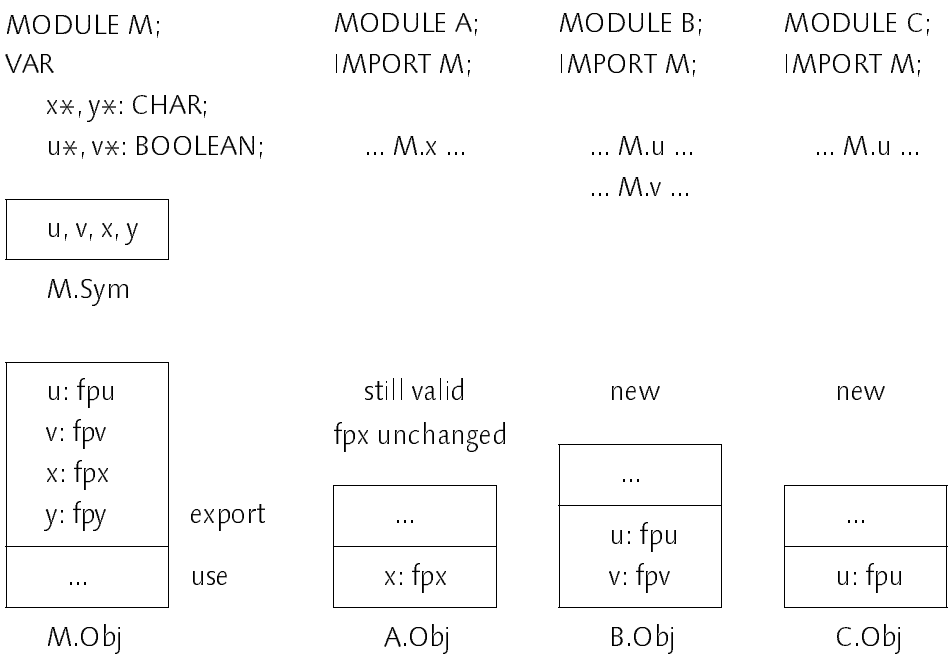

Figure 6.2 Interface extension without client invalidation

The fingerprint of the variable *x* is left unchanged by the extension of *M*. Therefore, the client *A* using *x* can be linked to *M* without being recompiled first. The linker verifies that exporter modules supply objects with the fingerprint values expected by client modules.

Note that the variable *y* is not used by any clients and can therefore be modified or even dropped from the interface of *M* without invalidating clients. In contrast, modifying the variable *x* will result in a new fingerprint value for *x* and *A* will be invalidated. Deleting *x* also invalidates *A*, since the linker will not find the fingerprint for *x* in the export section of *M* when linking *A*. However, modifying or deleting *x* neither affects *B* nor *C*.

In the layer model, the fingerprint of a layer is a hash function of the symbol file contents describing the layer and is also a function of the fingerprint of the preceding layer. As a consequence, the fingerprint is dependent on the history and is context-dependent. Fingerprints cannot be computed that way in the object model, since they have to be context-independent. Otherwise, the insertion of a new object could have side effects on the fingerprint of other objects.

Remember that fingerprints are only used to check consistency over module boundaries. Nonexported attributes of an exported object are therefore not relevant for the fingerprint computation. Obviously, the fingerprint of an object has to depend on its type, because the type is also an exported attribute of the object. The name of the type is not sufficient, because a modification of the

type structure may leave the name unchanged. Such a modification has to be detected by the module linker and must therefore influence the fingerprint value of the object. As a consequence, the fingerprint of an object is a function of all attributes defining the object in the symbol file, including those of the type of the object. The fingerprint of an object can only be context-independent if all exported attributes of the object are also context-independent.

A symbol file describing two objects of the same type does not list the type structure twice. The type of the second object is replaced by a number referencing the type of the first object. Now, the fingerprint of the second object should not depend on the presence of the first object. The reference number is a context-dependent attribute and hence cannot be used in the fingerprint computation. Structure reference numbers in symbol files can be seen as an optimization for avoiding the duplication of common type graphs. Similarly, a multiple traversal of a type graph should be avoided for efficiency reasons when computing the fingerprint of objects of a same type structure. As a result, every structure also receives a fingerprint. Structure fingerprints can be considered as common subexpressions in the computation of object fingerprints.

The fingerprint of an object is therefore a function of the fingerprint of its type structure. A clear distinction is made between objects and structures. A type involves an object and a structure, each with a fingerprint (*fpo* and *fps*), as illustrated by the following example:

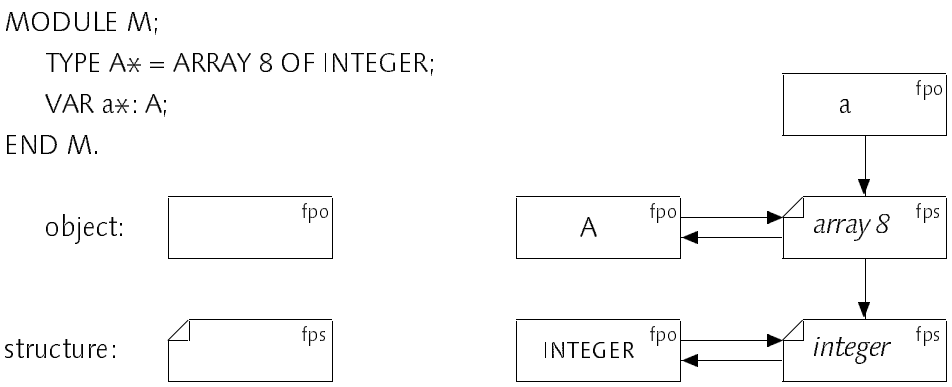

Figure 6.3 Example of a type declaration and associated fingerprints

Fingerprints of predefined types and objects are predefined constants. Fingerprints of user-defined types and objects in figure 6.3 are computed as follows:

```
fps(A) := "M.A" ⊕ Array ⊕ 8 ⊕ fps(INTEGER);
fpo(A) := Type ⊕ fps(A);
fpo(a) := Var ⊕ fps(A);
```

The symbol ⊕ represents the fingerprinting operator. The fingerprint of the structure of the type *A* depends on the module name and canonical name of the structure (*"M.A"*), on the fact that *A* is an array (*Array*, *Type*, and *Var* are small integers), on the number of elements (8), and finally, on the fingerprint of the element structure (the fingerprint of the structure *INTEGER*).

The fingerprint of a structure depends on the canonical name of the structure, because the canonical name is an attribute of the type. Also, the module name is part of the fingerprint because structures can be reexported, contrary to objects, whose fingerprint neither contains the object name, nor the module name, because objects cannot be reexported. Since exported objects are not dependent on other objects, but only on structures, the fingerprint of an exported object is never used to compute another fingerprint (record fields, methods, and parameters do not have their own fingerprint since they are not stand-alone exported objects). Therefore, the name of an object is not part of the fingerprint, but is explicitly listed with the fingerprint in object files.

The *value* of an imported constant object (not the object itself) may be reexported by a newly declared constant (alias). In that case, the fingerprint of the new constant does not depend on the fingerprint of the imported constant, but on its value only. A consistent constant value is guaranteed at link time, since the fingerprint of the imported constant is checked, when the reexporting module is linked.

Addresses of variables, entry numbers of procedures, and addresses of type descriptors do not appear as arguments of the fingerprinting function, because they are context-dependent attributes of objects. Therefore, the linker would not be able to detect an inconsistent use of these attributes by client modules, since the values of these attributes are not part of the fingerprints. As a consequence, these attributes cannot be used at compile time by client modules and hence cannot be listed in the symbol file. Without history of development, it is impossible to guarantee that inserting a new object in an interface will not modify addresses or entry numbers of existing objects. Therefore, the address of an external object is not inserted in the client code at compile time, but at link time. The address is present in the object file of the exporting module only. This requires a fix-up chain for each external object in client modules. External objects are then linked by name (see the section on the implementation).

Fingerprinting Recursive Types

Clearly, computing the fingerprint of a type involves a bottom-up traversal of the type tree. Unfortunately, a type is not always represented by a tree, but a more general graph is sometimes necessary. Consider the following type declaration and the corresponding attempt for a fingerprint computation (for better readability, the module name *M* is not shown in the expressions):

```
TYPE
   Ptr* = POINTER TO Desc;
   Desc* = RECORD next*: Ptr END ;

fps(Ptr) := "Ptr" ⊕ Pointer ⊕ fps(Desc);
fps(Desc) := "Desc" ⊕ Record ⊕ "next" ⊕ offset(next) ⊕ fps(Ptr);
fpo(Ptr) := Type ⊕ fps(Ptr);
fpo(Desc) := Type ⊕ fps(Desc);
```

Obviously, this is a recursive type declaration: *fps(Ptr)* depends on *fps(Desc)*, and vice-versa. The cycle has to be broken to compute the fingerprints. Considering that the computation involves a recursive depth-first traversal of the type graph, a fingerprint value may be required, when it is not completely computed at that time, because the corresponding node may belong to a cycle being traversed. A simple solution would be to use the partially computed value since the final one is not available yet. This would yield the following expressions, where the first one denotes a temporary value corrected in the third line:

```
fps(Ptr) := "Ptr" ⊕ Pointer;
fps(Desc) := "Desc" ⊕ Record ⊕ "next" ⊕ offset(next) ⊕ fps(Ptr);
fps(Ptr) := fps(Ptr) ⊕ fps(Desc);
```

The problem with this solution is that starting the computation with *Desc* instead of *Ptr* yields different expressions and hence different fingerprint values:

```
fps(Desc) := "Desc" ⊕ Record ⊕ "next" ⊕ offset(next);
fps(Ptr) := "Ptr" ⊕ Pointer ⊕ fps(Desc);
fps(Desc) := fps(Desc) ⊕ fps(Ptr);
```

Without using history of development, it is difficult to guarantee a constant evaluation order. Both alphabetical order and order of declaration may be perturbed by the insertion of new objects. In the example below, inserting the new type *A* in front of the declarations of *B* and *C* modifies both orders:

```
TYPE
   A* = POINTER TO C;
   B* = POINTER TO C;
   C* = RECORD z*: B END ;
```

Prior to the insertion of *A*, the fingerprint computation started by visiting *B*, then *C*, whether alphabetical order or declaration order was used. Now, *A* is visited first in both cases, and *C* is visited before *B*, because *A* depends first on *C*. Therefore, by using any of these orders, the fingerprint values for *B* and *C* would depend on the presence of *A*, which would be incorrect, because neither *B* nor *C* depends on *A*.

Actually, *B* and *C* form a strongly connected component of the type graph. A correct solution is to locate strongly connected components and to compute fingerprints of the nodes of these components first. A canonical node must be determined, so that the computation can always start at the same node. A new inserted type cannot alter the fingerprint values, except if the new type is a node of the strongly connected type graph. In that case, it is correct to modify the fingerprints, because this is a modification of the types in the graph.

Declaration order is inadequate to determine the canonical node of the strongly connected component, because swapping two declarations in the source text should have no effect on fingerprint values. Alphabetical order is a good choice; the node with the "smaller" name is the canonical representative. Applying this technique to the previous example results in the following statement sequence for computing the fingerprints of the structures in the strongly connected graph (the structure *A* does not belong to it):

```
fps(B) := "B" ⊕ Pointer;
fps(C) := "C" ⊕ Record ⊕ "z" ⊕ offset(z) ⊕ fps(B);
fps(B) := fps(B) ⊕ fps(C);
```

The node *B* is the canonical representative of the graph. The fingerprint of *C* uses a temporary value of the fingerprint of *B*. Object fingerprints are not shown; the computation of their fingerprint is never problematic, because objects never belong to cycles. The execution of these statements yields the following results:

```
fps(B) = "B" ⊕ Pointer ⊕ ("C" ⊕ Record ⊕ "z" ⊕ offset(z) ⊕ ("B" ⊕ Pointer));
fps(C) = "C" ⊕ Record ⊕ "z" ⊕ offset(z) ⊕ ("B" ⊕ Pointer);
```

There is nevertheless a problem with this technique: each fingerprint in a cycle should contain the complete type information of the cycle. Testing any fingerprint of the cycle at link time should simultaneously verify all types in the cycle.

Unfortunately, this is not the case. For example, the fingerprint of *C* above does not "know" that *B* is a pointer to *C*. It just knows that *B* is a pointer, because a temporary value for the fingerprint of *B* was used to compute the fingerprint of *C*. Breaking the cycle also cuts this information.

Although every fingerprint in a strongly connected component of a type graph does not contain the complete type information of the component, evidently, the sum of them does. This sum is a kind of global fingerprint of the component. Therefore, if this global fingerprint is added back to each fingerprint, then each of them contains the complete type information. This solution requires three traversals of the type graph:

1. Traverse the type graph to find strongly connected components and their canonical representatives; for each found component, execute steps 2 and 3.

2. Traverse the strongly connected component starting from its canonical representative to compute fingerprints, and combine them to form a global fingerprint.

3. Traverse the strongly connected component to combine the global fingerprint with each fingerprint.

This solution is too expensive and hence not acceptable. The following remark will help finding a more efficient solution: cycles made accessible to the outside by one node only cannot have more than one evaluation order for the fingerprint computation. In contrast, a cycle with several entry nodes can have different evaluation sequences for the same fingerprint. In other words, a different fingerprint value is obtained when the computation enters a cycle at a different node.

An entry node is always a named type that can be referenced by other types. The previous example had two entry nodes (*B* and *C*), but the following declaration has only one:

```
TYPE
  Desc* = RECORD
    handler*: PROCEDURE(VAR par: Desc; msg: INTEGER);
    next*: POINTER TO Desc
  END ;
```

Although the type graph is cyclic, the evaluation order is always the same, because the computation can only start at the node *Desc* (record structure). The partially computed fingerprint value for *Desc* used for fingerprinting the signature of *handler* and for fingerprinting the type of *next* does not contain the complete type information of the cycle, because the computation is not concluded yet. However, this is irrelevant, since the fingerprints of these field types will never be used by nodes other than *Desc*. Other nodes can only use the final fingerprint of *Desc* (which contains the complete type information), because *Desc* is the only accessible node from outside the cycle. Therefore, record field objects do not need fingerprints.

As a conclusion, the fingerprinting of a type graph does not pose any problems if each strongly connected component of the graph involves only one named type.

Breaking Cycles

The question now is: when, exactly, may a strongly connected component contain more than one named type? The scope rules of the Oberon language [6], more precisely the amendments of them, give the answer:

1. If a type *T* is defined as POINTER TO *T1*, then the identifier *T1* can be declared textually following the declaration of *T*, but it must lie within the same scope.

2. Field identifiers of a record declaration are valid in field designators only.

The second point confirms that record fields do not need fingerprints. The first amendment above is the only exception to the rule stating that identifiers must be declared before being used. Therefore, the only possibility for a strongly connected component of a type graph to describe more than one named type is to include at least one named pointer to a forward-declared named type, as demonstrated here:

- Type declarations introducing a new type name *T* cannot be nested. They all have the form *T* = ...;
- A strongly connected type graph declaring more than one type name involves several such type declarations; one of them appears first in the source text.

- Since the type graph is strongly connected, type declarations in the graph must directly or indirectly depend on each other.
- The first declaration must depend on a following declaration (cyclic module imports are not allowed); according to the Oberon scope rules, this declaration can only be a named pointer to a named type.

As explained before, a single recursive depth-first traversal of a type graph is capable of correctly computing all fingerprints of the graph nodes, only if each strongly connected component of the graph has at most one entry point. A correct fingerprint for a node *T* must contain the type information of *T* and the type information of every node that is reachable from *T*. Also, its value must not depend on the starting point of the computation. Now, strongly connected components with more than one entry point always contain a named pointer to a named type. Interrupting the recursion of the fingerprint computation each time a named pointer to a named type is encountered ensures that each strongly connected component with more than one entry point is decomposed into subgraphs consisting of strongly connected components with a single point of entry.

