AN APPLICATION FRAMEWORK FOR COMPOSITIONAL MODULARITY

by

Guruduth S. Banavar

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

December 1995

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Guruduth S. Banavar

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Gary Lindstrom

Robert Kessler

Joseph Zachary

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Guruduth S. Banavar in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Gary Lindstrom
Chair, Supervisory Committee

Approved for the Major Department

Thomas Henderson
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

This dissertation presents a framework for the application of compositional modularity, a module model that facilitates extensive reuse of highly decomposed software.

Compositional modularity supports not only the traditional notions of program decomposition and encapsulation but also effective mechanisms for module recomposition. Based on a previously developed model, a suite of operators individually achieve effects of adaptation and combination on a simple notion of modules viewed as self-referential namespaces. This dissertation extends the previous model by introducing the notion of hierarchical nesting as a composition operation. Furthermore, this work shows that compositional modularity is unifying in scope. Important effects and idioms of advanced modularity, including several varieties of inheritance in object-oriented programming, find convenient expression within this model.

Compositional modularity can be applied within a wide range of systems that manipulate self-referential namespaces. To demonstrate, four distinctively differing systems based on the model are presented: an interpreter for a module extension to the programming language Scheme, a programmable linker for composing compiled object files, a compiler front-end for a compositional interface definition language, and a compositional document processing system. It is shown that systems such as the above derive important benefits from incorporating compositional modularity.

To facilitate the application of compositional modularity, the model is itself realized as a generic, reusable software architecture — an object-oriented *application framework* named ETYMA. ETYMA comprises a collection of interacting classes corresponding to the essential concepts of the model. The framework may be reused to efficiently build *completions*, i.e., tools for compositionally modular

systems. Three of the four systems mentioned previously were built as direct completions of ETYMA, and the fourth evolved in parallel with the framework. Significant design and code reuse was achieved in the construction of these system prototypes as completions of the framework.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My foremost thanks go to Gary Lindstrom, my research advisor. I will fondly remember my weekly meetings with Gary, where most of the ideas presented here actually germinated. I have learned many things, both technical and otherwise, from Gary.

In performing this work, I have "stood on the shoulders of" several individuals, two in particular. First, Gilad Bracha's dissertation paved the way for this work. This work owes much to his continued involvement, including his willingness to serve formally on my committee. Second, Doug Orr's work on the OMOS system was absolutely essential. His use of Erick Gallesio's STk package eventually led to the development of my *CMS* language.

Thanks also to Bob Kessler, Joe Zachary, and Mark Swanson for serving as members of my committee. Jay Lepreau's feedback and encouragement played a significant role in my work. Robert Mecklenburg's critical outlook and lively presence in talks have always been educational. I also want to acknowledge Wayne Rossberg's thoughts on document processing.

My parents have supported me throughout with their unfailing trust and encouragement. My grandfather Kittanna has been a constant source of inspiration for me ever since my childhood. Thanks also to the rest of my family, especially to Rama aunty and to Gani, for support in untold ways.

The Utah Computer Science department is a wonderful place to work. The members of the Computer Systems Laboratory are a great group to work with and learn from. Two of them, Jeffrey Law and Pete Hoogenboom, provided me with much implementation and systems help. Also, I could not quite have gotten along without the UNIX system people, especially Scott and Jimmy. The front office staff, Colleen, Loretta, Shawn, Raelynn, and Monica, have always been cheerful and efficient in their help.

My friends at Utah have made my stay memorable. In particular, I will remember the time spent with Kittu, Jim, Elena, Kumar, Ravi, Ranjan, the wallyball group, the drama group, and many students in the Indian community. I have also spent memorable times with my friends outside Utah during these past few years, especially with Raghu and Deven.

My spouse Nevenka is the one individual who has been the most responsible for my continued motivation to do this work. She has been the sounding board for many of my ideas. This dissertation owes her a great deal.

Finally, I must thank the Defence Advanced Research Projects Agency for funding the research projects that made this work possible.

CHAPTER 1

INTRODUCTION

Modularity is a fundamental facility for controlling complexity in large systems, via decomposition and abstraction. In particular, software modules allow programmers to develop and maintain pieces of a large system relatively independent of each other. However, decomposition alone does not support software component *reuse*, a widely accepted enabler of efficient construction of large systems. For this, it is necessary to provide mechanisms for effective *recomposition*, by which conforming modules can be composed to obtain other modules.

This dissertation focuses on the problem of linguistic mechanisms for, and broad applications of, module composition that enables a high degree of reuse.

Compositional modularity is a model that supports a simple notion of modules along with a powerful notion of their composition. The model distills, unifies, and further advances many existing notions of modularity. The ultimate goal of compositional modularity is to enable maximal reuse of software components. It encourages breaking down software into the smallest possible independently meaningful units and fosters extensive reuse by providing sophisticated and reliable mechanisms to build complex programs from these pieces.

Furthermore, the model can itself be realized as a reusable software architecture. This enables one to efficiently construct tools for compositionally modular systems.

This dissertation presents a consolidated account of a model of compositional modularity and its formulation as a reusable software architecture. Additionally, the dissertation demonstrates that the model is sufficiently general to be applied across a broad spectrum of computer software systems.

1.1 Composition and Reuse

The term "software composition" is a broad one and can be used to mean any number of ways of putting software together. For instance, in functional programming, composition of conforming functions is a well-understood concept. In data flow programming, conforming data filters can be composed to process data in compound ways. In conventional modular programming, a collection of software modules that interact with each other by calling each others' functions can be said to be composed to make up a system. This dissertation addresses software composition of yet another variety: that of composing the interfaces and implementations of software modules to obtain new modules.

A composition framework typically requires that components meet specific criteria in order to be composable. Furthermore, properties of a composition can be derived from the properties of constituent components. For instance, in the case of function composition, functions must have conforming types to be composable, and the type of composite functions can be derived from the types of the constituent functions. Similarly, in the module composition framework described here, modules must have conforming types to be composable, and the type of the resultant module can be derived from the types of the constituent modules. Furthermore, modules in the present framework are self-referential structures; hence the self-referential structure of the resultant module can be derived from the self-referential structure of the constituent modules.

Composable software components can be *reused* in many situations because a single component, built for one purpose, can participate in multiple applicable compositions. From a programmer's point of view, therefore, an effective software composition mechanism enables incremental programming. New components need only be developed to support incremental functionality over and above that of existing components, since these new components can be composed with existing ones. This dissertation shows how an effective module composition mechanism can enable significant software reuse.

There are important advantages to software reuse. It has the potential to considerably reduce the cost of program development. Reusing tested code, as opposed to writing new code, usually results in increased reliability. Reusable software helps control problems associated with software maintenance and evolution. Furthermore, mechanisms of software reuse are a necessary step on the way to the longstanding dream of off-the-shelf pluggable software parts and their factories.

The traditional notion of software reuse is that of *code* reuse, such as that obtained by using function code libraries. However, it is widely agreed that software *design* reuse is equally or more important. The design of software is embodied in its decomposition structure and the interfaces of decomposed units. One is said to have accomplished software design reuse if one reuses the decomposition structure and interfaces of existing software.

In order to achieve practical reuse, software must be designed with the specific goal of reuse in mind. The manner of decomposition of a system into modules, the design of module interfaces, and the manner of implementation of modules are all factors that dictate the reusability of modules. Moreover, reuse clients often need to know much more about a module being reused than what is specified in its interface, such as reuse dependencies; thus informative documentation is needed as always.

Once a set of modules is designed to be reusable, linguistic mechanisms (such as those supported by compositional modularity) make it possible to actually reuse them. In fact, it can be said that the design of reusable software is often constrained by the mechanisms by which it can eventually be reused. Thus, although linguistic mechanisms are not by themselves *sufficient* for reuse, they nevertheless greatly influence reuse and are certainly *necessary* for enabling principled, reliable, and efficient reuse. For instance, a typing discipline can require programmers to explicitly declare the interfaces of reusable components, the type system can verify that the interfaces of reused and client components conform, and the compiler may be able to generate efficient code for reused software based on type information. The following section introduces the model of compositional modularity.

1.2 Compositional Modularity

There have been two somewhat independent lines of development in modularity in programming languages. One is the notion of decomposition of programs into modules, whose interactions are controlled via traditional module systems and environment tools. Traditional module systems typically support three important features: encapsulation, i.e., the notion of an externally visible interface of a module separated from its hidden implementation; static checking of type conformity of module interactions (imports and exports); and lexical nesting of modules. These features are widely acknowledged to facilitate independent development and maintenance of modules.

The other line of development is the notion of data abstraction to support programmer defined data types, such as abstract data types (ADTs) and *classes* in object-oriented programming. ADTs and classes also represent decomposed program pieces, with support for encapsulation, static type checking, and nesting. However, classes may also be *recomposed* into other, usually larger, classes via mechanisms of *inheritance*. OO inheritance is thus a linguistic mechanism that supports reuse via incremental programming; i.e., one can program by describing how one software component differs from another existing one.

Compositional modularity represents an advancement of the former line of development to achieve and surpass the goals of the latter one. Compositional modularity surpasses class inheritance in that it supports a stronger and more flexible notion of reuse than traditional OO systems.

There are two aspects to the model of compositional modularity: that of modules and that of their composition. A module is simply a collection of names associated with bindings, which may in turn have references to names that are defined either within the module itself or external to the module. Such modules can be adapted and fitted in various ways to compose other modules, which in turn make up entire systems, much like putting *Lego* pieces together.

To be more accurate, a module is modeled as an *abstracted namespace*, i.e., a namespace that is abstracted over the names referenced within it. Abstracted namespaces can be manipulated in many desirable ways before actually *instantiating* them into concrete namespaces. Manipulation is performed using a suite of *operators*, each of which achieves an individual effect such as encapsulation, rebinding, or hierarchical nesting. This technique is shown in the rest of this dissertation to enable a high degree of adaptability and flexibility in their manipulation and, thus, enhanced reuse.

A significant characteristic of the model is that it is *unifying* in scope, in that it captures many existing notions of modularity, including most varieties of OO programming. One of the first tasks undertaken in this dissertation is to demonstrate in detail the expressive power of this model by emulating various existing models of modularity.

Another significant characteristic of the model is that it is *abstract*. That is to say, the model can be presented and analyzed without committing to the nature of the actual values that are bound to names within modules. Thus, the model is independent of particular underlying computational paradigms. This property is used to advantage in developing an application framework, outlined in Section 1.4.

In introducing the model of compositional modularity, it is important to acknowledge its lineage. The semantic foundations of the model go back to record calculi pioneered by Cardelli, Mitchell, and others [36, 16]. Classes were first modeled as record generators (records abstracted over themselves) by Cook[24], who also introduced some operators to manipulate generators. Based on this, Bracha and Lindstrom[9] promulgated the idea that OO inheritance is really a form of modularity and introduced a comprehensive suite of operations to manipulate record generators. Furthermore, Bracha [7] expressed this model abstractly and suggested that it could be formulated as a framework.

This dissertation further advances the semantic models proposed previously, by introducing the notion of *compositional nesting*. A problem with previous models was that direct lexical nesting of modules restricts reuse of the nested modules. Compositional nesting makes it possible to retroactively nest an independently developed module into another conforming module, via a composition operation.

This mechanism supports much enhanced compositionality and reuse.

The bulk of the contribution of this research, however, is in the practical application of the semantic model to various systems and in the engineering of a reusable software architecture to facilitate such application, as described below.

1.3 Compositionally Modular Systems

The most important idea put forth in this work is that the model of compositional modularity is applicable within a wide range of systems, perhaps much beyond what is explicitly presented here. This stems partly from the simplicity of the model and partly from the pervasiveness of modularity and namespace manipulation within software systems.

Compositional modularity supports reuse akin to OO inheritance by means of operations on self-referential namespaces. A key insight in this work is that there is indeed a wide range of software artifacts that can be modeled as self-referential namespaces. For instance, it is well known that recursive interface types can be viewed as self-referential namespaces [14, 2]. A traditional compiled object file can also be viewed as a self-referential namespace. Furthermore, structured document fragments can be modeled as self-referential namespaces. Even other artifacts, such as GUI components and file system directories can be regarded as recursive namespaces.