For example, the following type declarations constitute a strongly connected component with four entry points, namely *A*, *ADesc*, *B*, and *BDesc*:

```
A* = POINTER TO ADesc;
B* = POINTER TO BDesc;
ADesc* = RECORD b*: B END ;
BDesc* = RECORD
   a*: A;
   p*: PROCEDURE(VAR b: BDesc)
END ;
```

Figure 6.4 shows the connectivity of the corresponding type graph. If the cycle is broken between a named pointer and its named base type (dashed lines), the strongly connected component splits into four strongly connected components with one entry point each (rectangle).

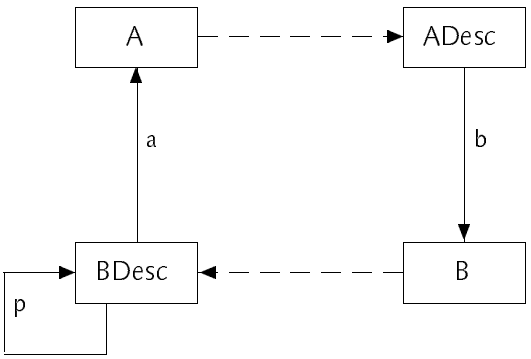

Figure 6.4 Breaking a cyclic type graph

Similarly, the recursion in the fingerprint computation can be interrupted if the fingerprint of a named pointer type does not depend on the fingerprint of the named pointer base type, but only on the *name* of the base type. Figure 6.5 illustrates the remaining dependences in the fingerprint computation.

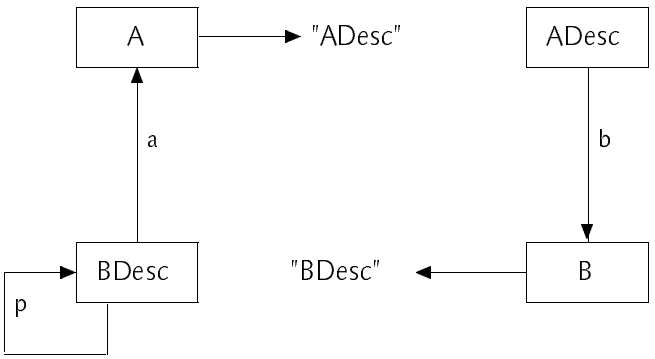

Figure 6.5 Fingerprinting recursive types

The strongly connected component containing the type *BDesc* is still cyclic, but it has only one entry point. Therefore, the computation of the fingerprint of *BDesc* does not pose any problems, as shown below:

```
fps(A) := "A" ⊕ Pointer ⊕ "ADesc";
fps(B) := "B" ⊕ Pointer ⊕ "BDesc";

fps(ADesc) := "ADesc" ⊕ Record;
fps(ADesc) := fps(ADesc) ⊕ "b" ⊕ offset(b) ⊕ fps(B);

fps(BDesc) := "BDesc" ⊕ Record;
fps(BDesc) := fps(BDesc) ⊕ "a" ⊕ offset(a) ⊕ fps(A) ⊕
   "p" ⊕ offset(p) ⊕ Proc ⊕ Varpar ⊕ fps(BDesc);
```

All fingerprints can be computed in one traversal of the type graph. If the value of a fingerprint is needed during its own computation, as for *fps(BDesc)* above, an initial value is used instead. This initial value contains the name and the form of the type, but it could be any special value meaning *self*, since an initial value is never used by another fingerprint computation, except by its own one. Indeed, there is only one entry point in a recursive computation.

This drastic simplification of the fingerprint computation has a serious drawback: consistency checking at link time can only guarantee that the base type of the imported pointer has the expected name, but it cannot verify that the structure of this base type has not changed. However, it is interesting to note that this verification is sufficient if the pointer is only assigned in client modules, but never dereferenced. Indeed, the memory integrity of the system of modules is preserved after a pointer assignment on one condition: that the assigned pointer is assignment-compatible with the destination variable. The client executing the assignment does not need to know exactly what the pointer is pointing at (opaque pointer types in Modula-2 make use of this property).

In Oberon, pointer assignment-compatibility rules are tightly coupled with the concept of record extension. A pointer can be assigned to a variable if the pointer type *extends* the type of the variable. Therefore, the fingerprint of the pointer must not only contain the name of the pointer base type, but also the names of all record types extended by the pointer base type. Otherwise, assignment incompatibility could not always be detected:

```
TYPE
   BT* = POINTER TO BTDesc;
   BTDesc* = RECORD ... END ;

   T* = POINTER TO TDesc;
   TDesc* = RECORD (BTDesc) next: T END ;

VAR
   bt*: BT;
   t*: T;
```

A client importing the module containing these declarations is allowed to assign *t* to *bt*, since *T* extends *BT*. In that case, the linker only checks the fingerprints of *t* and of *bt*, which depend on the fingerprints of *T* and of *BT* respectively. Now, if the declaration of *TDesc* is modified and does not extend *BTDesc* any longer, *t* is no more compatible with *bt*, and the client should be invalidated. If the fingerprint of *T* does not also include the name of the

extended record *BTDesc*, the inconsistency cannot be detected. So, the fingerprints of the variables *bt* and *t* must be the following:

```
fpo(bt) = Var ⊕ "M.BT" ⊕ Pointer ⊕ "M.BTDesc";
fpo(t) = Var ⊕ "M.T" ⊕ Pointer ⊕ "M.BTDesc" ⊕ "M.TDesc";
```

Of course, if a client dereferences a pointer, then the structure of the pointer base type must be consistent. In that case, both the fingerprints of the pointer and of the pointer base type are verified. Besides marking imported objects whose names are used in the source text, the compiler also marks the named base type of every dereferenced, named pointer. The fingerprints of marked objects are listed in the object file and are compared at link time to the fingerprints of the corresponding objects supplied by the exporter module.

Fingerprinting Signatures

Until now, the analysis of strongly connected components declaring more than one named type has only considered the original Oberon language. In the Oberon-2 version of the report [11], the second amendment to the scope rules, which is described above, also includes method identifiers besides field identifiers. This confirms that neither record fields nor methods need fingerprints.

In Oberon, every component of a named type is textually declared between the equal sign following the name of the type and the semicolon terminating the declaration. This is not the case in Oberon-2, since methods are procedures bound to a record type from outside the record declaration. This means that a record type has an implicit forward reference to its methods. This observation modifies the conclusion that a strongly connected component must include a named pointer to a named type in order to declare several named types. Indeed, a type-bound procedure may strongly connect two components of a type graph declaring a type name each:

```
TYPE
   T1* = RECORD ... END ;
   T2* = RECORD t: T1; ... END ;

PROCEDURE (VAR self: T1) M* (VAR t: T2); ... END M;
```

When fingerprinting the signature of the method *M* above, the fingerprint of the parameter type *T2* should not be computed, otherwise the fingerprint values become dependent on the starting point of the computation. The fingerprint of

the signature has nevertheless to reflect the type of its formal parameters so that a modification of the signature can be detected at link time.

It is interesting to note that memory integrity is preserved if the fingerprint of a procedure only depends on the type name and mode of the formal parameters, as well as on the name of the result type. Indeed, the complete structure of the formal parameter types and of the result type does not need to be included in the fingerprint of the procedure, because this structure is checked in the client module calling the (type-bound) procedure. For convenience, procedures and type-bound procedures are fingerprinted using the same algorithm.

The following example shows a module *B* exporting a procedure *P* whose formal parameter type *T* is imported from *A*, and two client modules *C* and *D* calling *P*:

```
MODULE A;
   TYPE T* = ... ;
   VAR t*: T;
END A.

MODULE B;
   IMPORT A;
   PROCEDURE P*(t: A.T); ... END P;
END B.   (* the fingerprint of A.T is checked *)

MODULE C;
   IMPORT A, B;
BEGIN
   B.P(A.t)
END C.   (* the fingerprints of B.P and of A.t are checked *)

MODULE D;
   IMPORT A, B;
   VAR t: A.T;
BEGIN
   B.P(t)
END C.   (* the fingerprints of B.P and of A.T are checked *)
```

Although the fingerprint of *B.P* does not reflect modifications in the structure of *A.T*, it is impossible to call *B.P* from an inconsistent client module. Indeed, if *A.T* is modified, its fingerprint and the fingerprint of the variable *A.t* are modified. Therefore, module *B* cannot be linked without being recompiled, since the fingerprint of *A.T* is checked (when a client module uses an imported identifier, the linker has to check the corresponding fingerprint). Note that the recom-

pilation of *B* does not result in a new fingerprint for *B.P*, since this one depends only on the name *"A.T"*.

The client modules *C* and *D* have to declare or to import a variable of type *A.T* in order to call the procedure *B.P*. Thereby, the fingerprint of *A.T* or of a variable of type *A.T* is checked and hence an inconsistency is detected if either *C* or *D* is not recompiled. On the other hand, if *A.T* is not modified, but the signature of *B.P* is (a new formal parameter type replaces *A.T*, for example), the inconsistency is detected by checking the fingerprint of *B.P*.

In other words, the fingerprint of a (type-bound) procedure depends only on the names occurring in its signature. It does not have to reflect the complete structure of the formal parameter types, since this is done when declaring or calling the procedure. The fingerprint of the signature only guarantees that the signature has not been modified textually, and that the type checking of actual and formal parameters performed by the compiler is still valid at link time.

Note that the name and the offset of a formal parameter are not fingerprinted. The name is present in the signature for documentation purpose only and can be modified without invalidating clients of the signature. The offset of a formal parameter cannot be inconsistent, because the order of the formal parameters is included in the signature fingerprint and the complete structure of every formal parameter type is verified as explained above.

Obviously, the name of a formal parameter type cannot be used in the fingerprint computation if the type is anonymous. The fingerprint has nevertheless to guarantee the validity of the type checking between actual and formal parameters performed at compile time. For example, a formal open array of some element type accepts any open or fixed-size array of the same element type as actual parameter. In this case, the fingerprint of the signature has to include the form of the type (open array) and the name of the element type. Also, a formal fixed-size or open array of characters accepts a constant string as actual parameter. In contrast, anonymous records do not make sense as formal parameters, since name equivalence in Oberon prohibits any assignment compatibility and thereby makes it impossible to call the procedure. The fingerprint computation for a signature exactly reflects the Oberon rules for parameter passing.

Fingerprint Computation

To resume, fingerprinting recursive types does not pose any problems if the recursion is broken at each occurrence of a named pointer type to a named base type, and at each occurrence of a (type-bound) procedure signature. The

fingerprint of such a pointer type and of a signature then guarantees that the declaration has not changed textually, but does not represent the complete type information. A complete type checking is performed by verifying the fingerprint of the base type when the pointer type is dereferenced, and by verifying the fingerprint of the parameters when a procedure is declared and called.

A fingerprint that contains the complete type information also guarantees the validity of the textual declaration. Therefore, the fingerprint of any type can be computed in two steps: first, the part containing information about the textual declaration only, and then the part containing the remaining type information. The first part is called the *identifier fingerprint*, since it mainly relies on the type identifier. The final fingerprint is a combination of both parts, except for procedure signatures and named pointer types to named base types, for which the identifier fingerprint is also the final fingerprint.

Table 6.1 describes the computation of the identifier fingerprint (denoted *idfp*) of any user-defined structures. Each predefined type has a different identifier fingerprint, which is a predefined constant (not shown in table 6.1).

Table 6.1 Computation of identifier fingerprints

Structure S	idfp(S)
POINTER TO T	name ⊕ Pointer ⊕ idfp(T)
RECORD ... END	name ⊕ Record
RECORD (T) ... END	name ⊕ Record ⊕ idfp(T)
ARRAY OF T	name ⊕ DynArr ⊕ idfp(T)
ARRAY n OF T	name ⊕ Array ⊕ n ⊕ idfp(T)
PROC (a_0: T_0; ...; VAR a_{n-1}: T_{n-1}): T	name ⊕ Proc ⊕ Valpar ⊕ idfp(T_0) ⊕ ... ... ⊕ Varpar ⊕ idfp(T_{n-1}) ⊕ idfp(T)

The name of the structure and the name of the module declaring the structure are included in the identifier fingerprint (the concatenation of both is denoted *name* in the table). However, if a structure is anonymous, no name is considered by the computation.

The identifier fingerprint of a pointer type also contains the identifier fingerprint of its base type. The identifier fingerprint of an extending record type also contains the identifier fingerprint of the record type it extends. The identifier fingerprint of a signature also contains the identifier fingerprint of its formal parameter types. In other words, the type name is not always sufficient to guarantee consistency, because Oberon uses structural type equivalence

instead of type name equivalence in some cases like procedure assignment, parameter passing to formal open arrays, as well as pointer and record assignment.

The identifier fingerprint of an array must include the identifier fingerprint of the element type, because a client module can pass a string as parameter to a formal array of characters; the final fingerprint of the array type is not checked in that case, but only the fingerprint of the procedure, which is independent of the final fingerprint of the array type.

The final fingerprint of a structure depends on the identifier fingerprint of that structure, as shown in table 6.2.

Table 6.2 Computation of structure fingerprints

Structure S	fp(S)
(named) POINTER TO (named) T	idfp(S)
POINTER TO T	idfp(S) ⊕ fp(T)
RECORD ...	idfp(S) ⊕ size ⊕ align ⊕ nofmeth ⊕ ...
fld*: T; ...	"fld" ⊕ offset ⊕ fp(T) ⊕ ...
meth* (...) PROC (...)	"meth" ⊕ methno ⊕ fp(signature)
END	
RECORD (T) ... END	idfp(S) ⊕ fp(T) ⊕ ... (like above)
ARRAY OF T	idfp(S) ⊕ fp(T)
ARRAY n OF T	idfp(S) ⊕ fp(T)
PROC (a_0: T_0; ...; VAR a_{n-1}: T_{n-1}): T	idfp(S)

Depending on the implementation, further attributes may also be included in the fingerprint computation: the offset of hidden pointer fields and/or of procedure fields, the method number of hidden methods, and the value of the system flag (*sysflag*, see chapter 4), for example.

The fingerprint of an object depends on the mode of the object and on the fingerprint of its type, but it does not include the object name because objects are linked by name. Table 6.3 shows the different object modes and the corresponding fingerprint computation.

Table 6.3 Computation of object fingerprints

Object x	fp(x)
CONST x* = value of type T	Const ⊕ form(T) ⊕ value
TYPE x* = T	Type ⊕ fp(T)
VAR x*: T	Var ⊕ Readwrite ⊕ fp(T)
VAR x–: T	Var ⊕ Readonly ⊕ fp(T)
PROC x*(...)	XProc ⊕ fp(signature)
PROC+ x*(...)	IProc ⊕ fp(signature)
PROC– x*(...) c_0, ... c_{n-1}	CProc ⊕ fp(signature) ⊕ n ⊕ c_0 ⊕ ... ⊕ c_{n-1}

XProc denotes a conventional exported procedure, *IProc* an interrupt procedure (which may have different calling conventions), and *CProc* a code procedure (inline procedure which is used for hardware interfacing purposes). Note that no distinction is made between a type object and its alias object. Both have therefore the same fingerprint and a client may use either of the two names. However, swapping the alias name with the canonical name in the type and alias declarations results in a new fingerprint, since the canonical name is included in the fingerprint of the structure.

Anonymous Types and Name Equivalence

Remember that the fingerprint of a recursive type defined as pointer to a base type has to include the name of the pointer type as well as the *name* of the base type, in order to break the cycle in the fingerprint computation. However, this is not possible if the base type is anonymous. In that case, the fingerprint of the pointer type depends on the *fingerprint* of its base type and includes the complete type information. In contrast, the fingerprint of the base type does not contain the complete information, but it is only used to compute the fingerprint of the pointer type:

```
TYPE
   T* = POINTER TO RECORD next*: T END ;

fps(T) = "M.T" ⊕ Pointer ⊕ Record ⊕ ...
   ... ⊕ "next" ⊕ offset(next) ⊕ ("M.T" ⊕ Pointer ⊕ Record);
```

Since there is only one named type in this graph, there is a single entry point; so, the fingerprint computation is not problematic, as explained in the preceding sections.

Anonymous types nevertheless pose a problem. Consider the following module exporting two variables of an anonymous type:

```
MODULE M;
  VAR
    p*, q*: POINTER TO RECORD ... END ;
END M.
```

A client assigning *M.p* to *M.q* has to list the fingerprints of *p* and *q* in its object file. These two fingerprints are identical, since both variables are of the same type, and since the fingerprint of an object does not include the object name. However, this is not the problem. Consider now the new version of module *M*:

```
MODULE M;
  VAR
    p*: POINTER TO RECORD ... END ;
    q*: POINTER TO RECORD ... END ;
END M.
```

The assignment of *M.p* to *M.q* is not allowed any longer, because the variables are of a different type, although the type structures are identical. Indeed, Oberon does not use structural equivalence, but name equivalence. The problem is that the modification has no effect on the fingerprint values and that the client is therefore not invalidated. Remember that the fingerprint has to be context-independent. It is inherently impossible to reflect this kind of modification in the fingerprint (which must be context-independent), since everything but the context remains unchanged.

At first sight, this seems to be disastrous and to cast doubts on the correctness of the proposed model. A more careful examination reveals that the problem is harmless. First, exported objects of an anonymous type are extremely rare. Second, the depicted interface modification would denote a dubious programming style and makes this scenario still more improbable. However, the compiler neither has to judge the quality of the submitted programs nor to rely on improbabilities.

If the incompatible assignment is really executed, the memory integrity of the system is not endangered (a language favoring structural type equivalence would admit the assignment). A program working before the modification will still work after it. One could fear a different behavior of type tests implied by assignments of dereferenced pointers in the exporting module, after an illegal

pointer assignment performed by an noninvalidated client; but the compiler does not generate implied type tests for anonymous types, because an anonymous type cannot be extended.

So, in some extremely rare cases, the object model may apply structural equivalence for anonymous types used over module boundaries. However, if the old symbol file is available when the modified interface is recompiled, which is normally the case, the compiler signals the interface change (the implementation is described later in this chapter).

A Fingerprint per Object Component

The granularity of type checking in the original linker is rather coarse: a single key guarantees the consistency of an entire interface. Therefore, any interface modification always results in a new key and clients of the interface are invalidated. The layer model improves the situation by prescribing a fingerprint for each layer of an interface, and the object model by a fingerprint for each object of an interface. One wonders whether still finer-grained checks are desirable and possible. Would a fingerprint for each object component be both practical and efficient?

Object components are formal procedure parameters, record fields, and type-bound procedures. Modifying the number of formal parameters in an exported procedure without invalidating clients does not make much sense. First, a mechanism for default parameters as well as for superfluous parameters (questionable idea!?) would have to be introduced in the language. Second, it would be difficult to distinguish between an interface modification with no desirable client invalidation and a modification with required invalidation. Furthermore, the consistency check would be more expensive, since as many fingerprints as parameters would be checked for each imported procedure.

Inserting and removing exported record fields or type-bound procedures pose similar problems. As explained in the preceding chapter, a name collision might occur between a newly inserted field and a field in a record extending the modified record. Here too, the language and in particular its scope rules would have to be modified to permit such extensions.

A component is tightly bound to its object, much more than an object is bound to the module interface it belongs to. For example, there is a well-defined order among object components, which is not the case among objects of a same module interface. So, any externally visible change to an object should be considered as an invalidating modification.

In contrast, it would be interesting to be able to modify the hidden part of an object. Similarly to the reimplementation of a procedure body that does not invalidate clients, the modification of the internal implementation of a record type is desirable. For example, inserting or deleting hidden fields (at least at the end of the record) should not invalidate clients of the record.

At first sight, this should be already possible now since implementation-specific and hidden attributes neither appear in the symbol file, nor in the fingerprint computation. A more careful attention reveals that this is not true, since the record size, hidden pointer fields, hidden procedure fields, and hidden methods may be used by some clients (to build type descriptors or method tables, for example) and are therefore included in the symbol file and in the fingerprint value.

The point is that these attributes are only used under special circumstances. A client accessing record fields through a pointer will not need the record size, for example. On the other hand, a client statically allocating a record, or copying a record will. It would be possible to classify the clients of a record in different categories, depending on the use the client makes of the record. A record may be accessed through a pointer, statically allocated, dynamically allocated, extended, copied, and so on. Unfortunately, each category would require a fingerprint, thereby yielding larger object files. The additional complexity in both the compiler and the linker would be too important in comparison with the relatively small gain in flexibility.

The object model makes a distinction between two classes of record clients only: on the one hand, clients using the *public* information about the record (like exported fields), on the other hand, the clients using the *private* information about the record (like hidden fields or record size). Note that the membership to one of these classes also depends on the system's implementation. For example, some implementation might determine the record size at run time before copying a record.

Each record has two fingerprints: a public one and a private one. Both fingerprints are listed in the object file exporting the record type, but only one of them is listed in the object file of a client importing this record type. If a client just accesses fields of the imported record, it then lists the public fingerprint of this record. Only this fingerprint will be checked by the linker. That means that a modification of the private information of the record, like the insertion of new hidden fields at the end of the record, cannot invalidate this client. In contrast, such a modification will generate a new private fingerprint and will thereby invalidate clients that declare a variable of this record type and therefore list the private fingerprint in their object file. Evidently, the modification of the public fingerprint of a record, caused by the insertion of a new *exported* field for

example, invalidates all clients of this record. The private fingerprint has therefore to depend on the public fingerprint.

Both public and private fingerprints of a record type are bound to the type structure, but not to the type object. The object itself has a single fingerprint, which depends on the public fingerprint of the structure only. The object fingerprint is only listed in the object file if the type name is exported, but both structure fingerprints are listed if the type appears in the symbol file, as shown in the following example:

```
MODULE M;
   TYPE
      Desc = RECORD ... END ;
   VAR
      p*, q*: POINTER TO Desc;
END M.

MODULE N;
   IMPORT M;
BEGIN
   NEW(M.p)
END N.
```

The module *M* lists four fingerprints in its object file: both public and private fingerprints for the structure *Desc* (which is exported through the exported variable *p* and *q*), as well as the fingerprints of the variables *p* and *q* that depend on the public fingerprint of *Desc*. However, the fingerprint of the object *Desc* is not listed, because the name *Desc* is not exported. The client *N* lists the fingerprint of *p* only in its object file. In this example, the linker will not verify that the private structure of *Desc* has not changed; it will only check the fingerprint of *p*, which does not depend on the private fingerprint of *Desc*. This is correct, because the client *N* does not use the private structure of *Desc*: the required information for the dynamic allocation is obtained at run time from the type descriptor and is therefore always consistent.

Consider now a new version of the client *N*:

```
MODULE N;
   IMPORT M;
BEGIN
   M.p↑ := M.q↑
END N.
```

The client performs a record assignment and hence needs to know the record size, which is found in the symbol file of *M*. The linker must verify that the record size is consistent when *N* is linked to *M*. Therefore, this version of *N* lists the private fingerprint of *Desc* along with the fingerprints of *p* and *q*. Note that an Oberon implementation taking the size at run time from the type descriptor of *Desc* would not require a verification of the private fingerprint of *Desc* at link time.

One observes that fingerprints of exported structures whose name is not exported may nevertheless be needed, if these structures can be the base type of a pointer type. This is the case for array, dynamic array, and record structures. Therefore, the private and public fingerprints of such exported structures are always listed in the object file, even if their name is not exported. In contrast, fingerprints of objects are only listed if the object's name is exported. Actually, an object cannot appear in a symbol file if its name is not exported, contrary to a structure.

Two fingerprints per record type is the right balance between the two extremes, namely, a single fingerprint per module interface or a fingerprint per object component. One can conclude that the object model is a trade-off between both simplicity of implementation and efficiency of consistency checking on the one hand, and flexibility in module extension on the other hand.

The Implementation

Similarly to the layer model, the object model has been implemented in the portable Oberon-2 Compiler OP2. However, the object model needed more editing changes in OP2 than the layer model, partly because of the new object file format. On the other hand, the storage allocation in the object model is much simpler than in the layer model and even simpler than in the original OP2, because the order in which objects are allocated is not relevant. Indeed, context-dependent attributes are not used over module boundaries. So, different values due to a different allocation order are not visible to the outside and hence cannot invalidate clients.

Symbol File Format

Basically, the object model employs the same symbol file format as the layer model. However, some context-dependent attributes have been eliminated from

the symbol file. This section enumerates the differences between the formats (the complete file format is described in the appendix).

In the object model, the symbol file is not built as a stack of layers separated by stoppers and containing objects, but as a single list of objects:

```
SymFile  =  0FBX Module {Object}.
```

The new one-byte file tag indicates a different file format. The layer model stores the fingerprint of each layer in the file. The usually small number of layers requires a small amount of disk space for the fingerprints. This is different in the object model, which requires one fingerprint for each object and two fingerprints for each named record or array structure. So, the fingerprints are recomputed when reading the symbol file and are not stored in the file. The time gained in reading more compact files compensates for the time spent in recomputing fingerprints. Writing the fingerprints into the symbol file would result in a file size increase of 20% in average.

The module specifier consists of the module name only, or a negative number referencing an already listed name, but it includes neither layer numbers nor layer fingerprints:

```
Module  =  0 | negmno | MNAME name.
```

The first module specifier in the file (after the file tag) lists the module name of the interface described by the file (the own module name). The number *0* refers then to the own module. Other module names and negative numbers refer to modules that are imported by the interface.

Variables have no offsets and (type-bound) procedures have no entry numbers, since these attributes are context-dependent:

```
Object   =  ...
          |  (RVAR | VAR) Struct name
          |  (XPRO | IPRO) Signature name
          |  ...

Method   =  (TPRO Signature name | HDTPRO) methno.
```

Offsets of variables are present in the object file and are used by the linking loader to compute absolute addresses, which are inserted in the code of client modules as well as in the exporting module (unless pc-relative addressing is used in the exporting module). Procedure entry numbers disappear completely, since absolute addresses can also be computed by the linking loader from the absolute code position in memory and from procedure offsets, which are stored

in the object file. The linker finds imported variables and procedures by their name.

The address of the descriptor of each record type is not listed any longer as an attribute of the record structure in the symbol file. Type descriptors are linked using the name of the type they describe. If this type is anonymous, its private fingerprint is used instead of its name.

```
Struct   =    negref
            | STRUCT Module name [SYS value]
              (  ...
              |  REC Struct size align nofMeth {Field} {Method} END
              |  ... ).
```

The type descriptor is a data structure allocated at run time that contains information about each record type. Type descriptors contain the hierarchy of type extensions, which is used for type tests at run time. They also include a method table dispatching the calls to type-bound procedures, as well as a pointer offset table accessed during garbage collection to locate pointer fields in records and in array elements.

From an implementation point of view, a type descriptor address is actually the address of a global variable initialized at load time to point to a type descriptor. In the previous models, a type descriptor is always indirectly accessed through this pointer variable. The object model, which requires one fix-up chain for each object, links every use of a type descriptor into one fix-up chain. This allows the absolute address of the type descriptor to be directly inserted in the code.

The elimination of these global pointers represents a gain in both memory space and execution speed of type tests: the type tag to be checked is compared to a constant value (immediate addressing mode) instead of being compared to the contents of a global variable (memory access). Note that this optimization is not possible if a compacting garbage collector is used, or else type descriptors have to be allocated outside the heap.

As explained in the preceding chapter, to each production of the symbol file grammar correspond both an externalizing and an internalizing routine. Since the format is almost the same in both models, the routines are very similar and hence not listed here.

In the layer model, fingerprints of imported layers are not computed, but read from the symbol file of imported modules, whereas fingerprints of exported layers are computed by the externalizing routines. This is different in the object model. First, the fingerprints of imported structures and objects cannot be read from imported symbol files, because symbol files do not contain

fingerprints. So, they are computed when these symbol files are internalized. Second, the externalizing routines do not need to compute fingerprints for exported items, since fingerprints are not written into the symbol file. However, fingerprints for exported items have to be listed in the object file. So, the fingerprinting routines that are called for imported items by internalizing routines are also called for exported items by the routine generating the object file. Consequently, these fingerprinting routines, which are presented in the following sections, are stand-alone procedures separate from the externalizing and internalizing routines.

Fingerprinting Structures

Remember that structure nodes need (public and private) fingerprints for two reasons. First, fingerprints of structure nodes may serve as common subexpression in the computation of the object fingerprints. Indeed, different objects may be of the same type. In that case, their fingerprint depends on the fingerprints of the common type structure. It would be a waste of time to compute these fingerprints several times. Second, the fingerprint of a named pointer type does not depend on the structure of its base type if this base type is named too (possible cycles are broken in this context). Dereferencing a variable of such a pointer type then requires the verification at link time of the fingerprints of the base structure.

The cost of determining whether a structure node really needs to recall its fingerprint values (whether the node is the root of a common subgraph, for example) is not negligible. It is much more efficient to store the fingerprint values in each structure node, without distinction.

The identifier fingerprint of a structure may be used several times by the computation of other fingerprints. For example, the identifier fingerprint of a base record is used for computing the identifier fingerprints of all record types extending this base record. Here too, common subexpressions can be eliminated by storing the identifier fingerprint in each structure node.

The data structure *StrDesc* in OP2, which represents nodes of compiled structures, is augmented by new fields holding the three different fingerprint values:

```
StrDesc* = RECORD
   ...
   fpdone, idfpdone: BOOLEAN;
   idfp, pbfp*, pvfp*: LONGINT;
   ...
END ;
```

An identifier fingerprint (*idfp*), a public fingerprint (*pbfp*) and a private fingerprint (*pvfp*) are stored in each structure node. *pbfp* and *pvfp* are exported from OPT, because their value may be written to the object file by the module OPL.

The same node of a cyclic type graph may be reached several times during the fingerprint computation. Therefore, already traversed nodes must be marked to avoid infinite loops. It is not possible to determine whether a fingerprint is already computed by looking at its value, because all 32-bit numbers can be a valid fingerprint value. Consequently, additional boolean fields are both necessary and convenient to mark the node. Public and private fingerprints are computed simultaneously. Accordingly, the completion of their computation is denoted by a single field (*fpdone*). A second boolean field (*idfpdone*) indicates that the identifier fingerprint has been computed.

The same hash function as in the layer model computes the fingerprints in an incremental fashion. The procedure *FPrint* of module OPM implements this fingerprinting function:

```
PROCEDURE FPrint*(VAR fp: LONGINT; val: LONGINT);
BEGIN fp := S.ROT(S.VAL(LONGINT, S.VAL(SET, fp) / S.VAL(SET, val)), 1)
END FPrint;
```

Each attribute the fingerprint has to depend on is passed as parameter to this procedure. For example, a name is fingerprinted by applying the hash function to each character of the name:

```
PROCEDURE FPrintName(VAR fp: LONGINT; VAR name: ARRAY OF CHAR);
   VAR i: INTEGER; ch: CHAR;
BEGIN i := 0;
   REPEAT ch := name[i]; OPM.FPrint(fp, ORD(ch)); INC(i) UNTIL ch = 0X
END FPrintName;
```

The fingerprint of a signature includes the identifier fingerprint of the types occurring in the formal parameter list of the signature. Now, the type of a formal parameter may in turn be a signature. This mutual recursion requires a forward declaration for one of the two procedures computing either identifier fingerprints (procedure *IdFPrint*) or signature fingerprints (procedure *FPrintSign*).