There currently exists a range of tools that manage the range of artifacts mentioned above. However, many such tools are usually based on disparate, and often impoverished, underlying models. It can be argued that it is advantageous to manage the above artifacts from the viewpoint of a well understood model such as compositional modularity, and design tools based on this viewpoint. The primary advantage of such an approach is that the underlying model of such tools can be significantly enriched, and reuse mechanisms akin to OO inheritance can be supported on the artifacts they manage. Moreover, the uniformity of the underlying model of such tools can be exploited to support better interactions between them.

The model of compositional modularity can be easily and effectively applied within systems that manipulate artifacts such as the ones given above. To demonstrate, this dissertation considers *four* radically differing systems and describes how to construct tools for them. The way in which each of these four systems can be modeled compositionally and the benefits to be derived from viewing them as such are given below.

I. *Scheme module composition.* In a conventional programming language such as Scheme, the notion of a module is that of an independent naming scope. A module comprises a set of identifiers bound either to locations (variables) or to any of the various Scheme values, including procedures. Procedures may contain references to other name bindings within the module. In this manner, a Scheme module may be modeled as a self-referential namespace.

   Several module systems for Scheme have been proposed previously [25, 77, 67], but these systems mainly provide a facility for structuring programs via decomposition. However, the ability to *recompose* first-class modules can additionally support design and implementation reuse akin to inheritance in OO programming. Furthermore, the notion of first-class modules and operations on them is consistent with the uniform use of first-class values and the expression-oriented nature of Scheme. Consequently, we argue that the incorporation of compositionality into a module system for Scheme can be very beneficial.

II. *Object module composition.* A separately compiled object file essentially consists of a set of symbols, each associated with data or code. This set of symbols is represented as a symbol table within the object module. Furthermore, internal references to these symbols are represented as relocation information within the object module. Thus, an object file can be modeled as a self-referential namespace.

   The traditional notion of linking object files corresponds to a rudimentary notion of composition. However, the full power of compositional modularity made available via a programmable linker can significantly enhance the ability to manage and bind object modules. In particular, facilities such as function interposition, management of incremental additions of functionality to libraries, and namespace management can be made more principled and flexible.

III. *Interface composition.* An interface is essentially a naming scope, with labels bound to types. Type constituents of the interface can recursively refer back to the interface itself. Thus, an interface can be regarded as a self-referential namespace.

Explicit specification and composition of interfaces, as embodied in interface definition languages (IDLs), are becoming necessities in modern distributed systems. Composition of interface specifications can help in the reuse and evolution of interface specifications.

IV. *Document composition.* A document can be regarded as a naming scope, consisting of section names, each associated with arbitrary text. Furthermore, there can be cross references from within a textual body to other section names.

Document composition can be useful in enterprises where several documents fragments are generated, edited, composed, maintained, and delivered in various ways. Document fragments generated for one purpose can be reused for other purposes. For example, a report, such as a user manual, can be composed from several document fragments, such as design documents.

1.4 An Application Framework

This dissertation presents a framework for applying the concepts of compositional modularity. It shows how to identify compositionally modular systems, how to model modules within those systems, and how to build tools for such systems. In this sense, it provides a conceptual "application framework" for compositional modularity.

However, the term *application framework* is used here in a more precise technical sense. An OO application framework is an abstract realization of the design of application software in a particular application domain. A framework captures the essential concepts in the domain as a set of interacting classes. Individual applications, called *completions*, are built by extending the framework in specific directions, i.e., by filling in the incomplete parts of the framework. Since OO programming provides several forms of effective software design and reuse mechanisms such as inheritance and polymorphism, it has been found to be suitable for developing application frameworks in several domains such as user interfaces and operating systems, as well as in several commercial and business applications.

The main point of application frameworks is that they enable applications in a particular application domain to reuse much of the design (and associated code) that is common to the domain. As argued earlier, this could significantly reduce the resources spent in developing applications, as well as increase the reliability of applications. Furthermore, it can be argued that studying an application framework for a particular domain is an efficient way to understand the domain, since the framework represents a "model" of the domain.

Naturally, the implementations of tools mentioned above share much in common, since they are all based on compositional modularity. It is therefore beneficial to abstract their common aspects, and realize them as a reusable software architecture. This dissertation presents an OO application framework for compositional modularity, realized in the C++ language. This framework, known as ETYMA, consists of several abstract and concrete classes corresponding to concepts commonly found in compositionally modular systems. For instance, at a very high conceptual level, such systems primarily consist of a domain of *values* and a corresponding domain of *types*. At a slightly more detailed level, typed values include *modules* and the *methods* constituting their attributes. The types of these values are known as *interfaces* and *function types*, respectively. ETYMA includes classes corresponding to each of the above concepts and several others.

However, ETYMA deliberately stops short of completely specifying all the details of its concepts, such as the precise nature of module attributes. Such details are supplied by individual completions, since these details differ from completion to completion. For instance, a module attribute may be a programming language function in one completion, whereas it may be something completely different such as a fragment of typesettable text in another completion. Within the framework, however, the differences between these two kinds of module attributes are abstracted away; both of them are viewed through a single abstract interface.

The architecture of the ETYMA framework is documented in this dissertation using the concept of design patterns [30]. The version of ETYMA documented here comprises about 45 reusable C++ classes in 7000 lines that evolved over six iterations over two years.

A tool for a system based on compositional modularity can be said to consist of a front-end that reads in command and data input, a processing engine that performs compositional operations on an internal representation (IR), and an optional back-end that transforms the IR into some external representation. The primary utility of the ETYMA framework is that it enables one to easily and rapidly build the processing engines, but not the front-ends and back-ends, for such tools. However, a tool front-end can use ETYMA classes to construct the IR from the module source. Module manipulation commands can then be translated into operations on the IR. In this manner, interpretive processors can directly manipulate the IR, whereas processors that do compilation must additionally provide a back-end that appropriately translates the IR into a target representation.

1.5 Completions

In order to construct a processing engine for a tool for a compositionally modular system, one must first identify the various kinds of name bindings comprising namespaces in the system. One can then identify generalizations of these concepts specified as classes in the ETYMA framework. For each such general ETYMA class, one must then subclass it to implement the more specific concept in the system. Once specialized classes for all relevant concepts have been defined, one is said to have modeled the system as a completion of ETYMA.

In this dissertation, four completions of ETYMA, corresponding to tools for the four systems mentioned in Section 1.3, are presented. Three of these were constructed as direct completions of the framework which resulted in significant design and code reuse. The fourth evolved in parallel with the framework and is thus termed as a "parallel completion."

A module extension to Scheme called Compositionally Modular Scheme, or *CMS* for short, is presented in Chapters 3 and 4. The implementation of an interpreter for *CMS* as a direct completion of ETYMA is presented in Chapter 5.

A programmable linking tool for compiled C language code that supports compositional modularity is presented in Chapter 6. This is a parallel completion of ETYMA, in that both the tool and the framework strongly influenced each other's development. However, although the tool's class design is the same as that of the framework, it is not physically derived from the framework.

A compiler front-end to an experimental compositional interface definition language, derived as a direct completion of ETYMA, is presented in Chapter 7. Also, an outline of how to extend a base language such as CORBA's IDL [60] is given there.

Finally, a compositionally modular document processing system, called MTEX, layered on top of the LATEX document preparation language is presented in Chapter 8. MTEX is also a direct completion of ETYMA.

1.6 Dissertation Organization

The next chapter presents the foundational and motivational concepts of compositional modularity. In particular, the starting point of this work, embodied in the programming language *Jigsaw*[7], is presented in some detail.

Following that, Chapter 3 demonstrates the expressive power of compositional modularity via several examples shown in the language *CMS*. The focus is on emulating the important idioms and styles of OO programming. This chapter also illustrates the programming style associated with compositional modularity.

Chapter 4 treats in detail the new notion of compositional nesting and its embodiment in *CMS*. Applications of nesting such as sharing and inheritance hierarchy combination are explored. Chapters 3 and 4 together should provide the reader a thorough understanding of the nature and power of compositional modularity.

The application framework ETYMA is described in Chapter 5. The design of its abstract and concrete classes is detailed. Immediately following that, the implementation of an interpreter for *CMS* as a completion of ETYMA is delineated. The reuse aspects of ETYMA and its evolution over reuse iterations are also explained.

Chapter 6 describes the application of compositional concepts to object modules. A software architecture that enables one to construct entire applications from individual components using these concepts is explained.

Chapter 7 shows how interfaces can be treated as compositional entities and describes the design of an IDL compiler as a completion of ETYMA. A system for document module composition is presented in Chapter 8.

Chapter 9 points to some future work, summarizes the accomplishments of this work, and presents conclusions.

At many points in this dissertation, denotational semantics and other mathematical formalisms are used as expository tools. It should become apparent that the purpose of formalism in this work is not to develop a complete theory but rather to express the ideas in a more precise manner, to clarify, or to relate to previous work in the area.

CHAPTER 2

SETTING THE STAGE

In this chapter, the basic concepts and problems that motivate the framework given in this dissertation are presented.

Classical modularity is primarily concerned with support for decomposition, encapsulation, static typing, and lexical nesting. In addition to these features, compositional modularity aims to support effective software reuse as well, via notions of module adaptation and composition.

The historical progression of increasing support for modularity in programming languages and systems is examined in Section 2.1, leading to the ideas underlying compositional modularity. In particular, the salient features of classical module systems and OO programming languages are examined. It is shown that these module systems fall short of fully supporting reuse.

The notion that compositional modules are abstracted namespaces is developed in Section 2.1.4. The prominent characteristics of the model of compositional modularity are described concisely in Section 2.1.5. Following that, the fundamental concepts of record generator manipulation and static typing are summarized in Section 2.2.1.

2.1 Modules and Module Systems

Intuitively, a software *module* is understood to be an independent unit of software with a well-defined interface. A *module system* is a model that supports the definition, manipulation, and use of modules. Module *manipulation* is concerned with composing modules by adapting them as necessary, in order to maximize their utility. Modules are used by client software by invoking the interface offered by them, possibly after *instantiating* them.

Several module systems have been proposed to date, but their detailed semantics vary greatly, especially as far as support for module manipulation is concerned. Classical module systems support a notion of modules mostly as a design-time program decomposition and structuring mechanism, with barely any support for module manipulation. At the other extreme, current day OO systems can be viewed as advanced module systems that support various forms of composition and reuse via inheritance.

2.1.1 Classical Module Systems

The simplest understanding of a module is as an environment that binds names to values — a *namespace*. All names in the module are directly accessible internally within the module, but a subset of the names, called the *interface*, is exported from the module for external access. An example of such modules is the *structure* construct of Standard ML (a generalized "record" with type, value, and structure components), whose public interface is given by a *signature* [76].

To enforce stronger separation between modules, some systems require that all interactions between modules be declared explicitly, by *importing* names used from the interfaces of other modules. Examples of such module systems are those of the Modula family [15] and the Scheme module system given in ref. [25]. Typically, such systems perform some level of static conformance checking (at compile time) and binding (at link time) between the imports and exports of modules. However, completely dynamic importation has also been proposed [77].

One way to specify modules is to describe each of them completely from scratch. However, such complete specification does not facilitate *reuse* of portions of the module that could potentially be common to several modules. To support reuse better, modules can be parameterized with the (free) names used within the module, as in SML functors [76] and ADA generic packages [39]. A parameterized module can be multiply instantiated (usually before run-time) with actual argument values to produce *concrete* modules, which are then used in the same manner as completely specified modules.

Despite support for import/export and parameterization, the above module systems are significantly impoverished in their support for our primary goals of decomposition, encapsulation, and reuse. For instance, decomposition is fixed at module definition time; module boundaries and interactions cannot be modified. The public interface of a module cannot be changed after the fact. Although parameterized modules can be reused by instantiating them multiple times with various argument values, there are no mechanisms to control the bindings of names other than the module parameters. In fact, the most one can do in these systems by way of module manipulation is to instantiate a parameterized module.

Clearly, more flexible and expressive module systems are needed. A step in this direction was taken by Tung [77], by supporting a simple notion of renaming imported names that conflict with those defined in a module. However, support for full compositionality and reuse requires the ability to perform several such operations on modules, as described later.

2.1.2 First-Classness

Another dimension of development in modularity in programming languages is that of what is considered to be a first-class run-time entity. A value is called first-class if it can be passed into and out of functions, stored in variables, and used as part of data structures in a programming language.