The procedure *FPrintSign* first calls the procedure *IdFPrint* on the function result type of the signature and on each formal parameter type of the signature, in order to compute their respective identifier fingerprint. The final fingerprint for the signature is a combination of these identifier fingerprints (see table 6.1).

```
PROCEDURE ↑IdFPrint*(typ: Struct);

PROCEDURE FPrintSign(VAR fp: LONGINT; result: Struct; par: Object);
BEGIN
   IdFPrint(result); OPM.FPrint(fp, result↑.idfp);
   WHILE par # NIL DO
      OPM.FPrint(fp, par↑.mode); IdFPrint(par↑.typ); OPM.FPrint(fp, par↑.typ↑.idfp);
      par := par↑.link
   END
END FPrintSign;

PROCEDURE IdFPrint*(typ: Struct);
   VAR btyp: Struct; strobj: Object; idfp: LONGINT; f, c: INTEGER;
BEGIN
   IF ~typ↑.idfpdone THEN
      typ↑.idfpdone := TRUE;
      idfp := 0; f := typ↑.form; c := typ↑.comp;
      OPM.FPrint(idfp, f); OPM.FPrint(idfp, c);
      btyp := typ↑.BaseTyp; strobj := typ↑.strobj;
      IF (strobj # NIL) & (strobj↑.name # "") THEN
         FPrintName(idfp, GlbMod[typ↑.mno]↑.name);
         FPrintName(idfp, strobj↑.name)
      END ;
      IF (f = Pointer) OR (c = Record) & (btyp # NIL) OR (c = DynArr) THEN
         IdFPrint(btyp); OPM.FPrint(idfp, btyp↑.idfp)
      ELSIF c = Array THEN
         IdFPrint(btyp); OPM.FPrint(idfp, btyp↑.idfp); OPM.FPrint(idfp, typ↑.n)
      ELSIF f = ProcTyp THEN
         FPrintSign(idfp, btyp, typ↑.link)
      END ;
      typ↑.idfp := idfp
   END
END IdFPrint;
```

The identifier fingerprint of a structure depends on the form of the structure (boolean, integer, array, and so on). If the structure is named, its fingerprint also depends on the canonical name of the structure and on the name of the module defining the structure. The identifier fingerprint of an (open) array depends on the identifier fingerprint of its element type (see the section on fingerprinting signature). An anonymous formal record type is not compatible with any actual type; therefore, its identifier fingerprint does not need to include information about its internal structure.

The flag *idfpdone* is set at the beginning of the procedure, so that recursive type definitions like the following ones do not cause the procedure to loop forever:

```
TYPE
   Proc = PROCEDURE(proc: Proc);
   Ptr = POINTER TO ARRAY OF Ptr;
```

In this case, the recursive use of an identifier fingerprint by its own computation yields the value *0*, which is entirely satisfying.

The procedure *FPrintStr* simultaneously computes the public fingerprint and the private fingerprint of the structure received as parameter:

```
PROCEDURE FPrintStr*(typ: Struct);
   VAR f, c: INTEGER; btyp: Struct; strobj, bstrobj: Object; pbfp, pvfp: LONGINT;
BEGIN
   IF ~typ↑.fpdone THEN
      IdFPrint(typ); pbfp := typ↑.idfp;
      IF typ↑.sysflag # 0 THEN OPM.FPrint(pbfp, typ↑.sysflag) END ;
      pvfp := pbfp; typ↑.pbfp := pbfp; typ↑.pvfp := pvfp;
      (* initial fingerprints may be used recursively *)
      typ↑.fpdone := TRUE;
      f := typ↑.form; c := typ↑.comp; btyp := typ↑.BaseTyp;
      IF f = Pointer THEN
         strobj := typ↑.strobj; bstrobj := btyp↑.strobj;
         IF (strobj = NIL) OR (strobj↑.name = "") OR
            (bstrobj = NIL) OR (bstrobj↑.name = "") THEN
            FPrintStr(btyp); OPM.FPrint(pbfp, btyp↑.pbfp); pvfp := pbfp
         (* else named pointer to named record; use idfp as pbfp and as pvfp *)
         END
      ELSIF f = ProcTyp THEN (* use idfp as pbfp and as pvfp *)
      ELSIF c IN {Array, DynArr} THEN FPrintStr(btyp);
         OPM.FPrint(pbfp, btyp↑.pvfp); pvfp := pbfp
      ELSE (* c = Record *)
         IF btyp # NIL THEN FPrintStr(btyp);
            OPM.FPrint(pbfp, btyp↑.pbfp); OPM.FPrint(pvfp, btyp↑.pvfp)
         END ;
         OPM.FPrint(pvfp, typ↑.size); OPM.FPrint(pvfp, typ↑.align);
         OPM.FPrint(pvfp, typ↑.n);
         nofhdfld := 0; FPrintFlds(typ↑.link, 0, TRUE);
         IF nofhdfld > OPM.MaxHdFld THEN OPM.Mark(225, typ↑.txtpos) END ;
         FPrintTProcs(typ↑.link);
         OPM.FPrint(pvfp, pbfp);    (* checking pvfp must also check pbfp *)
         strobj := typ↑.strobj;
         IF (strobj = NIL) OR (strobj↑.name = "") THEN pbfp := pvfp
            (* pbfp of an anonymous record must contain the complete information *)
         END
      END ;
      typ↑.pbfp := pbfp; typ↑.pvfp := pvfp
   END
END FPrintStr;
```

The initial value of both the public and private fingerprints of a structure is the identifier fingerprint of the structure. Remember that a named structure may refer to itself. In this case, the initial value is used as fingerprint during a recursive computation.

In contrast, a named pointer to a named structure, as well as a procedure type, breaks the recursion, by using the identifier fingerprint as public and private fingerprint (the public and private fingerprints of any pointer type are always equal). Remember that if a named pointer to a named structure is dereferenced, the public fingerprint of the structure will be checked at link time.

The fingerprint of other pointer types depends on the public fingerprint of the referenced structure. If this structure is allocated, copied or extended, its private fingerprint will be checked. However, if this structure is anonymous, the verification is not possible, because the structure cannot be identified. In this case, its public fingerprint also contains its private fingerprint, which is always the case for (open) array types. This dependence is forced for anonymous records. Example:

```
MODULE M;
   TYPE
      Desc = RECORD ... END ;
      P = POINTER TO Desc;
      Q = POINTER TO RECORD ... END ;
   VAR
      p0*, p1*: P;
      q0*, q1*: Q;
BEGIN ...
END M.

MODULE N;
   IMPORT M;
BEGIN
   M.p0↑ := M.p1↑;
   M.q0↑ := M.q1↑
END N.
```

The fingerprints of the variables *p0*, *p1*, and *q0*, *q1* depend on the fingerprints of *P* and *Q*, respectively, and are checked in module *N*, since these variables are imported and used by *N*. The fingerprint of *P* depends on the public fingerprint of *Desc* only. The private fingerprint of *Desc* is explicitly checked in *N*, because of the record assignment. However, it is not possible to check the private fingerprint of the base type of *Q*, because this base type is anonymous. Therefore, the fingerprint of the pointer type *Q* – and hence of *q0* and *q1* – has to depend on the private fingerprint of its base type. This is the case, since the public

fingerprint of an anonymous record is set to the value of its private fingerprint, and since the fingerprint of *Q* depends on this public fingerprint.

The public and private fingerprints of an (open) array structure are equal and depend on the public fingerprint of the element type of the structure. The fingerprint of a fixed-size array also depends on the number of elements, which is already contained in the identifier fingerprint.

The private fingerprint of a record structure depends on the complete allocation information listed in the symbol file (size, alignment factor, number of methods, both visible and hidden fields and methods), whereas its public fingerprint depends on the visible fields and methods only.

The local procedure *FPrintHdFld* searches for hidden fields that have nevertheless to be fingerprinted. This requires the scanning of nonexported fields being of a record type and the unrolling of nonexported fields being of an array type. Relevant hidden fields are pointers and procedures. They only contribute to the private fingerprint value of the record structure they belong to, if the flags controlling their presence in the symbol file are set. Public fingerprints do not include information on hidden pointers.

```
PROCEDURE FPrintHdFld(typ: Struct; fld: Object; adr: LONGINT);
(* modifies pvfp only *)
   VAR i, j, n: LONGINT; btyp: Struct;
BEGIN
   IF typ↑.comp = Record THEN FPrintFlds(typ↑.link, adr, FALSE)
   ELSIF typ↑.comp = Array THEN btyp := typ↑.BaseTyp; n := typ↑.n;
      WHILE btyp↑.comp = Array DO n := btyp↑.n * n; btyp := btyp↑.BaseTyp END ;
      IF (btyp↑.form = Pointer) OR (btyp↑.comp = Record) THEN
         j := nofhdfld; FPrintHdFld(btyp, fld, adr);
         IF j # nofhdfld THEN i := 1;
            WHILE (i < n) & (nofhdfld <= OPM.MaxHdFld) DO
               INC(adr, btyp↑.size); FPrintHdFld(btyp, fld, adr); INC(i)
            END
         END
      END
   ELSIF OPM.ExpHdPtrFld &
      ((typ↑.form = Pointer) OR (fld↑.name = OPM.HdPtrName)) THEN
         OPM.FPrint(pvfp, Pointer); OPM.FPrint(pvfp, adr); INC(nofhdfld)
   ELSIF OPM.ExpHdProcFld &
      ((typ↑.form = ProcTyp) OR (fld↑.name = OPM.HdProcName)) THEN
         OPM.FPrint(pvfp, ProcTyp); OPM.FPrint(pvfp, adr); INC(nofhdfld)
   END
END FPrintHdFld;
```

The local procedure *FPrintFlds* computes the contribution of each exported record field to the fingerprints of the enclosing record structure. The public fingerprint of a record structure depends on the accessibility (read-write or read-only), name and offset of each of its exported fields, as well as on the public fingerprint of the type of each of its exported fields. The private fingerprint of a record structure depends on the private fingerprint of the type of each of its exported fields, as well as on its hidden fields, if these are present in the symbol file.

```
PROCEDURE FPrintFlds(fld: Object; adr: LONGINT; visible: BOOLEAN);
(* modifies pbfp and pvfp *)
BEGIN
   WHILE (fld # NIL) & (fld↑.mode = Fld) DO
      IF (fld↑.vis # internal) & visible THEN
         OPM.FPrint(pbfp, fld↑.vis); FPrintName(pbfp, fld↑.name);
         OPM.FPrint(pbfp, fld↑.adr); FPrintStr(fld↑.typ);
         OPM.FPrint(pbfp, fld↑.typ↑.pbfp); OPM.FPrint(pvfp, fld↑.typ↑.pvfp)
      ELSE FPrintHdFld(fld↑.typ, fld, fld↑.adr + adr)
      END ;
      fld := fld↑.link
   END
END FPrintFlds;
```

The local procedure *FPrintTProcs* recursively traverses the scope graph of a record structure in order to find the procedures bound to the record. The public fingerprint of a record structure depends on the method number (*obj↑.linkadr*), on the signature and on the name of each exported type-bound procedure. The private fingerprint of a record structure depends on the method number of non-exported type-bound procedures if these numbers are listed in the symbol file.

```
PROCEDURE FPrintTProcs(obj: Object);    (* modifies pbfp and pvfp *)
BEGIN
   IF obj # NIL THEN
      FPrintTProcs(obj↑.left);
      IF obj↑.mode = TProc THEN
         IF obj↑.vis # internal THEN
            OPM.FPrint(pbfp, TProc); OPM.FPrint(pbfp, obj↑.linkadr);
            FPrintSign(pbfp, obj↑.typ, obj↑.link); FPrintName(pbfp, obj↑.name)
         ELSIF OPM.ExpHdTProc THEN
            OPM.FPrint(pvfp, TProc); OPM.FPrint(pvfp, obj↑.linkadr)
         END
      END ;
      FPrintTProcs(obj↑.right)
   END
END FPrintTProcs;
```

Fingerprinting Objects

Fingerprinting objects is much simpler than fingerprinting structures, because objects have only one fingerprint whose computation cannot be recursive. Similarly to *StrDesc*, the data structure *ObjDesc* of OPT gets a new field holding the fingerprint value, as well as a field indicating whether the fingerprint has been computed:

```
ObjDesc* = RECORD
   ...
   fpdone: BOOLEAN;
   fprint*: LONGINT;
   ....
END ;
```

The fingerprint value is stored in the object to avoid a recomputation when the fingerprint is needed more than once. The procedure generating the object file and the procedure comparing the new and old symbol files (see the next section) both require object fingerprint values.

```
PROCEDURE FPrintObj*(obj: Object);
   VAR fprint: LONGINT; f, m: INTEGER; rval: REAL; ext: ConstExt;
BEGIN
   IF ~obj↑.fpdone THEN
      fprint := 0; obj↑.fpdone := TRUE;
      OPM.FPrint(fprint, obj↑.mode);
      IF obj↑.mode = Con THEN
         f := obj↑.typ↑.form; OPM.FPrint(fprint, f);
         CASE f OF
         | Bool, Char, SInt, Int, LInt:
            OPM.FPrint(fprint, obj↑.conval↑.intval)
         | Set:
            OPM.FPrintSet(fprint, obj↑.conval↑.setval)
         | Real:
            rval := SHORT(obj↑.conval↑.realval); OPM.FPrintReal(fprint, rval)
         | LReal:
            OPM.FPrintLReal(fprint, obj↑.conval↑.realval)
         | String:
            FPrintName(fprint, obj↑.conval↑.ext↑)
         | NilTyp:
         ELSE err(127)
         END
      ELSIF obj↑.mode = Var THEN
         OPM.FPrint(fprint, obj↑.vis); FPrintStr(obj↑.typ);
         OPM.FPrint(fprint, obj↑.typ↑.pbfp)
```

```
      ELSIF obj↑.mode IN {XProc, IProc} THEN
        FPrintSign(fprint, obj↑.typ, obj↑.link)
      ELSIF obj↑.mode = CProc THEN
        FPrintSign(fprint, obj↑.typ, obj↑.link); ext := obj↑.conval↑.ext;
        m := ORD(ext↑[0]); f := 1; OPM.FPrint(fprint, m);
        WHILE f <= m DO OPM.FPrint(fprint, ORD(ext↑[f])); INC(f) END
      ELSIF obj↑.mode = Typ THEN
        FPrintStr(obj↑.typ); OPM.FPrint(fprint, obj↑.typ↑.pbfp)
      END ;
      obj↑.fprint := fprint
    END
  END FPrintObj;
```

The fingerprint of an object depends on the mode of the object. In case of a constant, it depends on the constant value. Module OPM exports additional fingerprinting routines for constant values that cannot be passed as parameter to the fingerprinting procedure without a machine-dependent type cast.

The fingerprint of a variable depends on its accessibility (read-write or read-only) and on the public fingerprint of its structure. The fingerprint of a normal or an interrupt procedure depends on the fingerprint of its signature only, whereas the fingerprint of a code procedure also includes the byte-stream inserted in the code at each call site of the code procedure (this byte-stream is present in the symbol file).

Finally, the fingerprint of a type object includes the public fingerprint of the type structure.