Module systems have traditionally been regarded as a design-time facility. Many languages, such as SML, support a compile-time module language separate from the core language. In these systems, one does not usually think of modules as first-class run-time values, or even as producing first-class run-time values.

However, there are compelling arguments for viewing instances of modules as first-class values. Most languages support some form of data abstraction facility that can be instantiated into first-class values, e.g., abstract data types (ADTs) and classes. Given that modules support design-time abstraction, it may be desirable to use one mechanism uniformly as the primary abstraction facility.

Furthermore, module systems facilitate the manipulation of namespaces. Given that namespaces exist as environments at run-time, it seems natural to support modules themselves as first-class entities. Additionally, this results in uniformity of manipulable values. Languages such as Pebble [11], Rascal [40, 41], and others [72] support first-class modules.

(It is worth mentioning that support for *higher-order* modules does not necessarily mean that modules are first-class. For instance, a higher-order SML module system such as in ref. [35] might support functors with functor parameters, or higher-order modules. However, functors are not run-time entities and, thus, not first-class. Alternatively, in this case, functors may be considered first-class with respect to the SML module language, although not with respect to the SML core language.)

First-classness can enormously enhance the reusability of modules. For instance, expressions over modules can allow various module manipulation mechanisms to be usefully composed to obtain composite effects. Moreover, familiar and expressive base-language mechanisms, such as conditionals and functions, can be used to create modules dynamically. Examples of such use of first-class modules are given throughout the next two chapters, particularly in Section 4.3.4.

Object-oriented programming, described in the following section, typically supports at least first-class instances of classes (modules) and occasionally first-class classes as well.

2.1.3 Object-Oriented Programming

Support for data abstraction and reuse have been the motivating forces behind classes and various forms of inheritance in OO programming.

Paradigmatically, an *object* in OO programming is an abstraction of a real-world entity. An object implements some behavior, which is exposed via an interface. Objects collaborate with other objects to perform user-level tasks, by exchanging *messages*. Upon receiving a message, the appropriate *method* that implements the behavior associated with the message is located and executed.

The common characteristics of a collection of objects is captured by the concept of a *class*, which may be instantiated into individual objects. Classes in OO programming essentially correspond to the notion of modules, since they support the same requirements of program decomposition, encapsulation, and reuse (via inheritance, described below).

Three characteristics distinguish OO programming: *encapsulation*, the notion that the implementation details of an object should be hidden from its clients; *inheritance*, the mechanism by which a class can reuse the design and code of other classes; and *polymorphism*, the notion that an object can be used in any context that requires (a subset of) the functionality exported by its interface. These three characteristics interact in important ways. Understanding their interactions and supporting all of them without compromising their integrity has been a serious research direction in the past few years.

Encapsulation is a primary requirement for large-scale software development. An implementor must be free to modify the implementation of a module as long as the interface, which is the module's contract with its clients, is not changed. (The expressiveness of the interface to satisfactorily state the contract is a crucial issue; present-day techniques typically employ some notion of type specification to express the contract, although more expressive mechanisms, e.g., ref. [70], are being studied.)

The mechanism of inheritance supports incremental programming. That is, a programmer can specify a module by stating how its implementation differs from that of an existing module. As a result, inheritance is a primary *implementation reuse* mechanism in OO programming. Inheritance gives rise to an implementation hierarchy of modules, which is often thought of and implemented as a graph-structured hierarchy of modules.

Inheritance must not violate encapsulation. That is, the inheritance history of a module is purely an implementation detail and must not be exposed via its interface. This can happen if classes are identified with types (described below) or when ancestors inherited from multiple inheritance paths are shared. In these seemingly innocuous situations, a class might cease to be valid if the inheritance relationships between classes (which is an implementation detail) are changed. Furthermore, if the implementation of a module directly depends on its inherited ancestors by naming them, its reusability will be compromised. For an extended treatment of these and other problems with violation of encapsulation, the reader is referred to Bracha's thesis [7], Chapter 2.

The interface supported by objects of a class is called their *type*. Many OO programming languages equate the concept of a class with that of a type. However, a type is an interface (a partial description of the behavior), whereas a class is the implementation of the behavior represented by the type.

An interface that supports at least as much functionality as another, possibly more, is called a *subtype* of the other. Polymorphism (more accurately, *inclusion* polymorphism [17]) is a direct consequence of subtyping relationships between objects' types. Polymorphism is usually implemented by using techniques of *late binding* of method calls to the actual code that implements the methods. Polymorphism is an important reuse mechanism in OO programming, since functions written for particular types of objects can be reused for objects having subtypes as well.

The question arises as to whether inheritance always results in classes that generate objects of a subtype of the "parent" class. The answer is *no*. In a sufficiently expressive language, inheritance does not necessarily result in subtypes. (The reason for this has to do with contravariant subtyping of binary methods [10].) Only recently has inheritance been widely understood as an implementation reuse mechanism, divorced from subtyping, which defines "*is-a*" relationships between objects [23].

High-performance OO languages that support static typechecking and separate compilation of classes impose special requirements on the implementation of inheritance. It must be possible to typecheck as well as compile a class with knowledge of only the interfaces of inherited and other classes used in the implementation of the class. (Typically, object layout information for these classes is also necessary.) Furthermore, it must be ensured that typechecked superclasses of a class continue to be type correct in the presence of inheritance, without actually re-typechecking the superclass.

Inheritance has traditionally been characterized as an operational mechanism. More recently, there have been some denotational characterizations of the notion of classes and inheritance [10, 24, 41]. The one that is relevant to this work is due to Cook, in whose original formulation [24], a class is viewed as a record abstracted over its own notion of what "self" means. References to symbols within the class are made via the abstracted self parameter. Furthermore, inheritance is viewed as an operation that appropriately modifies "self" and references to it.

The following section puts into context this idea of abstracting over self. It is shown that the what a namespace is abstracted over largely determines how effectively and flexibly modules may be manipulated.

2.1.4 What to Abstract Over?

As mentioned earlier, a module is essentially a namespace — a set of names bound to values. This can be modeled as a record in lambda calculus — a function from a finite set of names to their bindings. Some of the value bindings may be functions that *refer to* other names, as shown in box (a) of Figure 2.1. Some of these references will be to names within the namespace; others could be free references. It is crucial to determine how to model these references.

The central issue in this regard is to decide what modules should abstract over. Consider the following cases:

(i) Abstract over free variables. A module may be abstracted over only the free names referenced. This results in a parameterized module. In the example in Figure 2.1 (b), the namespace is abstracted over free references to b_1 and b_2. Such parameterized modules may be instantiated with actual argument values to produce concrete namespaces. In this manner, one gets control over the bindings of the abstracted names in the namespace.

(ii) Abstract over "self." Taking the above technique one step further, it may be desirable to get control not only over a subset of the names but over *all* the names in a namespace. For this, it is necessary to abstract over all the names in the namespace. However, for this notion to be useful, there must be a way to specify default bindings for abstracted names, which can be subsequently re-bound.

The above goals can be achieved by having modules abstract over the very namespace that is generated by the module, its notion of "self." As mentioned in the previous section, Cook presented this idea as a lambda abstraction he called a *generator*, as shown in Figure 2.1 (c). The formal parameter s stands for the namespace's notion of "self," which effectively lumps and abstracts over all the names in the namespace. The namespace contains names $a_1 \ldots a_n$ with bindings that refer to other names via the abstracted s parameter, e.g., $s.a_1, s.b_1$, etc.

In such a model, binding and rebinding of names is done via operations over generators that appropriately manipulate the s parameter. In fact, generators can be adapted and combined in many useful ways before actually instantiating them. Several examples of this are shown in Section 2.2.1. Bracha [7] shows that this notion of abstracting over self can be used to achieve a comprehensive array of individual effects of inheritance. Furthermore, a generator is instantiated by taking its fixpoint by applying the fixpoint operator Y. Such an individual *instance* is a concrete namespace. These notions are explained in more detail in Section 2.2.1.

(iii) Abstract over "self" as well as the surrounding environment. The above model still restricts reuse of nested modules, since nested modules are not accessible from outside the nesting module prior to instantiation of the outer module. Furthermore, consider that nested modules may refer to names in their lexically surrounding environment. It may not be desirable to "hard-wire" these nonlocal references to a particular environment by actually lexically nesting modules.

One way to further enhance the reusability of nested modules is to provide control over their nonlocal references as well. This can be done by abstracting a module further over its surrounding environment. This is shown in Figure 2.1 (d) as a lambda abstraction introduced here known as a *closed generator*. In a closed generator, references to names in the environment of a module is via the abstracted e parameter (e.g., $e.c_1$). At instantiation time, the environment of the namespace can be bound as desired. Closed generators permit one to retroactively embed a module into any conforming environment. This notion is developed in Chapter 4.

$$\begin{array}{ll} \{ & a_1 = \ldots a_2 \ldots; \\ & a_2 = \ldots\ldots; \\ & \ldots \\ & a_n = \ldots a_1 \ldots; \} \end{array}$$

$\boxed{\text{a}}$

$$\begin{array}{ll} \lambda b_1 \,.\, \lambda b_2 \,.\, \{ & a_1 = \ldots a_2 \ldots; \\ & a_2 = \ldots b_1 \ldots; \\ & \ldots \\ & a_n = \ldots b_2 \ldots a_1 \ldots; \} \end{array}$$

$\boxed{\text{b}}$

$$\begin{array}{ll} \lambda s \,.\, \{ & a_1 = \ldots s.a_2 \ldots; \\ & a_2 = \ldots s.b_1 \ldots; \\ & \ldots \\ & a_n = \ldots s.b_2 \ldots s.a_1 \ldots; \} \end{array}$$

$\boxed{\text{c}}$

$$\begin{array}{ll} \lambda e \,.\, \lambda s \,.\, \{ & a_1 = \ldots s.a_2 \ldots e.c_1 \ldots; \\ & a_2 = \ldots s.b_1 \ldots e.c_2 \ldots; \\ & \ldots \\ & a_n = \ldots s.b_2 \ldots s.a_1 \ldots; \} \end{array}$$

$\boxed{\text{d}}$

Figure 2.1. What to abstract over? (a) A module namespace modeled as a record. (b) A namespace abstracted over free references: a parameterized module. (c) A module abstracted over the entire namespace, "self" (shown as s): a *generator*. (d) A module abstracted over "self" as well as the surrounding environment: a *closed* generator.

2.1.5 Compositionality

The notion of full compositionality of modules unifies the major ideas from all the above progressions of module functionality. That is, it supports encapsulation, first-class modules, a variety of inheritance idioms, and compositional nesting via closed generators. In this section, a concise description of the goals and requirements of full compositionality is provided.

As mentioned earlier, the ultimate goal of compositional modularity is simply to get the maximum possible implementation reuse out of the components of a program. In order to maximize reuse, a program must first be decomposed into the smallest possible independently meaningful units. If programs are broken down into such units, then it is advantageous to be able to put them back together in as many ways as they can be profitably reused. In fact, the more powerful the mechanisms by which one can meaningfully combine them, the more reuse one can expect to get. This viewpoint, above all, characterizes the essence of the model of compositional modularity.

Another way to describe compositional modularity is to articulate a set of characteristics that are expected of module models that claim to support it. The desiderata for module systems, given in ref. [7], is a good starting point for such a list. Below, we first describe characteristics of traditional modularity and then describe how compositional modularity augments them.

Encapsulation is a crucial notion in traditional modularity. Modules must be able to hide their implementation and expose only an interface. Support for *nesting hierarchy* is also central, since decomposition naturally leads to hierarchy. Finally, *static type safety* should be attained when it is desirable and possible.

In addition to the above, we add the following two criteria for compositionality. First, the notion of *composability* says that one should be able to combine modules with each other to produce new modules, if they have compatible interfaces. This notion applies not only for combining modules at the same level but also for achieving hierarchical nesting. Second, the notion of *adaptability* means that one should be able to modify aspects of a module to make it suitable for reuse in new ways. Examples of adaptability are renaming and removal of attributes.

In practice, satisfying the requirements of composability and adaptability means at least that a module system supports a flexible form of inheritance, i.e., one which supports a large part of the spectrum of effects obtainable via single and multiple inheritance in various OO models. In this work, the above requirements are shown to be satisfied by supporting eight primary module operators, each of which achieves an individual effect of composition of modules. These operators can be used in combination to achieve familiar varieties of OO inheritance. Examples of their use are described in detail in Chapters 3 and 4.

In contrast, the common semantics of inheritance in OO languages is that classes and inheritance are *composite* notions and fulfill a variety of roles. For example, a class defines a module, the visibility of its attributes, the rebindability of its attributes, and also defines a type. Similarly, inheritance supports class combination, method redefinition, access to overridden methods, name conflict resolution, and definition of subtyping relationships among types.

Compositional modularity as defined above is at once a unification and distillation of several forms of modularity. The notion of module is boiled down to a simple notion of abstracted namespace, yet several sophisticated forms of manipulation can be performed on modules. The simplicity of the notion of module lends itself to application to a broad range of modular and nonmodular languages and even to notions such as object file linking, interface definition and combination, and document manipulation. In fact, the general concepts of compositional modularity can themselves be abstracted and expressed independent of the particular computational model of a *base language*. Such a generic model can then be expressed in an OO manner, constituting what is generally referred to as an *object-oriented framework* [42]. This approach was first suggested in Bracha's thesis [7], although no engineering strategies or implementation work was presented.

A more complete and detailed description of compositional modularity will be provided in Chapters 3 and 4 by presenting a language embodying its concepts.

2.2 A Formal Characterization

In this section, we summarize the formal semantics of *Jigsaw*[7], a module language that supports most notions of compositional modularity, and constitutes the starting point for this research. The concepts in this section will be covered in the concrete context of a variant of Scheme in the following chapter; thus this section may be skipped without loss of continuity. The formal semantics are given here so that extensions to this semantics can be presented formally in the rest of the dissertation.

2.2.1 Generator Manipulation

The semantics are specified here using the untyped lambda calculus — see the original description [7, 9] for full details.

As mentioned earlier, the basis of generator semantics goes back to record calculi. A record is characterized as a function from a finite domain of labels to a domain of values. Each label-value pair is called an *attribute*. Operators over records, such as for concatenation ($\|_r$), attribute overriding ($\leftarrow_r$), attribute removal ($\backslash_r$), renaming ($rename_r$) and selection ($._r$), are defined elsewhere [16] and not given here. The subscript r signifies that the operations are defined on records.

Applicatively, a module is modeled as a record generating function, or a *generator*. Its domain equation and an example follow:

$$Generator = Instance \rightarrow Instance$$

$$g = \lambda s.\ \{a_1 = v_1, a_2 = v_2, \ldots, a_n = v_n\}$$

The parameter s corresponds to the generator's notion of "self." The fixpoint $Y(g)$ of such a generator, a record, is called an *instance* of the module. Taking the fixpoint of the generator binds the generator's self-references $s.a_x$.

In terms of OO programming, a generator (or module) corresponds to a class, and an instance corresponds to an object of a class. Furthermore, the bindings of labels can actually refer to labels that are not defined within the generator. These undefined labels correspond to pure virtuals or abstract attributes, and the generator corresponds to an abstract class. Abstract classes should not be instantiated, since access to abstract attributes is undefined. The formal characterization reflects this fact: one cannot take the fixpoint of an abstract generator, since the range of the function is smaller than the domain.

Modules are combined using a suite of combinators that individually achieve effects of inheritance. Figure 2.2 shows the definitions of the primary combinators: merge, override, rename, restrict, freeze, hide, and copy-as. (Generator operators are given in the sans serif font for uniformity with their use in the rest of this dissertation.)

The operators given in the figure take generators as parameters and produce new ones whose s parameters are modified appropriately. For example, the operator merge takes in two generators g_1 and g_2 and produces a new generator whose body is a simple record concatenation ($\|_r$) of the two records given by $g_1(s)$ and $g_2(s)$.

$$\mathsf{merge} = \lambda g_1.\ \lambda g_2.\ \lambda s.\ g_1(s)\ \|_r\ g_2(s)$$

$$\mathsf{override} = \lambda g_1.\ \lambda g_2.\ \lambda s.\ g_1(s) \leftarrow_r g_2(s)$$

$$\mathsf{rename}\ a\ b = \begin{cases} \lambda g.\ \lambda s.\ g(s \leftarrow_r \{a = s._r b\})\ rename_r\ a\ b & \text{if } g \text{ defines } a \\ \lambda g.\ \lambda s.\ g(s \leftarrow_r \{a = s._r b\}) & \text{otherwise} \end{cases}$$

$$\mathsf{restrict}\ a = \lambda g.\ \lambda s.\ g(s) \setminus_r a$$

$$\mathsf{freeze}\ a = \lambda g.\ Y(\lambda f.\ \lambda s.\ g(s \leftarrow_r \{a = f(s)._r a\}))$$

$$\mathsf{hide}\ a = \lambda g.\ \lambda s.\ (freeze\ a)(g)(s) \setminus_r a$$

$$\mathsf{copy\text{-}as}\ a\ b = \lambda g.\ \lambda s.\ let\ super = g(s)\ in\ super\ \|_r\ \{b = super._r a\}$$

Figure 2.2. The generator definitions of the primary module combinators.

The definitions for the unary operators given in the figure are more involved; the interested reader is referred to ref. [7]. (Also, Chapter 3 explains the semantics of these operators, along with examples and idioms of their use, within the concrete context of an extension of the Scheme language.)

In order to model imperative semantics in this framework, we must account for the effect of instantiation on the store. This can be achieved by augmenting the semantics of generators using a notion of *constructors*. With this augmentation, when the fixpoint of a generator is taken, we get a constructor which can be applied to a store to get an instance. The domain equations are:

$$Generator = Constructor \rightarrow Constructor$$

$$Constructor = Store \rightarrow (Instance \times Store)$$

Modules are first class semantic entities; hence module attributes can themselves be modules. Such nested modules can contain free references, which implicitly refer to names in a surrounding lexical scope and are found in the module's *environment*. This semantics is modeled by passing the environment as an argument to the semantic function that creates constructors. The constructor retains the environment in which it was created. Upon instantiation, i.e., when applied to a store, it extends its retained environment with bindings arising from the attribute definitions in the module.

Given this, imperative operators analogous to the applicative ones given in Figure 2.2 can be formulated. However, we shall omit the details here for the sake of simplicity of presentation and work with the applicative operators.

2.2.2 Static Typing

A module has a type, called its *interface*, that is given by the types of all its attributes, both defined and declared. An attribute is said to be *defined* if it binds a name to a value and it is said to be *declared* if it simply specifies the type of the binding. For example, if a module defines attributes $a_1 \ldots a_m$ with values that have valid type signatures $\alpha_1 \ldots \alpha_m$ and declares attributes $d_1 \ldots d_k$ with valid type signatures $\delta_1 \ldots \delta_k$, then the module has an interface:

$$\{ \begin{array}{ll} \textbf{define} & a_1 : \alpha_1, \ldots, a_m : \alpha_m \\ \textbf{declare} & d_1 : \delta_1, \ldots, d_k : \delta_k \quad \} \end{array}$$

A static type system determines the type of new modules generated by module operators, provided that the types of the incoming modules satisfy certain properties. Rules for performing this, called *type rules*, for each module combinator have been specified as part of the *Jigsaw* language. The detailed rules are not reproduced here, but an English transcription for each operator is given below and summarized in Table 2.1.

In the following description, attributes from two modules are said to *conflict* if they have the same name. Furthermore, subtyping relationships between types can exist in the particular computational base language. Subtypes induce a partial ordering structure, usually a lattice structure, on types, and in most cases it is possible to determine the greatest lower bound (greatest common subtype) and the least upper bound (least common supertype) of pairs of types. For interface types, subtyping is given by type equality, since module operators require complete type information for their arguments, and thus are not polymorphic.

Table 2.1. Informal type rules for module combination.

Operator	Typing
merge(Module)	Combine interfaces. For conflicting attributes: definitions disallowed; a definition must be a subtype of declaration; for declarations, replace with greatest common subtype.
override(Module)	Same as merge, except conflicting definitions are allowed; incoming definition must be a subtype of conflicting definition.
restrict(Label)	Given attribute must be defined.
freeze(Label)	Given attribute must be defined.
hide(Label)	Given attribute must be defined.
rename(Label,Label)	First argument attribute must exist (declared or defined); second must not.
copy-as(Label,Label)	First argument attribute must be defined; second must not exist.

When two modules are merge'd, the interface of the resultant module is a combination of the interfaces of the incoming modules. The types of nonconflicting attributes in both modules are included in the resultant. For conflicting attributes, there are three cases, as follows. Conflicting defined attributes are altogether disallowed. If one is defined and the other declared, the defined attribute is included in the resultant if it is a subtype of the declared one. If both are declared, the attribute's type in the resultant module will be the *greatest common subtype* of the conflicting types.

The type rule for the override operator is the same as for merge, except in the case of conflicting defined attributes. In this case, the one from the right operand must be a subtype of the one from the left, and the attribute in the resultant module will be the more specific type: the subtype.

The type rules for restrict, freeze, and hide require that the given attribute is defined in the module. The rename operator requires that its first argument attribute is either defined or declared in the module and that the second argument is neither defined nor declared. The copy-as operator, on the other hand, requires that its first argument be defined and that the second one does not exist.

2.3 Summary

This chapter has explained the background and motivational concepts necessary to understand compositional modularity. The model has been placed in the context of progressions of flexibility, first-classness, reuse, and name-space abstraction in module systems. Classical module systems support decomposition, encapsulation, static type checking, and lexical nesting but do not support module manipulation to achieve reuse. OO programming systems do support manipulable modules but are usually inflexible and do not fully support nested modules.

Furthermore, a concise description of the goals and salient features of compositional modularity has been presented. The primary goal of compositional modularity is to support a flexible mechanism for composing highly decomposed programs to achieve a high degree of implementation reuse. It supports enhanced reuse over existing models with the new notion of compositional nesting.

We also summarized the formal generator semantics and static type system of the module manipulation language *Jigsaw*, which constitutes the starting point for this research.

CHAPTER 3

COMPOSITIONALLY MODULAR SCHEME

The purpose of this chapter is twofold. One is to introduce the concepts of compositional modularity in the concrete context of a programming language. The language presented here is called *Compositionally Modular Scheme*, or *CMS* for short, which is the programming language Scheme [22] extended to support compositional modularity.

The other purpose is to demonstrate that these notions are general enough to emulate idioms of advanced modularity such as OO inheritance. In particular, idioms such as abstract classes, super-based and prefix-based single inheritance, mixin-based and multiple inheritance, and wrapping of method definitions and calls are emulated.

Scheme is a dialect of the Lisp programming language. It is statically scoped, properly tail-recursive, dynamically typed and incorporates first-class procedures and continuations. It has a simple semantics and supports both imperative and functional styles of programming. In this spirit, the *CMS* module system supports modules as first-class entities, and it is dynamic and interactive.

The primitives of *CMS* are gradually introduced in Section 3.1. There are primitives to create modules, to combine and adapt them, to inspect them, to instantiate them, and to access their attributes.

Sections 3.2 and 3.4 then show how the above primitives can be applied individually or in combination with others to achieve various idioms of OO inheritance. All these concepts are illustrated with several examples.

3.1 Modules and Instances

In *CMS*, a module consists of a list of *attributes*, with no order significance. Attributes are of two kinds. *Mutable* attributes are similar to Scheme variables and can store any Scheme value. *Immutable* attributes are symbols bound to Scheme values in a read-only manner; i.e., they can be accessed but not assigned to.

A module is a Scheme value that is created with the mk-module primitive. Modules may be manipulated, but their attributes cannot be accessed or evaluated until they are instantiated via the mk-instance primitive. The syntax of these two primitives is:

(mk-module ⟨*mutable-attribute-list*⟩ ⟨*immutable-attribute-list*⟩)
(mk-instance ⟨*module-expr*⟩)

Expressions that create modules, such as the mk-module expression above, are denoted as ⟨*module-expr*⟩. Similarly, expressions that create instances are denoted as ⟨*instance-expr*⟩.

The attributes of an instance can be accessed via the attr-ref primitive and assigned to via the attr-set! primitive. Procedures within a module can access sibling attributes via the self-ref primitive and assign to them with the self-set! primitive. (These primitives are explained in more detail in Section 3.1.3.)