Consistency Checking at Compile Time

Inconsistencies should be detected as early as possible. Therefore, the compiler checks whether multiple imports of a same item are consistent. Inconsistent imports of constants, variables or procedures are not possible at compile time, because such objects can only be directly imported from a single symbol file. In contrast, types can be reexported and hence imported from different symbol files, as shown here:

```
MODULE M;
  TYPE
    T* = ... ;
END M.
```

```
MODULE N;
   IMPORT M;
   VAR a*: M.T;
END N.

MODULE O;
   IMPORT M, N;
   VAR b: M.T;
BEGIN b := N.a
END O.
```

In this example, the type *T*, which is originally exported from *M*, is imported into *N* and reexported from *N* as the type of the variable *a*. Therefore, *T* is present in both the symbol files of *M* and *N*. If *T* is modified and *M* is recompiled, but *N* is not recompiled, then the module *O* will import a different version of *T* from *M* than from *N*. This inconsistency can be detected when compiling *O*.

The compiler reads the symbol files in the order specified by the import list (which is not relevant). For each import of a module, a new module object is inserted into the symbol table, and a new scope graph is attached to this module object. Each object present in a symbol file is inserted into the scope graph of the module declaring this object. So, when the type *T* of the example above is read from the symbol file of *N* and is being inserted into the scope of *M*, the compiler notices that an object with the same name is already there, because it was inserted when reading the symbol file of *M*. In that case, the compiler has to compare the two versions of *T* and to inform the programmer of a possible inconsistency.

The original model simply compares the keys of the modules which are imported several times, whereas the layer model compares the fingerprints of the common layers. In the object model, the check consists in computing and comparing the fingerprints of both versions of *T*. The new version has to be completely loaded, so that its fingerprint can be computed. The problem is that the symbol table cannot hold multiple versions of the same type graph for several reasons. First, an object node reserves only one pointer field to hold its type graph. Second, type identity would not correspond to pointer identity any longer. Indeed, since Oberon favors type name equivalence, the compiler simply tests structure pointers for equality to decide whether two types are equal. Several versions of the same type would render this simple test impossible.

Managing two versions of a type graph (by attaching the second one to a temporary variable, for example) while keeping pointer identity for already loaded types is complex and may result in a different topology for a cyclic graph

and hence in a different fingerprint value. Storing the fingerprints of reexported types in the symbol file could simplify the problem, but would increase the symbol file size.

A simple solution to this problem exists: the procedure reading the symbol file first computes the fingerprint of the old version of a type before overwriting this old version with the new one. Overwriting means that the nodes describing the old structure are reused for the new version, thereby leaving the pointer values unchanged. The fingerprint of the new version is then computed and compared to the fingerprint of the old version. Note that only named types can be imported from different symbol files and hence can be inconsistent. Consequently, it is sufficient to reuse the nodes representing named structures only. Also, only the fingerprints of named structures are compared.

Before computing new fingerprints for a type, it is important to wait until all structures the type depends on are loaded from the symbol file. Otherwise, fingerprinting a partly loaded type graph would yield wrong results. For this reason, the routine internalizing structures from the symbol file does not call the fingerprinting routines after each loaded structure of a type graph, but after each strongly connected component of the graph. Since the routine externalizing the structures proceeds in a preorder fashion, the internalizing routine has to wait until a complete type graph is loaded.

It is then possible to compare old and new fingerprints for each named type. If the new public fingerprint of a structure is different, the compiler gives an error. In contrast, if the new private fingerprint is different, the compiler only gives an error if the private structure of the type is really used. For example, if the type *T* above was defined as a pointer to a record, it would be possible to insert new hidden fields in this record type and to recompile *M* without invalidating *N*, because declaring a variable of a pointer type does not require the private structure of the pointer base type. Also, the module *O*, which does not need this private structure either, could be compiled without errors, although two different private structures for the record type would be imported from *M* and *N*.

The compilation of *O* should not report an error because of inconsistent versions of *M.T*, otherwise it would be a waste of flexibility and would discard the advantages of having two fingerprints for record types. However, if the module *O* would export the pointer variable *b*, the pointer base type would be exported too and would have to be consistently imported then. The compiler therefore marks each type whose private part is inconsistently imported, and waits to report an error until this private part is really used or until this type is exported.

Comparing with the Old Symbol File

Contrary to the layer model, the object model does not require the history of development to ensure compatibility between a recompiled module and older clients. For this reason, the compiler does not need to read the old symbol file of a recompiled module. However, it is desirable that the compiler warns the programmer if an interface modification may invalidate clients. So, if the old symbol file is available, the compiler reads it to compare the new interface to the old one. If the interface is different, the compiler reports an error and does not register the new symbol file, unless the programmer explicitly allows the generation of a new symbol file by specifying a compiler option.

The compiler reads the old symbol file using the same routines as for symbol files of imported modules. It computes the fingerprints of old objects and structures that are also present in the new version of the interface. As in the layer model, a global record variable *impCtxt* manages context information that is discarded after each import of a symbol file. The field *ref* of this record is a table associating reference numbers with already loaded structures. The field *pvfp* holds the fingerprint value of each structure already in the symbol table until the fingerprints of the structures being loaded can be computed. The field *minr* is used to delay this fingerprinting until a strongly connected component of a type graph is completely loaded. The boolean field *self* indicates whether the symbol file being loaded is the symbol file of the module being compiled or of any other imported module.

Here is an excerpt from the routine that internalizes structures:

```
PROCEDURE InStruct(VAR typ: Struct);
   VAR mno: SHORTINT; ref: INTEGER; tag: LONGINT; name: OPS.Name;
      t: Struct; obj, old: Object;
BEGIN
   tag := OPM.SymRInt();
   IF tag # Sstruct THEN typ := impCtxt.ref[-tag]
   ELSE
      ref := impCtxt.nofr; INC(impCtxt.nofr);
      IF ref < impCtxt.minr THEN impCtxt.minr := ref END ;
      InMod(mno); InName(name); obj := NewObj();
      IF name = "" THEN
         IF impCtxt.self THEN old := NIL
         ELSE   (* insert an anonymous object, used to mark type descs *)
            obj↑.name := "@";
            InsertImport(obj, GlbMod[mno].right, old);   (* old = NIL *)
            obj↑.name := ""
         END ;
         typ := NewStr(Undef, Basic)
```

```
ELSE   (* insert a named object *)
   obj↑.name := name; InsertImport(obj, GlbMod[mno].right, old);
   IF old # NIL THEN   (* recalculate fprints to compare with old fprints *)
      FPrintObj(old); impCtxt.pvfp[ref] := old↑.typ↑.pvfp;
      IF impCtxt.self THEN   (* do not overwrite old typ *)
         typ := NewStr(Undef, Basic)
      ELSE   (* overwrite old typ for compatibility reason *)
         typ := old↑.typ; typ↑.link := NIL; typ↑.sysflag := 0;
         typ↑.fpdone := FALSE; typ↑.idfpdone := FALSE
      END
   ELSE typ := NewStr(Undef, Basic)
   END
END ;
impCtxt.ref[ref] := typ; impCtxt.old[ref] := old;
typ↑.ref := ref + maxStruct;   (* ref >= maxStruct means not exported yet *)
typ↑.mno := mno; typ↑.allocated := TRUE;
typ↑.strobj := obj; obj↑.mode := Typ; obj↑.typ := typ;
obj↑.mnolev := –mno; obj↑.vis := internal;   (* name not visible yet here *)
...
read structure into impCtxt.ref[ref]
...
IF ref = impCtxt.minr THEN   (* strongly connected component is complete *)
   WHILE ref < impCtxt.nofr DO
      t := impCtxt.ref[ref]; FPrintStr(t);
      obj := t↑.strobj;
      IF obj↑.name # "" THEN FPrintObj(obj) END ;
      old := impCtxt.old[ref];
      IF old # NIL THEN t↑.strobj := old;   (* restore strobj *)
         IF impCtxt.self THEN
            IF old↑.mnolev < 0 THEN
               IF old↑.history # inconsistent THEN
                  IF old↑.fprint # obj↑.fprint THEN
                     old↑.history := pbmodified
                  ELSIF impCtxt.pvfp[ref] # t↑.pvfp THEN
                     old↑.history := pvmodified
                  END
               (* ELSE remain inconsistent *)
               END
            ELSIF old↑.fprint # obj↑.fprint THEN
               old↑.history := pbmodified
            ELSIF impCtxt.pvfp[ref] # t↑.pvfp THEN
               old↑.history := pvmodified
            ELSIF old↑.vis = internal THEN
               old↑.history := same   (* may be changed to "removed" in InObj *)
            ELSE old↑.history := inserted   (* may be changed to "same" in InObj *)
            END
         ELSE
            (* check private part, delay error message until really used *)
```

```
                IF impCtxt.pvfp[ref] # t↑.pvfp THEN old↑.history := inconsistent END ;
                IF old↑.fprint # obj↑.fprint THEN FPrintErr(old, 249) END
              END
            ELSIF impCtxt.self THEN obj↑.history := removed
            ELSE obj↑.history := same
            END ;
            INC(ref)
          END ;
          impCtxt.minr := maxStruct
        END
      END
    END InStruct;
```

Record types need type descriptors for type tests and dynamic allocation at run time. The linker uses the name of the type to find and link type descriptors over module boundaries. If the type is anonymous, then the private fingerprint is used to identify the descriptor. A client using a type descriptor marks the type object as used (see next section). For this reason, the procedure *InStruct* creates anonymous objects for anonymous imported types.

The own symbol file is read after the parsing of the source text and the construction of the syntax tree. So, the objects and structures loaded from the old symbol file do not need to be kept after the fingerprint comparison. They are hence not inserted into the symbol table. It is important that declared structures and objects are not overwritten with possibly obsolete versions of them still contained in the old symbol file.

The result of the comparison is stored into a new object field called *history*. Structures have no *history* field, because only named structures are compared. So, each result of a structure comparison can be stored in the object containing the canonical name of the structure. This field *history* may hold 6 different values listed along with their meanings:

inserted
: The object is not present in the old symbol file. This is the default value.

same
: The fingerprints of the new and old objects are identical, as well as the public and private fingerprints of their respective structures.

pbmodified
: The fingerprints of the new and old objects differ.

pvmodified
: The fingerprints of the new and old objects are identical, as well as the public fingerprints of their respective structures, but the private fingerprints of their structures differ.

removed

The object is present in the old symbol file, but has been removed in the new interface.

inconsistent

The type object is imported from different symbol files with different private structures. This is not a problem if the public part only is used.

It is not possible to know whether the name of a type object was exported when reading its structure from the symbol file. One has to wait until the corresponding object is found or until the end of the symbol file is reached. So, if the new version of the object is not exported, the field *history* is tentatively set to *same* when the old structure is read, which is correct if the old object is not found. However, if the old object is found later on (see the next procedure), that means that it was exported and that its export mark has been removed in the new interface. Therefore, the field *history* is modified from *same* to *removed*.

On the other hand, if the new version is exported, the field *history* is tentatively set to *inserted* when the old structure is read, which is correct if the old object is not found, because it was not exported and an export mark has been inserted in the new interface. However, if the old version is found, that means that it was exported and the field is therefore modified from *inserted* to *same*.

Of course, the field *history* cannot be set to *same* if the fingerprints of the old and new objects are different. Here is the routine that internalizes objects:

```
PROCEDURE InObj(mno: SHORTINT): Object;
   VAR i, s: INTEGER; ch: CHAR; obj, old: Object; typ: Struct;
      tag: LONGINT; ext: ConstExt;
BEGIN
   tag := impCtxt.nextTag;
   IF tag = Stype THEN
      InStruct(typ); obj := typ↑.strobj;
      IF ~impCtxt.self THEN obj↑.vis := external END    (* type name is visible now *)
   ELSE
      obj := NewObj();
      ...
      read obj
      ...
   END ;
   FPrintObj(obj);
   IF (obj↑.mode = Var) &
      ((obj↑.typ↑.strobj = NIL) OR (obj↑.typ↑.strobj↑.name = "")) THEN
      (* compute a global fprint to avoid structural type equivalence for anonymous types *)
      OPM.FPrint(impCtxt.reffp, obj↑.typ↑.ref – maxStruct)
```

```
    END ;
    IF tag # Stype THEN
      InsertImport(obj, GlbMod[mno].right, old);
      IF impCtxt.self THEN
        IF old # NIL THEN
          (* obj is from old symbol file, old is new declaration *)
          IF old↑.vis = internal THEN old↑.history := removed
          ELSE FPrintObj(old);   (* FPrint(obj) already called *)
            IF obj↑.fprint # old↑.fprint THEN
              old↑.history := pbmodified
            ELSIF obj↑.typ↑.pvfp # old↑.typ↑.pvfp THEN
              old↑.history := pvmodified
            ELSE old↑.history := same
            END
          END
        ELSE obj↑.history := removed
        END
      END
    ELSE   (* obj already inserted in InStruct *)
      IF impCtxt.self THEN
        IF obj↑.vis = internal THEN obj↑.history := removed
        ELSIF obj↑.history = inserted THEN obj↑.history := same
        END
      END
    END ;
    RETURN obj
  END InObj;
```

As explained in the section on anonymous types and name equivalence, two variables of two different anonymous types may have the same fingerprints. It is therefore impossible to detect such an interface modification as the following:

```
VAR
  a*, b*: ARRAY 4 OF INTEGER;
  c*: ARRAY 4 OF INTEGER;
```

which is then modified to:

```
VAR
  a*: ARRAY 4 OF INTEGER;
  b*, c*: ARRAY 4 OF INTEGER;
```

All three variables *a*, *b*, and *c* have the same fingerprints in both versions, but *a* and *b* are not assignment-compatible any longer in the second version. The compiler computes a global fingerprint on the symbol file in order to detect

such an interface modification. The global fingerprint is simply a combination of all structure reference numbers of anonymous types of exported variables appearing in the file. So, the global fingerprint for the first version is

$16 \oplus 16 \oplus 17$

whereas the fingerprint of the second version is

$16 \oplus 17 \oplus 17$

which is different (remember that the reference numbers from 0 to 15 are reserved for predefined types).

The procedure *InStruct* sets the field *history* in all type objects the procedure loads, but the procedure *InObj* sets the field *history* only in objects loaded from the old symbol file. Other nontype objects are imported only once from other modules than the module being compiled. These objects have no history, cannot be inconsistent, and cannot be reexported. The field *history* is checked when a structure or an object is externalized into the new symbol file:

```
PROCEDURE OutStr(typ: Struct);
   VAR strobj: Object;
BEGIN
   IF typ↑.ref < expCtxt.ref THEN OPM.SymWInt(–typ↑.ref)
   ELSE
      OPM.SymWInt(Sstruct);
      typ↑.ref := expCtxt.ref; INC(expCtxt.ref);
      IF expCtxt.ref >= maxStruct THEN err(228) END ;
      OutMod(typ↑.mno); strobj := typ↑.strobj;
      IF (strobj # NIL) & (strobj↑.name # "") THEN OutName(strobj↑.name);
         CASE strobj↑.history OF
         | pbmodified: FPrintErr(strobj, 252)
         | pvmodified: FPrintErr(strobj, 251)
         | inconsistent: FPrintErr(strobj, 249)
         ELSE    (* checked in OutObj or correct indirect export *)
         END
      ELSE OPM.SymWCh(0X)
      END ;
      ...
      write typ
      ...
   END
END OutStr;

PROCEDURE OutObj(obj: Object);
   VAR i, j: INTEGER; ext: ConstExt;
BEGIN
   IF obj # NIL THEN
      OutObj(obj↑.left);
      IF obj↑.mode IN {Con, Typ, Var, LProc, XProc, CProc, IProc} THEN
```

```
        IF obj↑.history = removed THEN FPrintErr(obj, 250)
        ELSIF obj↑.vis # internal THEN
          CASE obj↑.history OF
          | inserted: FPrintErr(obj, 253)
          | same:   (* ok *)
          | pbmodified: FPrintErr(obj, 252)
          | pvmodified: FPrintErr(obj, 251)
          END ;
          ...
          write obj
          ...
        END
      END ;
      OutObj(obj↑.right)
    END
  END OutObj;
```

The procedure *FPrintErr* gives an error only the first time it is called, to avoid a large list of redundant errors. The error message includes the name of the concerned object, so that the programmer can decide either to allow the generation of a new symbol file (recompilation with option) or to edit the object declaration. The option *e* allows an interface to be extended only. This guarantees that no client will be invalidated. The option *s* allows an interface to be modified, which does not exclude a client invalidation. Depending on the options, the procedure *FPrintErr* suppresses the output of some errors. Here is the list of the possible error messages:

249: X is not consistently imported, recompile imports
250: X is no longer visible, compile with \s
251: X is redefined (private part only), compile with \s
252: X is redefined, compile with \s
253: X is new, compile with \e

X is replaced by the effective object name. The error number is listed here for comparison with the code listed above, but is not visible in the real message.

Object File Format

The compiler has to write linking information into the object file, so that the linking loader can resolve external references. This information may have the form of a fix-up chain linking all the instructions that access an imported

variable, for example. The linking loader follows this chain and inserts the absolute address in the code. The imported variable may be of a record or array type, and may then be accessed with an offset. Each address is formed by adding the absolute address of the variable to the offset, which is written into the code array of the object file together with the link to the next instruction.

Depending on the target architecture, it might not always be possible to store both the variable offset (typically 32-bit unsigned) and the link (typically 16-bit signed for 32K aligned instructions, i.e. 128KB of code per module for RISC architectures). This is not a problem with the MIPS processor [28] which needs two instructions to access a 32-bit address. Since the first instruction loading the upper half of the address is implicitly known (*LUI*, *Load Upper Immediate*) and since its target register is also the base register of the next instruction, all 32 bits of this first instruction are used to code the offset. The 16 bits coding the lower half of the address in the second instruction can be used for the link. If the target architecture does not allow such a compression or if the restriction of 128KB of code per module is too severe, a separate table can replace the chain in the code array.

In the original model, as well as in the layer model, a single fix-up chain is sufficient for all variable accesses of a module, because the offset of each variable is known from the symbol file. This is different in the object model: the offset of an imported variable is not known at compile time, because the offset is a context-dependent attribute not listed in the symbol file. Therefore, a separate fix-up chain is necessary for each imported object.

This has repercussions on the code generator of OP2. Indeed, the module OPL has to manage a chain for each imported object whose access needs fix-up at link time. The root of the chain is stored in the field *linkadr* of the object. Since type descriptors may also be accessed over module boundaries, they also need a chain. This is why the procedure *InStruct* creates objects for anonymous types.

The linking loader checks the consistency of each imported object by comparing the fingerprint of the object listed in the exporting module with the fingerprint of the object listed in the importing module. If a mismatch is detected, the module is unloaded and an error is reported.

The object file needs therefore an export section and an import section. The object file format may be slightly different from one target architecture to the other. The format presented here is used in the Oberon System running on MIPS-based workstations. The export section (export block *ExpBlk*) consists of a list of exported items:

```
ExpBlk   =  82X {EConst | EType | EVar | EProc | ECProc | EStruct | TDesc | LinkProc} 0X.
EConst   =  1X name fprint.
EType    =  2X name fprint.
EVar     =  3X name fprint offset.
EProc    =  4X name fprint entry.
ECProc   =  5X name fprint.
EStruct  =  6X name pbfprint pvfprint.
TDesc    =  8X (name | 0X pvfprint) link recsize ( –1 | basemod (name | 0X pvfprint))
            nofmth nofinhmth nofnewmth nofptr {mthno entry} {ptroff}.
LinkProc =  9X entry link.
```

The format of each item depends on the kind of object as indicated by the first byte. Although the value of an exported constant (*EConst*) is not linked in client modules but inserted at compile time in the code, its fingerprint must nevertheless be checked at link time, because the client might use an obsolete value.

A type object (*EType*) is exported with its name and fingerprint. Remember that structure objects (*EStruct*) have a public fingerprint and a private fingerprint and must be listed separately to type objects. Each exported variable (*EVar*) also lists its offset. Since offsets of global (exported or nonexported) variables are known at compile time, a single chain links them all. The root of this chain appears in the header of the object file (see appendix). For some target architectures, the compiler could take advantage of pc-relative addressing for global variables.

Exported procedures (*EProc*) are listed with their relative entry point. No fix-up chain is necessary for global (exported or nonexported) procedures called in the module of their declarations, because pc-relative addressing is used. However, assignments of global procedures to variables use absolute addresses inserted by the linker (*LinkProc*). Code procedures (*ECProc*) are not linked, but are very similar to constants, since their "value", the byte-stream to be inserted into the code at the call site, is present in the symbol file.

The export section also contains information for allocating type descriptors (*TDesc*). Type descriptors are not checked for consistency, because the information listed here is not used by client modules or is already checked by structure fingerprints (hidden pointer fields, for example). Type descriptors are listed in the export section for convenience only: the same traversal of the symbol table generates information for exported objects and for type descriptors. They could be listed in a separate section.

For each type descriptor, a fix-up chain (*link*) links the instructions referencing the type descriptor (type tests and calls to NEW). Also, information is provided for allocating and initializing the type descriptor (record size, base record, new methods, pointer offsets). The type extension table (used for run-time type tests) is copied from the type descriptor of the base record and

the own type is then inserted in the new table. Similarly, the method dispatch table is copied from the base type descriptor – *nofinhmth* lines are copied to the new table consisting of *nofmth* lines – and the entry address (*entry*) of the new methods (*nofnewmth*) are inserted into the lines corresponding to the method number (*mthno*), possibly overriding copied entry addresses from the base type descriptor.

Note that record types declared in procedures may also need type descriptors. However, such a type descriptor is always listed without its name in the object file, in order to avoid a possible name collision with a descriptor of a globally declared type. An anonymous descriptor is identified by its private fingerprint. This explains the presence of anonymous types as base type of an extended type in the production *TDesc* above. In the same production, the number *-1* means that the type is not extended. Otherwise the module declaring the base type is specified by its index *basemod* in the list of imported modules (*ImpBlk* below) and the base type is identified by its name or its private fingerprint, if it is anonymous (0X).

If several exported anonymous types have the same private fingerprint, the linker allocates only one type descriptor and uses it for all of these types in a client module. The linker picks the first anonymous type descriptor with the required private fingerprint in the export section. Remember that structural type equivalence replaces type name equivalence for anonymous types used over module boundaries without affecting memory integrity.

The import section of the object file consists of an *import block* and a *use block*. The import block lists the names of imported modules:

```
ImpBlk  =  81X {name}.
```

The use block contains as many lists of used items as there are modules in the import block, and in the same order. Each list is *0X*-terminated and enumerates used items imported from the corresponding module of the import block. Each item is listed with its name, its fingerprint (except for type descriptors) and, if necessary, the root of its fix-up chain (*link*):

```
UseBlk   =  89X {{UConst | UType | UVar | UProc | UCProc | UpbStr | UpvStr | LinkTD} 0X}.
UConst   =  1X name fprint.
UType    =  2X name fprint.
UVar     =  3X name fprint link.
UProc    =  4X name fprint link.
UCProc   =  5X name fprint.
UpbStr   =  6X name pbfprint.
UpvStr   =  7X name pvfprint.
LinkTD   =  8X (name | 0X pvfprint) link.
```

The object file lists only the imported objects that are really used. A new boolean field *used* is declared in the object descriptor *ObjDesc* and is set to *true* if the object is accessed by the symbol table handler. Examining the root of the chain in an object is not sufficient to decide whether this object is used or not: an imported constant, for example, is never linked.

Remember that structures are sometimes checked separately from objects. Clients may use the public part of a structure only, and sometimes the private part too. Two new boolean fields, *pbused* and *pvused*, are declared in the structure descriptor *StrDesc*. The front-end sets the field *pbused* when a variable of a named pointer type pointing to the named structure is dereferenced. The field *pvused* is set when the structure is

- the base record of an extending record
- the element type of an (open) array
- the type of a declared variable or of a record field
- the type of a formal value parameter
- the type of the destination of an assignment
- the actual parameter of a call to the standard function SIZE

In other words, a client module that does not use a structure in any of these cases is not dependent on the private part of this structure. Therefore, such a client is not invalidated when the private part of this structure is modified.

Linker and Run-Time Data Structures

In the Oberon System, a module is loaded from its object file into the main store when a command of this module is invoked for the first time, or when a client module of this module is being loaded. A loaded module then remains in memory for the rest of the session, unless the user explicitly unloads it.

The declarations of the run-time data structures necessary to represent loaded modules in the Oberon heap are the following:

```
TYPE
   Name = ARRAY 32 OF CHAR;

   Export = RECORD
      name: Name;
      fprint, adr: LONGINT;
      mode: INTEGER
   END ;
```

```
Module = POINTER TO ModuleDesc;
ModuleDesc = RECORD
   next: Module;
   name: Name;
   refcnt: INTEGER;
   exports: POINTER TO ARRAY OF Export;
   imports: POINTER TO ARRAY OF Module;
   tdescs, data, code: POINTER TO ARRAY OF LONGINT;
   ...
END ;
```

A loaded module is represented by a module descriptor and a module block. Module descriptors are fixed-size records (*ModuleDesc*) linked to form the module list. Each descriptor contains several attributes of the module, such as its name, its reference count (used for module unloading) and different pointers to the different sections of its module block. The module block is a juxtaposition of sections containing the module code and global data, the list of imported modules, the list of type descriptor addresses, and so on. The size of each section is determined at load time. The size of the module block is therefore different for each module.

The array of *Export* nodes is particular to the object model. It replaces an array of layer fingerprints and an array of procedure entry addresses in the layer model. These arrays are used for consistency checking and for module linking. Loading, linking, and consistency checking of a module M involves the following steps:

1. Open the object file of M, read the import block, and recursively load, link, and check each module imported from M.

2. Search for module M in the module list; if it is found, then it has been correctly loaded through an explicit call to the loader during the initialization of a module loaded in step 1, quit.

3. Allocate a module descriptor and a module block for M on the heap; load the various sections of the object file into the corresponding sections of the module block, except for the use block, which is not stored in the module block, but discarded after step 4.

4. For each item scanned from the use block in the object file, find the corresponding exported item in the *Export* array of the corresponding imported module, compare the fingerprints of both items, patch the item references in the code of M with the absolute address obtained from the

field *adr* of *Export*; if the item is not found or if the fingerprints differ, report an error, unload the module M, and quit.

5. Initialize the type descriptors and call the initialization part of module M.

These steps are identical in the original model and in the layer model, except for step 4. Contrary to the object model, both other models do not require used items to be searched in the export section, since variable offsets and procedure entry numbers are known from the symbol file at compile time already. Also, type descriptors are accessed indirectly through global pointers. However, the number of references to be patched is the same in all three models.

If the same order is used for items in use blocks as for items in export sections, step 4 can be executed by two parallel sweep phases, one on each list of used items and one on each corresponding export section of imported modules. For this reason, the compiler writes the items into the various lists in the object file in alphabetical order. Consequently, the execution time of step 4 is linearly proportional to the sum of used items and of exported items in imported modules (see the next chapter for benchmarks).

The procedure *FindExp* searches for a named item in an export section of an imported module. The procedure takes the mode, name, and fingerprint of the item as parameter, and returns the address of the item, which is a dummy address for types or constants. The pointer variable *curexp* indicates the current position of the sweep phase in the export array and *limexp* indicates the limit of the array (pointer arithmetic is used for efficiency reasons). The contents of the variables *curexp* and *limexp* are preserved over successive calls to the procedure *FindExp*:

```
VAR curexp, limexp: POINTER TO Export;

PROCEDURE FindExp(mode: INTEGER; VAR name: Name; fprint: LONGINT; VAR adr:
LONGINT);
BEGIN
   LOOP
      IF curexp = limexp THEN object not found; EXIT END ;
      IF (curexp.name = name) & (curexp.mode = mode) THEN
         IF curexp.fprint # fprint THEN fingerprint mismatch END ;
         adr := curexp.adr;
         INC(S.VAL(LONGINT, curexp), SIZE(Export));
         EXIT
      END ;
      INC(S.VAL(LONGINT, curexp), SIZE(Export))
   END
END FindExp;
```

The procedure searching type descriptors is slightly more complicated, because a type descriptor may be anonymous. In that case, the type descriptor is identified through its private fingerprint. Anonymous type descriptors are listed together at the beginning of the *UseBlk* section and of the array of *Export* nodes (because they have the "smallest" possible name: *0X*), but they are not sorted according to their fingerprint values. This requires to restore the current position (*curtd*) of the sweep phase at the beginning of the list, each time an anonymous type descriptor has been searched.

```
TYPE TDescPtr = pointer to type descriptor;
VAR curtd, limtd: LONGINT;

PROCEDURE FindTDesc(VAR name: Name; fprint: LONGINT; VAR adr: TDescPtr);
   VAR save: LONGINT; td: TDescPtr;
BEGIN save := curtd;
   LOOP
      IF curtd = limtd THEN tdesc not found; EXIT END ;
      S.GET(curtd, td);
      IF td.name = name THEN
         IF (td.pvfprint = fprint) OR (fprint = 0) THEN
            adr := td; INC(curtd, 4); EXIT
         END ;
         IF name # "" THEN fingerprint mismatch; EXIT END
      END ;
      INC(curtd, 4)
   END ;
   IF name = "" THEN curtd := save END
END FindTDesc;
```

Remember that the fingerprint of a named type descriptor is not checked. In that case, the procedure *FindTDesc* is called with *0* as *fprint* value.

When an error occurs, both the mode and name of the item are written to exported variables of the loader, so that a clear error message can be generated by the system to inform the user of the exact problem.

Chapter 7

Efficiency Considerations and Conclusions

Both the layer model and the object model have been implemented in the version of OP2 generating native MIPS code. This version of the compiler is used in the Oberon System implementations for MIPS-based workstations, namely *DECoberon* for Digital Equipment DECstations and *SGIoberon* for Silicon Graphics workstations [29]. After a period of testing and comparison, the object model has become the standard and is now distributed with these Oberon System implementations in place of the original model.

The first section of this chapter compares the new models to the original one, in terms of efficiency and implementation costs. Possible improvements to the object model are proposed in the second section. The third section draws the conclusions of this thesis.

Implementation Costs and Measurements

In the following, different cost factors such as compilation time, symbol and object file size, linking and loading time, and run-time memory requirements are measured and compared for all three models. The Oberon's line drawing system, called *Draw* [12, Chap. 13], is used for the benchmarks. This graphics editor is a typical Oberon application consisting of 5 modules representing a total of 46200 bytes of compiled MIPS code for 1684 lines of Oberon code:

Table 7.1 Modules of the Draw graphics editor

Module	Code	Source	Imports	Exports
Graphics	14736	566	47	92
GraphicFrames	12448	483	104	29
Draw	7656	287	95	15
Rectangles	4560	125	56	8
Curves	6800	223	35	9
Total	46200	1684		

The last two columns in table 7.1 reflect the number of imported and used items, as well as the number of exported items. An item can be an object, a type, or a type descriptor. These numbers are particularly relevant in the object model, since an imported item increases the linking time, whereas an exported item requires run-time memory. Furthermore, both imported and exported items consume space in the object file, as shown in the following table:

Table 7.2 Object file size

Module	Orig. Model	Layer Model	Object Model
Graphics	18256	18270 (+0.08%)	19552 (+7%)
GraphicFrames	14532	14570 (+0.26%)	16027 (+10%)
Draw	9319	9337 (+0.19%)	10526 (+13%)
Rectangles	5491	5503 (+0.21%)	6156 (+12%)
Curves	7942	7948 (+0.08%)	8383 (+6%)
Average		+0.16%	+9.2%

As expected, the size increase due to the layer model is almost unnoticeable, especially as each module had only one layer for the benchmarks. Every additional layer takes only 4 bytes in the object file.