Figure 3.1 shows the basic module operations. Figure 3.1 (a) shows a module bound to a Scheme variable fueled-vehicle. The module has one mutable attribute fuel and two immutable attributes: empty?, bound to a procedure which checks to see if the fuel tank is empty, and fill, bound to a procedure that fills the fuel tank of the vehicle to capacity. The fill method refers to an attribute capacity that is not defined within the module but is expected to be the fuel capacity of the vehicle in gallons. In the vocabulary of traditional module systems, the above module exports the three symbols fuel, empty? and fill, and implicitly imports one symbol capacity. (An alternative language design could support explicit importation via explicit interfaces.)

(a)	(define fueled-vehicle (mk-module
	    ((fuel 0))
	    ((empty? (lambda ()
	        (= (self-ref fuel) 0)))
	     (fill (lambda ()
	        (self-set! fuel (self-ref capacity)))))))

(b)	(define encap-fueled-vehicle (hide fueled-vehicle '(fuel)))
	(describe encap-fueled-vehicle)
	    ⟹
	    ((empty? (lambda () (= (self-ref <priv-attr> 0)))) (fill ...))

(c)	(define capacity-module
	    (mk-module ()
	        ((capacity 10)
	         (greater-capacity? (lambda (in)
	            (> (self-ref capacity) (attr-ref in capacity)))))))
	(define vehicle (merge encap-fueled-vehicle capacity-module))

(d)	(define new-capacity (mk-module () ((capacity 25)))) (define new-vehicle (override vehicle new-capacity))
(e)	(define v1 (mk-instance vehicle))

Figure 3.1. Basic module operations. (a) Definition via mk-module, (b) encapsulation via hide, (c) combination via merge, (d) rebinding via override, and (e) instantiation via mk-instance.

3.1.1 Encapsulation

As mentioned earlier, one of the most important requirements of module systems is encapsulation. This is supported by the primitive hide, which returns a new module that encapsulates the given attributes.

(hide ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)

In Figure 3.1 (b), the hide expression creates a new module with an encapsulated fuel attribute that has an internal, inaccessible name. This is shown by the describe primitive, which simply prints out the attributes of a module, as <priv-attr>.

Hiding results in what is known as *object-level* encapsulation, i.e., the hidden attributes of a particular instance of a module are accessible only by self-reference primitives (e.g., self-ref) within that individual instance. They are not accessible externally (e.g., via attr-ref), not even by the incoming parameter of a binary method such as the in parameter of the greater-capacity? method of module capacity-module shown in Figure 3.1 (c). This style of encapsulation is in contrast to the *class-level* encapsulation supported by ADTs, and also the C++ language.

It is important to note that such retroactive encapsulation *shrinks* the interface of a module. As a result, functions expecting an instance of a particular module may not necessarily operate correctly on an instance of the module subjected to a hide operation. This represents the widely accepted notion that an inherited module does not necessarily result in a subtype of the "parent," in effect separating inheritance from subtyping [14].

3.1.2 Combination

The module capacity-module given in Figure 3.1 (c) exports two symbols: capacity, which represents the fuel capacity of a vehicle in gallons, and greater-capacity?, bound to a procedure that determines if the current instance has greater fuel capacity than the incoming argument.

The module encap-fueled-vehicle can be combined with capacity-module to satisfy its import requirements. This can be accomplished via the primitive merge, which has the following syntax:

(merge ⟨*module-expr1*⟩ ⟨*module-expr2*⟩)

The new merged module vehicle in Figure 3.1 (c) exports four symbols and imports none. (One can check if the import requirements of individual modules are satisfied by using introspection primitives described in Section 3.1.6.)

The primitive merge does not permit combining modules with conflicting defined attributes, i.e., attributes that are defined to have the same name. If there are name conflicts, one can use the operator override:

(override ⟨*module-expr1*⟩ ⟨*module-expr2*⟩)

In the presence of conflicting attributes, override creates a new module by choosing ⟨*module-expr2*⟩'s binding over ⟨*module-expr1*⟩'s in the resulting module. For example, the module new-capacity in Figure 3.1 (d) cannot be merged with vehicle since the two modules have a conflicting attribute capacity. However, new-capacity can override vehicle, as shown. This way, immutable attributes can be re-bound, and mutable attributes can be associated with new initial values.

An instance of the vehicle module, such as v1 in Figure 3.1 (e), represents exactly the kind of module interconnectivity that can be specified by the use of import/export operations in traditional module systems.

3.1.3 Attributes and Their Access

Immutable attributes correspond to the fixed "behavior" of the abstraction represented by the module, whereas mutable attributes correspond to its "state." Immutable attributes that are bound to procedures are referred to as *methods*, borrowing from OO programming. Immutable attributes can also be bound to other modules, called *nested modules*, dealt with in Chapter 4. Immutable attributes have the potential to be shared among all instances of the module.

Mutable attributes, on the other hand, are bound to fresh locations upon module instantiation and initialized with the value associated with each attribute. Thus, mutable attributes can never be re-bound as such; they can only be re-initialized, e.g., via override. Structured values (e.g., lists) are only shallow copied, staying consistent with the reference semantics of values in lisp based languages. Values that are stored (or initialized) into mutable attributes are not really "bound" to the attribute; hence a Scheme procedure stored into a mutable attribute cannot access other attributes of the module via self-ref. If this were allowed, then the procedure would effectively become a first-class closure that could be passed around; however, *CMS* provides other means for creating such a closure (via attr-refc below). Similarly, a module can be stored into a mutable attribute, but it cannot access the attributes of the outer module via env-ref (described later).

The attributes of an instance are accessed with the following primitives:

(attr-ref ⟨*instance-expr*⟩ ⟨*attribute-name*⟩ ⟨*arg-expr**⟩)
(attr-refc ⟨*instance-expr*⟩ ⟨*attribute-name*⟩)
(attr-set! ⟨*instance-expr*⟩ ⟨*attribute-name*⟩ ⟨*expr*⟩)

The values of both mutable and immutable attributes are accessed with the primitive attr-ref. If the referenced attribute is a method, it is applied with the given argument(s) and its value returned. Syntactically, accessing the value of a nonmethod attribute via attr-ref is exactly the same as applying a method with no arguments. A method can also be accessed as a first-class closure, without applying it, via the primitive attr-refc. For nonmethod attributes, attr-refc is semantically equivalent to attr-ref. Mutable attributes are assigned with the primitive attr-set!.

A method can access the instance within which it is executing via the expression (self). Thus, a method can access a sibling attribute within the same instance as (attr-ref (self) ⟨*attr-name*⟩). However, encapsulated attributes cannot be accessed in this manner. For this, a method uses the analogous primitives self-ref and self-refc to access the values of attributes, and self-set! to assign to mutable attributes, of the instance within which it is executing.

(self-ref ⟨*attribute-name*⟩ ⟨*arg-expr**⟩)
(self-refc ⟨*attribute-name*⟩)
(self-set! ⟨*attribute-name*⟩ ⟨*expr*⟩)

Accesses via these primitives are called *self-references*, whereas accesses via attr-ref and attr-set! are called *external references*. Figure 3.1 shows examples of the use of some of these primitives.

3.1.4 Abstract Modules

An attribute is called *undefined* if it is self-referenced, or referenced from a nested module, but is not specified in the module. If it is specified, it is called *defined*.

A module is *abstract* if any attribute is left undefined. In keeping with dynamic typing in Scheme, an abstract module can be instantiated, since it is possible that some methods can run to completion if they do not refer to undefined attributes. It is a checked run-time error to refer to an undefined attribute. It should be noted that this goes beyond the normal compilation-oriented policy stated in Section 2.2.1.

3.1.5 Adaptation

Thus far, we have shown how *CMS* supports the notions of traditional module systems. From this point on, we go beyond traditional module systems. In this section, operators to adapt particular characteristics of modules are described. In the following section, ways to find out information about first-class modules and instances are given. Following that, we show how to use all the primitives introduced to simulate composite idioms of OO programming.

Besides hide, there are four other primitives (see Figure 3.2) which can be used to create new modules by adapting some aspect of the attributes of existing modules.

(restrict ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)
(rename ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩ ⟨*to-name-list-expr*⟩)
(copy-as ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩ ⟨*to-name-list-expr*⟩)
(freeze ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)

The primitive restrict simply removes the definitions of the given (defined) attributes from the module, i.e., makes them undefined. An example is shown in Figure 3.2 (a).

(a)	(describe (restrict vehicle '(capacity))) ⟹ ((empty? ...) (fill ...) (greater-capacity? ...))
(b)	(describe (rename vehicle '(capacity) '(fuel-capacity))) ⟹ ((fuel-capacity 10)(fill (... (self-ref fuel-capacity))) ...)
(c)	(describe (copy-as vehicle '(capacity) '(default-capacity))) ⟹ ((capacity 10)(default-capacity 10)(fill (... (self-ref capacity))) ...)
(d)	(describe (freeze vehicle '(capacity))) ⟹ ((capacity 10)(fill (... (self-ref <priv-attr>)))...)

Figure 3.2. Module adaptation operations. (a) Removing an attribute via restrict, (b) renaming an attribute and self-references to it via rename, (c) copying an attribute via copy-as, and (d) statically binding self-references to an attribute via freeze.

The primitive rename changes the names of the definitions of, *and* self-references to, attributes in its argument ⟨*from-name-list-expr*⟩ to the corresponding ones in ⟨*to-name-list-expr*⟩. An example is shown in Figure 3.2 (b). Undefined attributes, i.e., attributes that are not defined but are self-referenced, can also be renamed.

The primitive copy-as copies the definitions of attributes ⟨*from-name-list-expr*⟩ to attributes with corresponding names in ⟨*to-name-list-expr*⟩. The *from* argument attributes must be defined. An example is shown in Figure 3.2 (c).

The primitive freeze statically binds self-references to the given attributes, provided they are defined in the module. Freezing the attribute capacity in the module vehicle causes self-references to capacity to be statically bound, but the attribute capacity itself is available in the public interface for further manipulation, e.g., rebinding by combination. (This effect is similar to converting accesses to a virtual C++ method into accesses to a nonvirtual method. The difference is that C++ allows nonvirtual methods to be in the public interface of a class — the general philosophy here is that all public attributes are rebindable, or virtual, like in Smalltalk.) As shown in Figure 3.2 (d), frozen self-references to capacity are transformed to refer to a private version of the attribute. Operationally, the binding of the private version is shared with the public version, as long as the public version is not re-bound to a new value via overriding. This implies that frozen references to mutable attributes are always shared, since mutable attributes can never be rebound; they can just be initialized to new values.

In Figure 3.3, all the module adaptation operators are shown pictorially. At the top left is a box representing the given module with an inner box representing the method meth. Self-references to meth from within the rest of the module are shown bundled into one line pointing to meth. At the top center, the inner shaded box labeled <meth> indicates an encapsulated meth attribute. At the top right, references to the undefined method meth are shown bundled into a line pointing out of the module.

The above module manipulation primitives are applicative, in the sense that they return new modules without destructively modifying their arguments. De-

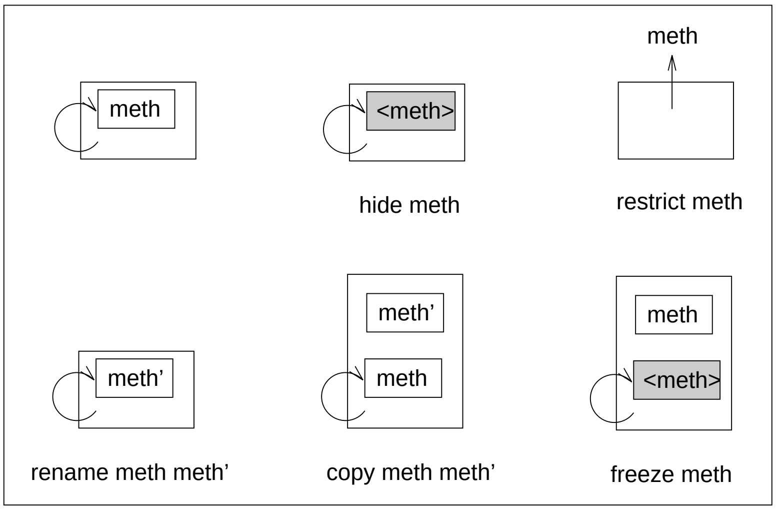


Figure 3.3. Pictorial representation of adaptation. Given a module with a method meth and self-references to it, this figure shows the effect of various adaptation operators. Shaded boxes with <meth> indicate encapsulated meth attributes.

structive versions of the operators could permit one to express composite module operations without compromising efficiency by making unnecessary copies. However, such operations could result in disastrous behavior. For example, an instance that introspectively queries for its module via module-of (described in the next section) could obtain a different module from the one from which it was instantiated. As a result, destructive module operations are not supported.

3.1.6 Introspection

There are several primitives available for determining various kinds of information about modules and instances. Some of them are:

(attrs-of ⟨*module-expr*⟩)
(mutable-attrs-of ⟨*module-expr*⟩)
(module-of ⟨*instance-expr*⟩)
(defined? ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)
(self-refs-in ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)
(conflicts-between ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)

Some examples are shown in Figure 3.4. The names of the publicly accessible attributes of a module are accessible via the attrs-of primitives. For example, the mutable attributes of a module can be encapsulated with the expression in Figure 3.4 (a).

(a)	(hide vehicle (mutable-attrs-of vehicle))
(b)	(defined? vehicle (self-refs-in vehicle greater-capacity?))
(c)	(let ((conflicts (conflicts-between mod1 (attrs-of mod2)))) (rename mod1 conflicts (prepend "super-" conflicts)))

Figure 3.4. Introspection operations. (a) Hiding mutable attributes via mutable-attrs-of, (b) checking if a method will run to completion using defined? and self-refs-in, and (c) renaming conflicting attributes via conflicts-between.

The primitive defined? is used to determine if an attribute is defined in a module. It returns #f if any one of the given attribute names is undefined in the given module. If all of them are defined, it returns the incoming list of attribute names. The primitive defined? can actually be implemented in terms of attrs-of.

The module from which an instance was created can be obtained with the primitive module-of. Thus, (module-of (self)) is similar to self class in Smalltalk, like current in Eiffel [57] and myclass given in Canning et al. [14].

It is sometimes useful to know the names of public attributes that are self-referenced within a method. The primitive self-refs-in returns a flat list comprising the set of all the self-referenced public attributes within the bindings of the given attribute names. Argument attribute names that are nonexistent or bound to nonmethod values are ignored. For example, to determine if a method will execute without run-time errors relating to locally undefined public attributes (private attributes are always defined), one can evaluate the expression in Figure 3.4 (b).

The primitive conflicts-between returns a list of attribute names that are defined in the given module and also exist in the given list. For example, all the attributes of a module that conflict with another module can be *renamed* by using the expression in Figure 3.4 (c). Renaming is an adaptation operator described in the following section.

Two other primitives not enumerated above are module? and instance? which are predicates that tell if their argument value is a module or instance respectively.

The introspective operations given above do not permit access to any aspect of private attributes of modules. Being meta-level primitives, they do permit exposing some details of method implementations, such as self-references. However, it has been argued in the literature [52] that the self-reference dependencies of methods are indeed an aspect of their inheritance interface, rather than their implementation.

3.2 Single Inheritance

From the programmer's point of view, it is necessary to know not only the available constructs in a language but also the intent and usefulness of the constructs. Thus, it is necessary to show how *CMS* can emulate composite notions of advanced modular programming, such as OO programming. In this section, we show how *CMS* supports several styles of single inheritance. In Section 3.4, styles of multiple inheritance are illustrated.

3.2.1 Super-based Single Inheritance

Super-based single inheritance is illustrated in Figure 3.5. Single inheritance systems such as Smalltalk have the notion of a class consisting of methods and encapsulated instance variables. In these systems, it is possible to specify a class declaration similar to that shown in Figure 3.5 (a). (The define-class construct will be explained below.) In this example, the attribute fuel is intended to be encapsulated as an instance variable and the Scheme constant #f (false) indicates that the class has no superclasses. Such a class declaration is equivalent to writing a mk-module expression and hiding the fuel attribute of the resultant module.

Given such a vehicle module, a land vehicle can subsequently be defined in such systems by specifying the way in which land vehicles incrementally differ from vehicles. That is, a subclass land-vehicle of vehicle can be specified in a manner similar to Figure 3.5 (b). In this definition, a new immutable attribute wheels is added, and the display binding is overridden with a method that accesses the shadowed method as (self-ref super-display).

The crucial effect of inheritance is the appropriate rebinding of self-references in the superclass to redefined attributes in the subclass. The question is how to achieve this in *CMS*. For the vehicle example, a new module that captures the

(a)

```
(define-class vehicle #f
      ((fuel 0))
      ((capacity 10)
       (fill (lambda ()
                  (self-set! fuel (self-ref capacity))
                  (self-ref display)))
       (display (lambda () (format #t "fuel = ~ a (capacity ~ a) "
                     (self-ref fuel) (self-ref capacity))))))
```

(b)

```
(define-class land-vehicle vehicle
  ()
  ((wheels 4)
   (display  (lambda ()
                  (self-ref super-display)
                  (format #t "wheels = ~ a " (self-ref wheels))))))
```

(c)

```
(define land-vehicle
     (hide (override (copy-as vehicle '(display) '(super-display))
                     (mk-module ()
                          ((wheels 4)
                           (display (lambda ()
                                         (self-ref super-display)
                                         (format ... (self-ref wheels)))))))
           '(super-display)))
```

Figure 3.5. Super-based single inheritance. (a) The superclass vehicle, (b) the subclass land-vehicle, and (c) the module expression that the macro in box (b) expands to.

characteristics of a land vehicle is created, and this must be combined with the vehicle module. Consider the following cases:

1. *Simply override.* The subclass cannot simply override the superclass, since in that case, the superclass display will be wiped out.

2. *Rename and override.* If the superclass display is renamed to super-display, self-references to display are also renamed. Thus, the self-reference to display in the fill method will never execute the subclass' display.

3. *Copy and override.* If the superclass display is copied to super-display and then

overridden, self-references to display in the superclass will execute the subclass' display, as desired. However, the resultant module will have an extra method super-display in its public interface.

4. *Copy, override, and hide.* This is the correct solution, shown in Figure 3.5 (c).

One can write a macro in *CMS* to translate define-class expressions such as those for vehicle and land-vehicle into module expressions. In fact, a library of several such useful macros accompanies *CMS*. One macro for single inheritance accepts the following syntax:

(define-class $\langle name \rangle$ $\langle super \rangle$ $\langle inst\text{-}var\text{-}list \rangle$ $\langle method\text{-}list \rangle$)

This macro automatically finds conflicting attributes between modules by using the introspective primitive conflicts-between and uses the expression in Figure 3.5 (c) to achieve single inheritance.

The general form of the module expression shown in Figure 3.5 (c) turns out to be a useful idiom in *CMS*. It can be used for expressing other effects such as prefix-based inheritance, wrapping, and mixin combination, described later. We shall refer to this form as the *copy-override-hide* idiom.

3.2.2 Prefixing

The programming language Beta [51] supports a form of single inheritance called *prefixing*, which is quite different from the single inheritance presented in Section 3.2.1.

In prefixing, a superclass method that expects to be re-bound by a subclass definition uses a construct called inner somewhere in its body. Within instances of the superclass, calls to inner amount to null statements, or no-ops. Subclasses can redefine the method and, in turn, call inner. Within subclass instances, the superclass method is executed first; then the subclass' redefinition is executed upon encountering the inner statement. It is easy to see that this mechanism ensures that a method redefinition is an extension of the original method, rather than a replacement.

The above effect can be achieved in *CMS* with the macro define-prefix, illustrated with the vehicle example in Figure 3.6. Figure 3.6 (a) focuses on the display method of the vehicle class. The expression (self-ref inner-display) corresponds to the inner construct. This class definition expands to the module expression shown in 3.6 (b), where a dummy inner-display attribute is merged in. In fact, the define-prefix macro adds such a dummy attribute (prepended with inner-) for every immutable attribute in the definition.

A subclass land-vehicle of vehicle is defined in Figure 3.6 (c), which expands to an expression similar to the one in 3.6 (d). In this expression, assume that a module with the subclass characteristics is first merged in with a dummy inner-display to produce the module land-veh-chars. This module's display method is then copied as sub-display and overridden with the superclass in which the dummy inner-display attribute is removed and references to it are renamed to sub-display. Lastly, sub-display is hidden away so that there is only one display method in the resultant.

(a)	<pre>(define-prefix vehicle #f
	    (...)
	    (...(display (lambda () ... (self-ref inner-display) ...))))</pre>

(b)	<pre>(define vehicle (mk-module (...)
	                  (...
	                  (display (lambda () ... (self-ref inner-display) ...))
	                   (inner-display #t))))</pre>

(c)	<pre>(define-prefix land-vehicle vehicle
	   (...)
	   (...(display (lambda () ... (self-ref inner-display) ...))))</pre>

(d)	<pre>(define land-vehicle
	   (hide (override (copy-as land-veh-chars '(display) '(sub-display))
	                (rename (restrict vehicle '(inner-display))
	                        '(inner-display) '(sub-display)))
	         '(sub-display)))</pre>

Figure 3.6. Prefix-based inheritance. (a) A vehicle prefix, (b) expansion of the inner construct, (c) a land vehicle subclass, and (d) the module expression to combine prefixes.

This example also uses the copy-override-hide idiom introduced in Section 3.2.1. The difference here is that the adapted superclass overrides the subclass as opposed to the reverse in Section 3.2.1. Indeed, this is the difference between prefix-based and super-based forms of single inheritance. This can be contrasted side by side pictorially in Figure 3.7 (the diagramming conventions used are the same as in Figure 3.3).

3.2.3 Wrapping

The notion of wrapping enables one to interpose a piece of code (the "wrapper") between a method and its callers. Wrapping, similar to the CLOS notion of :around methods, is useful in many contexts. In fact, wrapping method definitions can be

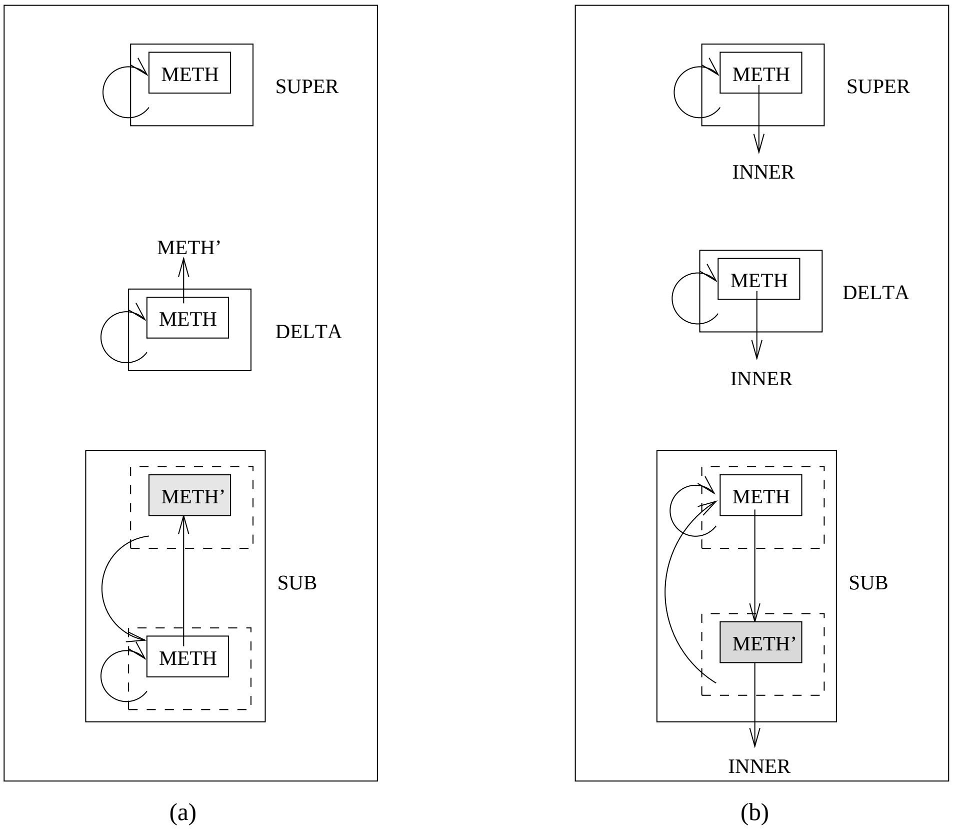


Figure 3.7. Pictorial representation of single inheritance. (a) Super-based: (hide (override (copy-as SUPER METH METH') DELTA) METH'), and (b) prefix-based: (hide (override (copy-as DELTA METH METH') (rename SUPER INNER METH')) METH')

used to simulate :before and :after methods of CLOS as well, since new code can be interposed before or after the call to the old code. It is easy to wrap method definitions using the copy-override-hide idiom shown earlier. *CMS* provides a macro called wrap-method to achieve this effect, but we shall omit its description here to conserve space.