The object model is more greedy. Each import or export of an individual object costs 10 bytes on average in the object files of the modules above. This number depends on the identifier length and therefore on the programming style.

The relative increase of 9.2% is rather pessimistic, because the modules of Draw import and export many objects in proportion to their small code size. The same measurements on all modules of the Oberon base system report an average increase of 6%.

Table 7.3 shows the size of the symbol files in each model. The size reduction is due to the new symbol file format, which is more compact.

Table 7.3 Symbol file size

Module	Orig. Model	Layer Model	Object Model
Graphics	2193	2100 (–4.2%)	1975 (–9.9%)
GraphicFrames	1402	1362 (–2.8%)	1243 (–11.3%)
Draw	225	227 (+0.9%)	205 (–8.9%)
Rectangles	642	639 (–0.5%)	571 (–11.1%)
Curves	660	654 (–0.9%)	585 (–11.4%)
Average		–2.7%	–10.6%

The difference between the layer model and the original model is small, because the space occupied by the fingerprints in the layer model is compensated by the encoding of integers. In contrast, the object model does not store fingerprints in the symbol file. The cumulated effect of integer encoding and elimination of context-dependent attributes like variable offsets is clearly visible in the last column.

The largest part of the original compiler is left unchanged by the implementation of either the layer model or the object model. The compilation tasks affected by modifications are the reading and writing of symbol files, storage allocation, and generation of the object file, but the lexical analysis, syntax analysis, and code generation are (almost) not modified. All three models generate identical code, with the exception of the object model, which eliminates an indirection in the code accessing type descriptors.

Table 7.4 shows the time necessary to compile all five modules of Draw, using the different models. Besides the total execution time, the time spent in each of the different compilation tasks is listed separately.

The measurements have been done on a DECstation 5000 Model 200 running at 25MHz with 24MB of RAM and a 332MB hard disk with 16ms average seek time. All times are *elapsed time* expressed in milliseconds (the sum of *user time* and *system time* would not include the disk access time). Due to a rather low clock resolution of about 16ms, compilations were repeated several times. Displayed results are therefore average values.

Table 7.4 Compilation time of all 5 Draw modules in ms

Compilation task	Orig. Model	Layer Model	Object Model
Reading symbol files	380	326	412
Parsing & tree	933	932	943
Reading own sym file	–	43	54
Sym file generation	59	42	23
Storage allocation	14	–	10
Code generation	303	304	293
Writing object file	634	642	652
Total	2323	2289	2387
		–1.5%	+2.8%

Although the symbol files are more compact in the object model than in the original model, reading the symbol files takes more time (+8%) with this model. The reason is that the object model immediately computes the fingerprints of the items that are imported several times from different symbol files. Avoiding these recomputations by storing the fingerprints in the symbol file would be a bad choice. Indeed, the symbol files, which are now 10% smaller than in the original model, would be about 20% larger, i.e. about 10% larger than in the original model. Since the time necessary to read a symbol file is proportional to the file size, the time difference would be higher than the current 8%.

In the layer model, the old symbol file is read before parsing the source text. After each declaration of a global object, the scope graph of the old symbol file is traversed to find the old version of the just declared object. This symbol table lookup is necessary to keep the object in the same interface layer, in order not to invalidate clients. This lookup is neither done in the original model, nor in the object model. However, the time difference is too small to be noticeable.

The time indicated for generating the symbol file does not include the time necessary for writing this file to the disk, because, in the benchmarks, the symbol file was identical to the old one and hence not registered. On the other hand, this time includes the time necessary to compare the new interface to the old one. Each model uses a different technique: the object model compares the fingerprints of old and new objects, the layer model the fingerprints of old and new layers, and the original model the byte-streams of the old and new files.

One can see that the byte-stream comparison is the more expensive technique because it involves the reading of a file, but it is largely compensated

by the fact that the original model does not need to internalize the own symbol file into the symbol table.

In the layer model, storage allocation is intertwined with symbol file generation and is a multi-pass process. Therefore, the time indicated for the symbol file generation includes the time for storage allocation. In contrast, storage allocation in the object model is always done in a single pass, without differentiating exported objects from nonexported objects (remember that the allocation order is not relevant, since context-dependent attributes are not stored in the symbol file), and without possible object reallocation as in the layer model.

The total time for symbol file generation, symbol file internalization, and storage allocation is nearly the same for all three models. Also, the cost of code generation is identical in all three models. The object model produces larger object files and computes the fingerprint of used items, which explains the time difference for object file generation.

In order to determine the impact of the number of interface layers on the compilation time in the layer model, measurements have been done with 8 layers per module interface. No noticeable difference could be observed, because of the noise induced by unpredictable cache effects and file buffering.

Table 7.5 shows the loading and linking time of the Draw modules. Here too, the same operation was repeated several times. However, the precision of the measurements is better here, because fewer files and smaller code were involved.

Table 7.5 Loading and linking time of the Draw modules in ms

Module (buffered)	Original Model	Layer Model	Object Model
Graphics	41.1	41.2	46.6
GraphicFrames	33.1	33.2	39.5
Draw	23.6	24.0	28.8
Rectangles	15.7	16.0	19.3
Curves	19.9	20.1	22.2
Total	133.4	134.5	156.4
		+0.8%	+17%
Total (not buffered)	421	421	456
		+0%	+8%

Modifications in the object file format are reflected in the structure of the linking loader. Since the layer model only replaces a key by a list of fingerprints in the export section of an object file and a key by both a number and a fingerprint in the import section, the modifications in the linker are minimal. Consequently, the loading and linking time are the same in the original model and in the layer model.

The last column of the table indicates that the object model is more expensive. However, the prominent slow-down of 17% is biased. Indeed, the consecutive loading and linking of the same module, in order to get more precision, results in a buffering of the object files by the underlying operating system. If object files are not buffered, which is usually the case when a module is loaded, the loading time increases by a large amount, whereas the linking time remains constant. The percentage of overhead therefore decreases from 17% to 8%, which corresponds approximately to the object file size increase in the object model. The cost of a more complicated linking technique is hence negligible in comparison to the cost of loading larger object files.

Memory requirements at run time are almost the same for the original model and for the layer model, as shown in the following table:

Table 7.6 Run-time memory requirements in bytes

Module	Orig. Model	Layer Model	Object Model
Graphics	17888	17920	21696
GraphicFrames	14272	14304	15424
Draw	9600	9632	10208
Rectangles	5504	5536	5824
Curves	8032	8064	8352
Total	55296	55456	61504
		+0.29%	+11.2%

Here again, the object model is more greedy, since a description of each exported item is kept in memory to allow clients to be linked later on. Each description includes the item mode, name, address, and fingerprint, which occupies a total of 44 bytes, whereas the corresponding information occupies only 4 bytes for each procedure entry in the other models. The size of debugging information is included in the numbers above and is the same in all three models, about 5KB in total.

The last table reflects the implementation costs of each model in the OP2 compiler. The code size of the MIPS version is listed for each module of the compiler. The self-compilation time is also displayed.

Table 7.7 Compiler size and self-compilation time

	Orig. Model	Layer Model	Object Model
Code size (bytes)			
OPM	4776	4344	4800
OPS	7600	7600	7600
OPT	16280	17296	20264
OPB	39072	39144	39200
OPP	24400	24696	24496
OPL	28000	28408	30856
OPC	33896	34280	34496
OPV	18072	20832	18120
OP2	3592	3592	3696
Total	175688	180192	183528
		+2.6%	+4.5%
Source size (lines)	7085	7344	7516
		+3.6%	+6.1%
Self-compilation (ms)	7476	7696	7731
		+2.9%	+3.4%

The overhead in both code size and compilation time of about 4% stays within very tolerable limits.

One point has not been brought up yet: all three models write exported objects in alphabetical order to the symbol file. Internalizing a symbol file into the symbol table therefore results in a degenerated tree. Measurements have shown that using degenerated trees instead of balanced trees for imported objects costs 0.8% of the total compilation time. Rebalancing these trees is therefore not recommended, especially as this operation would also cost execution time. Contrary to other models, the object model does not require the objects being written in a canonical order to the symbol file. However, this order simplifies the detection of interface modifications concerning exported variables of an anonymous type (see precedent chapter).

Outlook

Currently, the object model has also been adopted by two other Oberon implementations based on OP2, namely *HP-Oberon* [31] for Hewlett-Packard PA-RISC workstations and a commercial Oberon programming environment. It is also planned that other Oberon implementations from ETH will switch to the object model.

Migrating from the original model to the object model does not only require modifications in the compiler and in the module loader, but also in some associated tools. For example, the object file decoder must be adjusted to the new object file format. Similarly, the module browser, which generates text from symbol files, is replaced by a portable version based on the revised symbol file format, which is identical on all platforms. The browser first calls module OPT of the compiler to internalize a symbol file and then generates the corresponding interface text while traversing the loaded symbol table.

Automating Recompilations

One can imagine further tools to help the programmer to restore consistency in a system after an inconsistency has been detected. The compiler and the module loader produce error messages containing the name of the faulty object. This facilitates the identification of the module needing a recompilation. However, recompiling this module may in turn require further client recompilations. This phenomenon is known as trickle-down recompilation. A tool could help the programmer to find the optimal recompilation sequence. The tool would take a list of modules as parameter. The top modules of a hierarchy of modules would be sufficient. The tool would then analyze the dependences between these modules and the modules imported by them and return a list of modules to be recompiled in topological order.

The structure of this tool would be very similar to the one of the module loader. The tool would recursively read object files and check fingerprints. Module names and export sections of the corresponding object files would be kept in an internal data structure, whereas code and data sections would not be loaded. In order to predict trickle-down recompilations, it would be necessary to analyze the effect of a recompilation onto a module interface. The tool would therefore call the compiler to load symbol files (fingerprint checking in internalizing routines would have to be disabled) in order to calculate the new fingerprint values for exported objects.

This tool would generate the compile command with the list of modules to be recompiled, or call the compiler to automate recompilations. However, a fully automated recompilation would not be possible in case of a source code invalidation, which would require the intervention of the programmer for editing changes.

Inserting New Type-Bound Procedures

As in the layer model, the revelation of hidden fields or hidden methods is impossible in the object model without invalidating clients. As explained before, a modification of the Oberon scope rules for record fields would be necessary to avoid field name collisions in extending record types. Contrary to record fields, the revelation or insertion of type-bound procedures would not pose any problems in client modules. Indeed, an existing type-bound procedure would just override an equally named type-bound procedure newly inserted in a base record.

The fingerprint of a record type would not contain information about type-bound procedures (methods) any longer, so that its value remains unchanged when type-bound procedures are inserted or revealed. Each type-bound procedure would have its own fingerprint. Method table indices (method number) would not be attributed to type-bound procedures any longer, since such a number is a context-dependent attribute that could be modified by the insertion of a new type-bound procedure. The total number of type-bound procedures for a record type would be determined at link time only, since record types being extended by this record might have more type-bound procedures than expected.

Method tables would have to include the method names in addition to the method addresses, so that method tables for extending records can be constructed at link time using the method tables of the extended records, and so that a table index can be attributed to each imported method. This would require a fix-up of each external method call at link time. All the calls to the same method would be linked by a fix-up chain. An entry in the use block of the object file would list the root of this chain, as well as the name of the type-bound procedure, the name of the static type of the receiver, and the fingerprint value. The fix-up would consist in finding the index of this type-bound procedure in the method table of the receiver type using the method name, and to insert this index in the code.

This improvement in flexibility would result in larger object files, more work at link time, and greater run-time memory requirements. It is not clear whether

this improvement would be really useful in practice. As an example, one can imagine a class hierarchy of graphical objects being unable to print themselves in a first release. The insertion of a *print* method in the base class would not invalidate derived classes. However, this method would be useless until derived classes override it with their own method knowing how to print derived objects.

Similarly, the improvement in flexibility introduced by the distinction between public and private fingerprints for record types has not really been demonstrated yet. This distinction does not cost as much as the improvement described above would, but it is not for free, since additional fingerprints take space in object files and run-time data structures. Also, some compiler routines could be simplified if the distinction would not have to be done. Hopefully, the utilization of the object model in long-term Oberon projects will provide an answer.

Multiple Interfaces

A symbol file plays two roles in Oberon. First, it describes the part of the module symbol table needed by the compiler to perform type checking over module boundaries. Second, it is a compressed representation of the module interface that can be made available in readable form to the programmer of client modules by the browser. In this last respect, the symbol file often documents the interface at a level of abstraction that is too low for most of the clients. As proposed by J. Gutknecht [32], multiple interfaces should reflect the services provided by modules at different levels of abstraction.

The object model is best suited to support multiple interfaces. One can imagine a *symbol file editor* that would allow the programmer of a module to remove some particular objects from the module interface before distributing the edited symbol file to less trustworthy clients. For example, the programmer could hide procedures manipulating sensitive data structures that these clients should not use. This symbol file editor would work without any modifications either in the current compiler or in the module linker.

Note that multiple interfaces are possible without requiring an additional tool if the source code of a module is available, which is usually the case in this context. Indeed, the programmer can remove export marks in the text of a module before recompiling it, thereby producing a slimmed version of the symbol file for this module.

The layer model is less appropriate for this kind of interface editing. The programmer would have to be careful only to remove entire layers of objects

from the top of the layer stack of an interface. A symbol file editor is therefore recommended for the layer model.

Applying the Models to Other Languages

Both presented models could also be applied to compilers for other strongly-typed programming languages. The layer model is rather universal and could rapidly be adapted to other languages using symbol files or some canonical representation of module interfaces. More work would probably be necessary for the object model, since the fingerprint computation is highly dependent on the type system of the language. For example, fingerprints would not include type names in a language favoring structural type equivalence instead of type name equivalence. Furthermore, the computation of the fingerprint for a recursive type might require several traversals of the cyclic type graph if the cycle cannot be easily cut as in Oberon. However, the principles of fingerprint computation presented in this thesis are applicable to other languages. Context-dependent attributes of an object should not be included in the fingerprint of this object, because such attributes depend on the history of development of the module declaring the object. It is preferable that fingerprints do not depend on history in order to avoid invalidations caused by the accidental loss of this history.

Another problem could arise for implementing the consistency check at link time if a standard linker must be used. In that case, fingerprint values could be checked by some code inserted in the initialization part of each module by the compiler. As an alternative, the fingerprint could be appended to the name of each item of the object file needing link editing. The linker would not be able to link an inconsistent item since its name would be different in the exporting and importing modules due to the different fingerprint value. A similar technique is proposed by M. A. Ellis and B. Stroustrup [33] to increase the safety of function linkage in the C++ language. The string appended to the function name simply encodes the type of every formal parameter of the function, by using one character for each predefined type or the complete name for each user-defined structure. However, the string does not reflect the internal structure of a formal parameter type and is only used for functions. The authors admit that this technique has severe limitations and is just a step in the right direction, contrary to the object model, which improves the flexibility of separate compilation, which is safe already.

Conclusions

The objective of this work was to allow modules to be extended by new exported items without requiring a recompilation of client modules and without sacrificing type safety at link time. This has been achieved by increasing the resolution of consistency checking at compile time and at link time.