A more interesting and less often explored effect is to wrap self-referenced *calls to* particular methods, as illustrated in Figure 3.8. Say we have a module veh-sim shown in Figure 3.8(a), which is intended to be combined with the vehicle module. Its method sim-fill calls the undefined method fill upon some condition fill-condition. Say we want to count the number of calls to fill that sim-fill makes. We do not want to wrap the method fill in vehicle, since we want to count only calls from sim-fill. Also, we cannot wrap the sim-fill method to do this, since every call to it does not necessarily result in a call to fill, due to the fill-condition test.

Thus, we need to wrap *calls to* fill from the veh-sim module using the wrap-call

(a)	<pre>(define veh-sim (mk-module (...)
	                    ((sim-fill (lambda (v)
	                                  (if (fill-condition v)
	                                      (self-ref fill)))))))</pre>

(b)	<pre>(define counted-veh-sim
	   (let ((count-sim (merge veh-sim (mk-module ((count 0)) ()))))
	     (wrap-call count-sim fill
	              (lambda ()
	                 (self-set! count (+ (self-ref count) 1))
	                 (self-ref fill)))))</pre>

(c)	<pre>(hide (merge (rename count-sim '(fill) '(wrap-fill))
	              (mk-module ()
	                 ((wrap-fill (lambda ()
	                                (self-set! count (+ (self-ref count) 1))
	                                  (self-ref fill))))))
	      '(wrap-fill))</pre>

Figure 3.8. Wrapping calls to methods. (a) A vehicle simulation module veh-sim, (b) wrapping calls to fill from veh-sim using the wrap-call macro, and (c) the module expression that wrap-call expands to.

macro shown in box (b). We add a mutable attribute count to veh-sim and wrap its calls to fill to increment the counter. The module expression that the wrap-call expands into is given in box (c). In this expression, we first rename the undefined attribute fill to wrap-fill, thus changing the self-references correspondingly. We then merge in a wrap-fill method that increments count and calls the old fill method in the resulting module.

3.3 Idioms in *CMS*

The general form of the expression in Figure 3.8 (c) is another useful idiom in *CMS* and will be referred to as the *rename-merge-hide* idiom. The distinction between the copy-override-hide idiom and the rename-merge-hide idiom is worth exploring. Figure 3.9 pictorially shows the use of these idioms for method definitions in the first row and method calls in the second. In the figure, shaded boxes represent hidden methods.

For method definitions, both the idioms are used when a method METH of M1

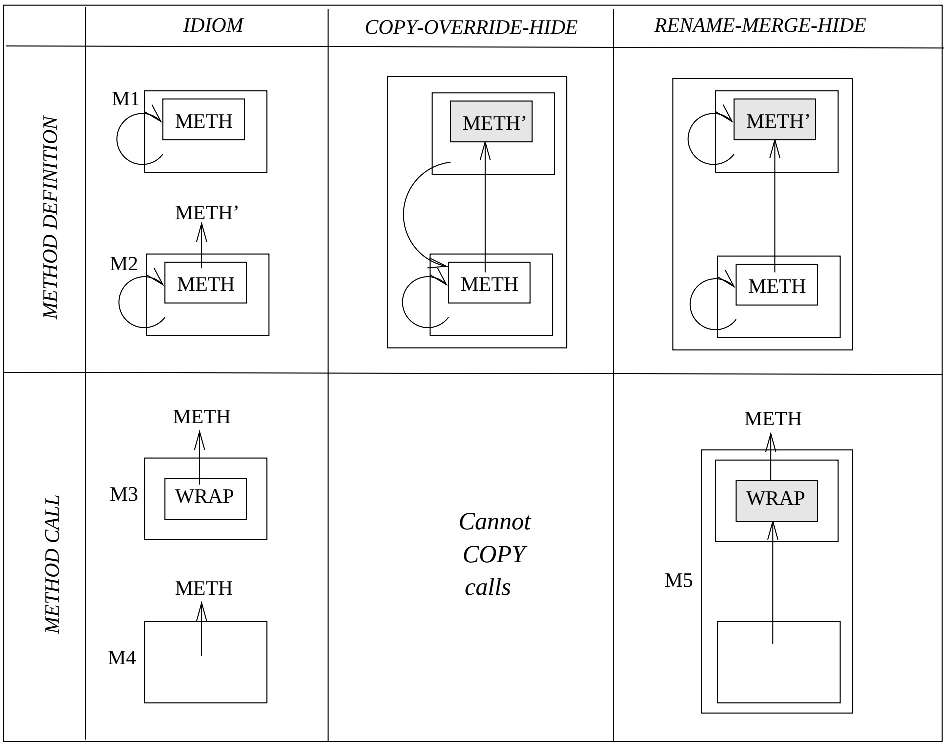


Figure 3.9. Idioms in *CMS*.

is being redefined by M2, and the old definition of the method is referred to in the redefinition as METH'. The difference is that copy-override-hide is used when M1's references to METH are to refer to the new METH in the combined module. Rename-merge-hide is used when M1's references are to refer to the old definition renamed as METH', and M2's references are to refer to the redefinition.

As an example scenario, rename-merge-hide is not appropriate to achieve the right effect of single inheritance. For example, in Figure 3.5(c), renaming vehicle's display method instead of copying it would not work, since in that case self-references to display in vehicle would also be renamed — we want self-references in the superclass to refer to the new, rebound display method.

For method calls, only the rename-merge-hide idiom applies, since undefined attributes cannot be copied. In Figure 3.9, module M4 has a call to METH which is wrapped to produce M5 as shown. An example was given in Section 3.2.3.

3.4 Multiple Inheritance

We have seen in Section 3.2 how to express the creation of a subclass from a single superclass. With multiple inheritance (MI), there is the additional problem of how to compose the superclasses by resolving conflicts and sharing attributes between them. Typically, a language supporting multiple inheritance makes available to the programmer a small number of choices for attribute sharing and conflict resolution. The advantage of OO programming with operator-based inheritance is that the programmer has numerous options for, and fine-grained control over, decisions taken while combining multiple modules.

The most important advantage of operator-based inheritance as given here is that the programmer has *explicit control* over the various aspects of combination of modules. This is in contrast to many existing languages, which provide default language behavior for aspects of multiple inheritance. Two examples are default linearization of ancestors and sharing of ancestors common to multiple paths. Depending upon such default rules can cause unexpected consequences if the inheritance relationship between classes (which is an implementation detail) is changed. With explicit, fine-grained control over aspects of multiple inheritance as supported by compositional modularity, inheritance relationships can be changed as desired, as long as the interface of the inherited module is not changed. This makes the approach more compositional.

3.4.1 Mixins and Linearized MI

Consider the case of linearized multiple inheritance as in Flavors and Loops, where the graph of ancestor classes of a class are linearized into a single inheritance hierarchy. Each of these languages specifies a different default rule for the linearization of ancestor classes. For example, both these languages do a depth-first, left-to-right traversal of ancestor classes up to join classes, i.e., classes that are encountered more than once, which get traversed on their first visit in Flavors and last visit in Loops.

It has been argued that currently used linearizations do not ensure that "the inheritance mechanism behaves 'naturally' relative to the incremental design of the inheritance hierarchy" [27]. Moreover, changing the inherited superclass of a class (an implementation detail) can change the computed linearization of superclasses, producing a completely different behavior than before.

Perhaps it is better to let the programmer select the precedence order of superclasses as dictated by individual applications. In the case of CLOS, a programmer with considerable expertise can use the meta-object protocol of the language and adapt the default rule. In contrast, programming with operator-based inheritance gives the programmer direct control over combination, as shown in Figure 3.10.

Say we want to create modules for land vehicles and sea vehicles as subclasses of `vehicle`. We can define modules with the characteristics of land vehicles (number of wheels) and sea vehicles (surface vessel or submarine) as shown in box (a). In these modules, one can think of the expression `(self-ref super-display)` as being the equivalent of `call-next-method` in CLOS. Abstract modules such as these are sometimes called "mixins" — reusable abstractions that require other abstractions in order to be usefully applied. Such abstractions have been characterized as functions from classes to classes [8]. However, the approach of operator-based

(a)

```
(define land-veh-chars (mk-module ()
                         ((wheels 4)
                          (display (lambda () (self-ref super-display)
                                     (format #t "wheels = ~ a " (self-ref wheels)))))))
(define sea-veh-chars (mk-module ()
                        ((surface #t)
                         (display (lambda () (self-ref super-display)
                                    (format #t "surface = ~ a " (self-ref surface)))))))
```

(b)

```
(define-subclass land-vehicle (land-veh-chars vehicle))
(define-subclass sea-vehicle (sea-veh-chars vehicle))
(define-subclass amphibian (land-veh-chars sea-veh-chars vehicle))
```

(c)

```
(define amphibian
    (hide (override (copy-as vehicle '(display) '(super-display))
            (hide (override (copy-as sea-veh-chars '(display) '(super-display))
                            land-veh-chars)
                '(super-display)))
          '(super-display)))
```

Figure 3.10. Linearized multiple inheritance.

inheritance given here uniformly treats all aspects of inheritance as operations over modules, as was first developed in [7].

With the definitions in box (a), we can create land-vehicle and sea-vehicle "subclasses" of vehicle as shown in box (b). This is achieved using the copy-override-hide idiom (Figure 3.9), but with the macro define-subclass which accepts a slightly different syntax. Similarly, we can "chain" the creation of subclasses so that the call to super-display in each class calls the display method of the next lower precedence superclass. Thus, we can create an amphibian class that inherits both the characteristics of land and sea vehicles. The define-subclass macro for amphibian expands to the module expression shown in box (c) (note the cascaded use of the copy-override-hide idiom) and extends to an arbitrary number of superclasses, as desired.

3.4.2 MI with No Common Ancestors

Let us now consider the case of multiple superclasses that are not linearized and have no common ancestor. Say we have a module color defined as in Figure 3.11(a). We can combine color with the module land-vehicle shown earlier into car-class, as shown in Figure 3.11 (b). This expression uses the rename-merge-hide idiom introduced in Section 3.3. The method display that conflicts in the "superclasses" vehicle and color is renamed in each and the superclasses are merged together. A new module that defines a display method that calls the renamed display methods is then merged in to create the desired car-class. This example can be extended to more than two superclasses and can be automated via a macro that uses the introspective primitive conflicts-between to rename attributes.

The rename-merge-hide idiom works fine for this example, since there are no self-references to the renamed attribute in the superclasses. However, the right effect of inheritance can only be obtained with copy-as, so that self-references in the superclasses are not changed, followed by a merge, so that accidental conflicts between superclasses do not get quietly re-bound. The problem with copying conflicting attributes and merging is that the conflicts will still persist. This can be remedied by restrict'ing (Section 3.1.5) after copying, and then merging and hiding.

(a)

```
(define color
     (mk-module
          ((color 'white))
          ((set-color   (lambda (new-color) (self-set! color new-color)))
           (display     (lambda () (format #t "color = ~ a" (self-ref color))))))
```

(b)

```
(define car-class
   (hide (merge (merge (rename color '(display) '(color-display))
                       (rename land-vehicle '(display) '(vehicle-display)))
                (mk-module ()
                    ((display (lambda () (self-ref vehicle-display)
                                         (self-ref color-display))))))
         '(color-display vehicle-display))
```

Figure 3.11. Multiple inheritance with no common ancestors.

There is some similarity between such a copy-restrict-merge-hide operation and the copy-override-hide idiom.

3.4.3 MI with Common Ancestors

In the case of superclasses with a common ancestor, such as in the "diamond" problem of multiple inheritance, the situation gets more complex. In this case, the attributes of the common ancestor are clearly conflicting in the direct superclasses of the inheriting class. Furthermore, there is the choice of inheriting either a single copy or multiple copies of mutable attributes from the common ancestor.