The first presented model, the layer model, organizes every module interface as a stack of extension layers growing as new items are exported from the module. This approach has the severe drawback that the history of development of each module is included in the symbol file of this module. If this history information gets lost, the recompilation of an unchanged module may nevertheless invalidate clients.

This problem is solved by the second model, the object model, which increases the resolution of consistency checks from the layer level to the object level. Each object receives at compile time a fingerprint containing its type information. The linking loader checks the fingerprints of exported and used objects for equality.

Both models attain the envisaged goal, but the object model has proven more practical than the layer model. Besides the fact that the history of development is not retained in the object model, this model also allows the elimination of obsolete objects without invalidating clients not using these objects, which is impossible in the layer model.

In comparison to other work, the models do not require clients to be known at compile time in order to ensure consistency. This is particularly important in today's systems of modules that do not live in closed programs as in the past, but in object-oriented environments open to new clients and ready to accept new functionality at any time.

The new models are integrated in the compiler and linking module loader and do not require a database, dictionary, or similarly centralized information recording, thereby avoiding maintenance problems. The only context information is present in the form of symbol files, which are unavoidable for separate compilation. These symbol files can always be reconstructed from the source text, since they include neither timestamps nor arbitrary keys. Furthermore, the consistency checking is not an optional operation performed by a separate tool, but is performed unconditionally at link time by the linking loader itself.

At a modest implementation cost, both presented models combine the flexibility of module extension in an open environment with the security offered by separate compilation. However, the object model is preferred to the layer model, since it does not depend on the history of development.

Appendix A: Layer Model File Formats

Names are sequences of characters terminated by 0X. Lower case identifiers denote numbers. A digit appended to an identifier indicates the length of the number in bytes (LSByte first). Otherwise, the number is compressed into a variable number of bytes by the procedure *WriteNum* of the Oberon module *Files* (LSByte first, base 128, cleared MSBit is stop bit, see below). The binary representation of a set is interpreted as an integer word and is coded by *WriteNum*. Floating point numbers are in IEEE format (LSByte first).

```
PROCEDURE WriteNum(x: LONGINT);
  BEGIN
    WHILE (x < – 64) OR (x > 63) DO
      Write(CHR(x MOD 128 + 128)); x := x DIV 128
    END ;
    Write(CHR(x MOD 128))
END WriteNum;
```

Symbol File

```
SymFile    =  0FAX Module {{Object} FPrint}.

Module     =  0 | ((negmno layerno | MNAME name) {FPrint} END).

FPrint     =  FPRINT value.

Constant   =     CHAR value:1
             |   BOOL (FALSE | TRUE)
             |   (SINT | INT | LINT | SET) value
             |   REAL value:4
             |   LREAL value:8
             |   STRING name
             |   NIL.

Object     =     Constant name
             |   TYPE Struct
             |   ALIAS Struct name
             |   (RVAR | VAR) Struct offset name
             |   (XPRO | IPRO) Signature entryno name
             |   CPRO Signature len {code:1} name.
```

```
Field     =  ((RFLD | FLD) Struct name | (HDPTR | HDPRO)) offset.

Method    =  (TPRO Signature name | HDTPRO) methno entryno.

Signature =  Struct {(VALPAR | VARPAR) Struct offset name} END.

Struct    =      negref
             |  STRUCT Module name [SYS value]
                (  PTR Struct
                |  ARR Struct nofElem
                |  DARR Struct
                |  REC Struct size align descAdr nofMeth {Field} {Method} END
                |  PRO Signature).
```

MNAME	=	16.	XPRO	=	31.	predefined refs:					
FPRINT	=	17.	IPRO	=	32.						
END	=	18.	CPRO	=	33.	BYTE	=	1.			
TYPE	=	19.	STRUCT	=	34.	BOOL	=	2.			
ALIAS	=	20.	SYS	=	35.	CHAR	=	3.			
VAR	=	21.	PTR	=	36.	SINT	=	4.			
RVAR	=	22.	ARR	=	37.	INT	=	5.			
VALPAR	=	23.	DARR	=	38.	LINT	=	6.			
VARPAR	=	24.	REC	=	39.	REAL	=	7.			
FLD	=	25.	PRO	=	40.	LREAL	=	8.			
RFLD	=	26.				SET	=	9.			
HDPTR	=	27.				STRING	=	10.			
HDPRO	=	28.	boolean constants:			NIL	=	11.			
TPRO	=	29.	FALSE	=	0X.	NOTYP	=	12.			
HDTPRO	=	30.	TRUE	=	1X.	POINTER	=	13.			

MIPS Object File

```
ObjFile   =  OFtag HeaderBlk EntryBlk CmdBlk PtrBlk ImpBlk LinkBlk ConstBlk CodeBlk
             TypeBlk RefBlk.

OFtag     =  0F8X 36X.

HeaderBlk =  refsize:4 nofentr:2 nofcom:2 nofptr:2 nofrec:2 nofmod:2 noflink:2
             datasize:4 consize:2 codesize:2 noflayer:2 {fprint:4} modname.

EntryBlk  =  82X {pc:2}.

CmdBlk    =  83X {name pc:2}.

PtrBlk    =  84X {off:4}.

ImpBlk    =  85X {noflayer:2 [fprint:4] name}.
```

```
LinkBlk    = 86X {mod:1 entry:1 pc:2}.

ConstBlk   = 87X {con:1}.

CodeBlk    = 88X {instr:4}.

TypeBlk    = 89X {recsize:4 tdadr:2 basemod:2 baseadr:2 nofmth:2 nofinhmth:2
             nofnewmth:2 nofptr:2 name {mthno:2 entno:2} {ptroff:4}}.

RefBlk     = 8AX {0F8X procend savedr savedf frame callarea name
             {Mode Form adr name}}.

Mode       = Var | VarPar.
Var        = 1X.
VarPar     = 3X.

Form       = Byte | Bool | Char | SInt | Int | LInt | Real | LReal | Set | String | Pointer.
Byte       = 1X.
Bool       = 2X.
Char       = 3X.
SInt       = 4X.
Int        = 5X.
LInt       = 6X.
Real       = 7X.
LReal      = 8X.
Set        = 9X.
String     = 0AX.
Pointer    = 0DX.
```

Appendix B: Object Model File Formats

Names are sequences of characters terminated by 0X. Lower case identifiers denote numbers. A digit appended to an identifier indicates the length of the number in bytes (LSByte first). Otherwise, the number is compressed into a variable number of bytes by the procedure *WriteNum* of the Oberon module *Files* (LSByte first, base 128, cleared MSBit is stop bit, see below). The binary representation of a set is interpreted as an integer word and is coded by *WriteNum*. Floating point numbers are in IEEE format (LSByte first).

```
PROCEDURE WriteNum(x: LONGINT);
  BEGIN
    WHILE (x < – 64) OR (x > 63) DO
      Write(CHR(x MOD 128 + 128)); x := x DIV 128
    END ;
    Write(CHR(x MOD 128))
END WriteNum;
```

Symbol File

```
SymFile    =  0FBX Module {Object}.

Module     =  0 | negmno | MNAME name.

Constant   =      CHAR value:1
             |    BOOL (FALSE | TRUE)
             |    (SINT | INT | LINT | SET) value
             |    REAL value:4
             |    LREAL value:8
             |    STRING name
             |    NIL.

Object     =      Constant name
             |    TYPE Struct
             |    ALIAS Struct name
             |    (RVAR | VAR) Struct name
             |    (XPRO | IPRO) Signature name
             |    CPRO Signature len {code:1} name.
```

```
Field     =  ((RFLD | FLD) Struct name | (HDPTR | HDPRO)) offset.

Method    =  (TPRO Signature name | HDTPRO) methno.

Signature =  Struct {(VALPAR | VARPAR) Struct offset name} END.

Struct    =      negref
             |  STRUCT Module name [SYS value]
                (  PTR Struct
                |  ARR Struct nofElem
                |  DARR Struct
                |  REC Struct size align nofMeth {Field} {Method} END
                |  PRO Signature).
```

MNAME	=	16.	XPRO	=	31.	predefined refs:					
not used		17.	IPRO	=	32.						
END	=	18.	CPRO	=	33.	BYTE	=	1.			
TYPE	=	19.	STRUCT	=	34.	BOOL	=	2.			
ALIAS	=	20.	SYS	=	35.	CHAR	=	3.			
VAR	=	21.	PTR	=	36.	SINT	=	4.			
RVAR	=	22.	ARR	=	37.	INT	=	5.			
VALPAR	=	23.	DARR	=	38.	LINT	=	6.			
VARPAR	=	24.	REC	=	39.	REAL	=	7.			
FLD	=	25.	PRO	=	40.	LREAL	=	8.			
RFLD	=	26.				SET	=	9.			
HDPTR	=	27.				STRING	=	10.			
HDPRO	=	28.	boolean constants:			NIL	=	11.			
TPRO	=	29.	FALSE	=	0X.	NOTYP	=	12.			
HDTPRO	=	30.	TRUE	=	1X.	POINTER	=	13.			

MIPS Object File

```
ObjFile   =  OFtag HeaderBlk ImpBlk ExpBlk CmdBlk PtrBlk ConstBlk CodeBlk UseBlk
             RefBlk.

OFtag     =  0F9X 36X.

HeaderBlk =  refsize:4 nofexp:2 noftdesc:2 nofcom:2 nofptr:2 nofimp
             newreclink newsyslink newarrlink datalink datasize consize codesize modname.

ImpBlk    =  81X {name}.

ExpBlk    =  82X {EConst | EType | EVar | EProc | ECProc | EStruct | TDesc | LinkProc} 0X.
EConst    =  1X name fprint.
EType     =  2X name fprint.
EVar      =  3X name fprint offset.
EProc     =  4X name fprint entry.
```

```
ECProc     = 5X name fprint.
EStruct    = 6X name pbfprint pvfprint.
TDesc      = 8X (name | 0X pvfprint) link recsize ( –1 | basemod (name | 0X pvfprint))
             nofmth nofinhmth nofnewmth nofptr {mthno entry} {ptroff}.
LinkProc   = 9X entry link.

CmdBlk     = 83X {name entry}.

PtrBlk     = 84X {off}.

ConstBlk   = 87X {con:1}.

CodeBlk    = 88X {instr:4}.

UseBlk     = 89X {{UConst | UType | UVar | UProc | UCProc | UpbStr | UpvStr | LinkTD} 0X}.
UConst     = 1X name fprint.
UType      = 2X name fprint.
UVar       = 3X name fprint link.
UProc      = 4X name fprint link.
UCProc     = 5X name fprint.
UpbStr     = 6X name pbfprint.
UpvStr     = 7X name pvfprint.
LinkTD     = 8X (name | 0X pvfprint) link.

RefBlk     = 8AX {0F8X procend savedr savedf frame callarea name
             {Mode Form adr name}}.

Mode       = Var | VarPar.

Var        = 1X.
VarPar     = 3X.

Form       = Byte | Bool | Char | SInt | Int | LInt | Real | LReal | Set | String | Pointer.

Byte       = 1X.
Bool       = 2X.
Char       = 3X.
SInt       = 4X.
Int        = 5X.
LInt       = 6X.
Real       = 7X.
LReal      = 8X.
Set        = 9X.
String     = 0AX.
Pointer    = 0DX.
```

References

[1] J. D. Ichbiah.
Reference Manual for the Ada Programming Language.
ANSI/MIL-STD-1815 A, United States Department of Defense, 1983.

[2] J. G. Mitchell, W. Maybury, and R. Sweet.
Mesa Language Manual.
Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[3] N. Wirth.
Programming in Modula-2.
Springer Verlag, fourth edition 1988.

[4] G. Nelson, editor.
Systems Programming with Modula-3.
Prentice Hall, 1991.

[5] N. Wirth.
The Programming Language Oberon.
Software – Practice and Experience 18(7): 671–690, July 1988.

[6] M. Reiser and N. Wirth.
Programming in Oberon: Steps beyond Pascal and Modula.
Addison-Wesley, 1992.

[7] D. G. Foster.
Separate Compilation in a Modula-2 Compiler.
Software – Practice and Experience 16(2): 101–106, Feb. 1986.

[8] M. L. Powell.
A Portable Optimizing Compiler for Modula-2.
Proceedings of the SIGPLAN 84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19(6): 310–318, June 1984.

[9] J. McCormack, Digital Western Research Laboratory, Palo Alto.
Private communication, 25 June 1992.

[10] E. Muller, Digital Systems Research Center, Palo Alto.
Private communication, 9 August 1993.

[11] H. Mössenböck and N. Wirth.
The Programming Language Oberon-2.
Structured Programming 12(4): 179–195, 1991.

[12] N. Wirth and J. Gutknecht.
Project Oberon: The Design of an Operating System and Compiler.
Addison-Wesley, 1992.

[13] J. Gutknecht.
Separate Compilation in Modula-2: An Approach to Efficient Symbol Files.
IEEE Software 3(6): 29–38, November 1986.

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

[15] SoftTech Microsystems Inc.
UCSD Pascal Users Manual.
San Diego, 1981.

[16] L. B. Geissmann.
Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith.
PhD dissertation No 7286, ETH Zürich, 1983.

[17] N. Wirth.
A Fast and Compact Compiler for Modula-2.
Report 64, Institut für Informatik, ETH Zürich, July 1985.

[18] J. J. Eberle.
Development and Analysis of a Workstation Computer.
PhD dissertation No 8431, ETH Zürich, 1987.

[19] R. Crelier.
OP2: A Portable Oberon-2 Compiler.
Proceedings of the Second International Modula-2 Conference, pages 58–67, Loughborough University of Technology, United Kingdom, 1991.

[20] R. Griesemer.
On the Linearization of Graphs and Writing Symbol Files.
Report 156, Departement Informatik, ETH Zürich, March 1991.

[21] N. Wirth.
From Modula to Oberon and the Programming Language Oberon.
Report 82, Institut für Informatik, ETH Zürich, September 1987.

[22] C. Bron, E. J. Dijkstra, and T. J. Rossingh.
A Note on the Checking of Interfaces Between Separately Compiled Modules.
ACM SIGPLAN Notices 20(8): 60–63, August 1985.

[23] M. Rain.
Avoiding Trickle-Down Recompilation in the Mary2 Implementation.
Software – Practice and Experience 14(12): 1149–1157, December 1984.

[24] W. F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems 8(3): 273–291, July 1986

[25] S. Feldman.
Make – A Program for Maintaining Computer Programs.
Software – Practice and Experience 9(3): 255–265, March 1979.

[26] R. Hood, K. Kennedy, and H. A. Müller.
Efficient Recompilation of Module Interfaces in a Software Development Environment.
ACM SIGPLAN Notices 22(1): 180–189, January 1987.

[27] H. Eidnes, S. O. Hallsteinsen, and D. H. Wanvik.
Separate Compilation in CHIPSY.
ACM SIGSOFT Software Engineering Notes 14(7): 42–45, November 1989.

[28] G. Kane.
MIPS R2000 RISC Architecture.
Prentice Hall, 1987.

[29] M. Brandis, R. Crelier, M. Franz, and J. Templ.
The Oberon System Family.
Report 174, Departement Informatik, ETH Zürich, April 1992.

[30] R. Rivest.
The MD5 Message-Digest Algorithm.
RFC 1321, MIT and RSA Data Security, Inc., April 1992.

[31] J. Supcik.
HP-Oberon: The Oberon Implementation for
Hewlett-Packard Apollo 9000 Series 700.
Report 212, Departement Informatik, ETH Zürich, February 1994.

[32] J. Gutknecht.
Variations on the Role of Module Interfaces.
Structured Programming 10(1): 40–46, 1989.

[33] M. A. Ellis and B. Stroustrup.
The Annotated C++ Reference Manual.
Addison-Wesley, 1990.