Consider the case of a language providing a default rule of sharing the attributes of the common ancestor. Now, if the inheritance structure is changed, say by inheriting from two separate classes instead of the common ancestor, then there is the potential for attribute clashes. Thus, the approach of compositional modularity is to require explicit specification of every aspect of combination of modules.

To illustrate, consider the two previously given modules land-vehicle and sea-vehicle which have each inherited from the vehicle module. Say we want to create an amphibian module that inherits from these two modules but needs two copies of the fuel attribute to model two different kinds of fuels for amphibians. This can be achieved with the expression in Figure 3.12. In this example, the fuel attribute is renamed for each type of module. The two modules are then overridden since the conflicting attributes capacity and fill are known to be identical, and the method display will be overridden in the final module. A new display method that displays all the attributes in an appropriate way is included in the final composition to get the desired module.

```
(override (override (rename land-vehicle '(fuel) '(land-fuel))
                    (rename sea-vehicle '(fuel) '(sea-fuel)))
          (mk-module ()
            (display (lambda ()
                       (format ... (self-ref land-fuel) (self-ref sea-fuel) ... ))))))
```

Figure 3.12. Multiple inheritance with common ancestors.

The distinction between programming with first-class modules and the operational style more often found in OO languages is illustrated by this example. Problems of conflicts and sharing clearly manifest themselves and compel the programmer to resolve them as the particular situation demands using introspection and inheritance operators. For example, conflicts between superclasses can be inspected with `conflicts-between`, and superclasses can be overridden in some appropriate order to resolve attribute conflicts. If multiple copies of mutable attributes from the common ancestor are desired, they can be renamed within each superclass, as shown in the example above. However, if desired, the burden of resolving conflicts in each individual case can be removed by writing macros that perform a user-chosen method of composition.

3.5 Related Work

This chapter has already shown the relationship of *CMS* constructs to inheritance mechanisms found in several OO languages. In this section, its relationship to various other module models is given.

Most classical module systems such as SML [76], and Ada [39] do not support first-class modules. Some ML systems [35], however, do support higher-order functors, although not as first-class run-time values. Also, these module systems closely associate a static type system with the module system. From the viewpoint of compositionality and reuse, these systems are quite limited, as explained in Section 2.1.1.

An early effort to incorporate first-class modules into a language was in Pebble [11]. There, the uniformity and expressive power obtained by using first-class modules were recognized. More recently, many other languages such as FX [72] and Rascal also support first-class modules.

Support for explicit interfaces separate from implementations is a frequent feature of module systems [39, 59, 76, 25, 67]. This is known to support large-scale programming, since clients can be written (and compiled) based on a module's interface. The module's implementors can then associate (possibly multiple) implementations with the interface.

The *CMS* language presented here does not explicitly support interfaces. However, interfaces can be built up dynamically, by specifying a module's public interface attributes and providing dummy error methods. Subsequently, implementations for this interface can be combined with it via override and private attributes encapsulated via hide.

Several module systems have been developed for the Scheme programming language. Curtis and Rauen's system [25] supports explicit interfaces and modules with import and export specifications. Tung's [77] system additionally supports a notion of renaming conflicting imports, as well as dynamic binding of imported attributes. These systems do not support composition and, in effect, provide little more than the functionality described in Sections 3.1.1 and 3.1.2.

Lisp packages [73] are namespaces that map strings to symbols. Symbols exported from one package may be imported by another. The Scheme 48 module system [67] is a more sophisticated namespace manipulation mechanism, in that it supports multiple instantiation of modules into packages, as well as explicit interfaces. However, there are no mechanisms for composition or adaptation in either of the above systems.

Some Scheme implementations support first-class environments. In these systems, first-class environments can be dynamically created and extended, and expressions evaluated within them. The environment at any point can also be captured by using a special primitive, such as the-environment. However, the only useful operation defined on environments is eval.

Reflective operations on first-class environments have been proposed in the language Rascal [41]. In this language, one can construct an environment with lexical and public bindings, *reify* the environment into a data structure, and subsequently *reflect* the data structure back into an environment. Only public bindings are visible when environments are reflected. The effect of the reification operation can be constrained as desired by using *barriers*. Programmers can also unbind names in an environment by using a *restrict* operation. Using the above operations, programmers can achieve many effects of modularity in Rascal.

The goals of *CMS* and Rascal are very similar, although the approaches used are entirely different. Rascal uses the approach of reflection, whereas *CMS* uses notions of generator manipulation. Specifically, *CMS* does not support reflection and reification of environments. On the other hand, Rascal does not support a wide array of name conflict resolution mechanisms (e.g., rename), static binding (freeze), and retroactive encapsulation (hide).

More significantly, *CMS* differs from Rascal in that *CMS* is based on compositional modularity, which is shown in the rest of this dissertation to apply not only to dynamic languages but to several other systems. In particular, static typing is not explored in Rascal. Also, compositional nesting, as given in the following chapter, is unique to *CMS*.

A popular language family for OO programming with Lisp is the CLOS family of languages [45, 50]. CLOS supports a quite different model of OO programming from the one described here, with multiple-dispatch, generic functions, but much weaker encapsulation. *CMS*, on the other hand, supports only single dispatch. CLOS also supports a protocol to interact with its meta-architecture. Dexterity of multiple inheritance as given in Section 3.4.1 was a primary practical consideration in the design of the CLOS MOP.

CMS must also be compared to the approach of open implementations. Meta-object protocols (MOPs) for CLOS [47] and C++ [20] expose the OO implementation of the language processor to the programmer, via a controlled protocol. Many aspects of the language's implementation, such as the mechanisms of inheritance, object data layout, and method dispatch, are controllable via such MOPs.

As far as inheritance is concerned, *CMS* provides the programmer flexibility similar to that provided by MOPs, for all practical purposes. *CMS* provides this flexibility by supporting a small set of well-designed primitives that can effectively control the desired aspects of inheritance, whereas MOPs do so by opening up the meta-architecture implementation to direct user programming. As a result, our approach does not give the user the full power of altering a language's behavior as a full-blown MOP can. However, it is not clear that completely opening up the implementation will not result in much complexity and inadvertent abuse by users.

Finally, inheritance of first-class modules must be compared with class-less programming with prototypes and delegation as in SELF [19]. Class-less programming languages allow individual objects to inherit from (or delegate to) other objects. Thus, there is no notion of classes instantiable into objects. The crucial characteristic of delegation is that inherited objects are *shared* parts of inheriting objects, whereas in ordinary inheritance, inherited classes are *copied* into inheriting classes. Although it is true that instances of nested classes in ordinary inheritance share a copy of the enclosing instance (as explained in the next chapter) in the same manner as delegation, there is still no notion of dispatching to the shared parent in nesting. It should also be pointed out that delegation-based languages such as SELF do (or at least used to) support sophisticated mechanisms of inheritance of their own, such as prioritized multiple inheritance.

3.6 Summary

This chapter has presented a realistic imperative language design that embodies compositional modularity and has illustrated typical programming styles and idioms in a language supporting compositional modularity.

In the language presented here, called Compositionally Modular Scheme (*CMS*), modules are manipulated with a suite of operators that individually achieve effects such as encapsulation, combination, sharing, and introspection. Compositional module operators enable one to combine modules in a number of ways that enable reuse, while preserving encapsulation.

The above language is expressive and flexible enough to model most previously existing techniques of OO programming. We have shown by examples that the language can emulate an unprecedentedly broad array of idioms such as single, prefix-based, mixin-based, and multiple inheritance, abstract classes, and wrapping of method definitions and calls. Thus, the language provides mechanisms to support all of the above but does not enforce any one inheritance policy. In effect, this language represents a unification of the design space of dynamic, single-dispatch, OO programming languages.

This model of modularity has been smoothly integrated into the programming language Scheme while keeping with its original design philosophy that "... a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today" [22].

CHAPTER 4

MODULE NESTING

Software development by decomposition naturally leads to hierarchical nesting of modules. Thus, support for nesting is an important requirement for module systems. In this chapter, the notion of module nesting is developed and integrated with the notion of compositionality.

Previous module models support *lexical* nesting, in which nested modules cannot be developed independent of the (outer) nesting modules. This compromises the modularity and reuse of the nested module. In compositional modularity, separately developed modules may be retroactively composed with other modules to achieve hierarchical nesting.

Nesting constructs in the language *CMS* are described in Section 4.2. One can either directly lexically nest modules or retroactively nest them via a composition operation. In effect, direct nesting can be regarded as syntactic sugar for separately defining a module and retroactively nesting it.

Section 4.3 illustrates the many applications of nested modules such as name-space control, sharing, inheritance hierarchy combination, and the notion of manager modules, within the context of the language *CMS*.

Finally, the semantic concept of *closed* generators is proposed in Section 4.4 to model retroactive nesting. Ways to integrate this concept with earlier semantics of generators, and their static typing, are discussed.

4.1 Goals and Benefits

In addition to the general requirements of compositional modularity, a system supporting nested modules must meet the following goals related specifically to module nesting:

1. *Separate processability.* A fundamental requirement of modularity is that individual modules must be specifiable independent of a particular context. Thus, although a module is to be nested within another, it must be possible to specify it independent of the other. Additionally, it might be desirable to have the interface of a *nestable* module with a surrounding context be explicitly specified. (This is distinct from specifying the interface of the module with respect to its "siblings" at the same level.)

2. *Composability.* Any independently developed module must be composable not only at the same level, but also in a hierarchical sense. That is, one must be able to *retroactively* nest a separately developed module within any other that has a compatible interface.

3. *Static safety.* Properties such as type compatibility between a nestable module and its outer module must be statically checked for safety.

4. *Namespace independence.* There must be support for accessing similarly named values in the local and nonlocal environments. Such explicit access has the benefit that it eliminates the possibility of accidental reference to nonlocal names.

It may at first seem that the modularity requirements of lexical nesting and separate processability are at odds with each other: how can one develop a nested module that depends on its lexical environment separately from its environment? The solution is to have modules *abstract over* its entire lexical environment, as outlined in Section 2.1.4. A more precise description of this idea is given in Section 4.4.

The benefits of compositional nested modules derives from the ability to retroactively nest modules:

1. Nested modules can be independently developed, thus supporting team development even in the presence of hierarchical structure. This, along with

the notion of static typechecking of retroactive nesting, supports large-scale software development.

2. Without retroactive nesting, nested modules are directly lexically nested within other modules, thus imposing a permanent nesting relationship and restricting reuse of the nested module. In contrast, a compositional nested module can be embedded into, and thus reused in, any module that generates a conforming environment. The conformance can be statically checked.

3. One can control the namespace of modules themselves. That is, groups of interacting modules within an application can be retroactively bundled into a separate namespace, so that all of them are not in the global application namespace.

The general approach of compositional modularity is to provide facilities to compose independently developed modules in various possible (and desirable) ways. Thus, it is natural in this framework that modules can be composed to achieve hierarchical nesting as well.

In a sense, support for compositional nested modules is analogous to support for mixin-based inheritance in OO languages. It is important for languages that support incremental programming via inheritance also to support the use of the increments themselves (called mixins) as independent reusable abstractions. Similarly, a language that supports compositional programming as well as nested modules must also support nested modules as independent composable abstractions.

In the following section, we introduce the concepts and applications of nesting via the language *CMS*.

4.2 Nesting in *CMS*

In *CMS*, a module can be either directly lexically nested (Section 4.2.1), or it can be nested after the fact (Section 4.2.2). In either case, a module that is bound to an immutable attribute of another module is referred to as a *nested module*. As argued in Section 3.1.3, a module stored within a mutable attribute is not properly called a nested module.

The methods of modules can refer to bindings in their surrounding environment using the following primitives, which refer to the given name in a lexically surrounding scope that has a binding for that name.

```
(env-ref ⟨attribute-name⟩ ⟨arg-expr*⟩)
(env-refc ⟨attribute-name⟩)
(env-set! ⟨attribute-name⟩ ⟨expr⟩)
```

These three primitives serve functions analogous to the three self-reference primitives given in Section 3.1.3. Thus, names in surrounding scopes are accessed explicitly via these environment reference primitives. It is a checked run-time error to refer to a name that does not have a binding in some surrounding scope.

4.2.1 Lexical Nesting

Modules follow static scoping rules just like the rest of Scheme. The environment of a module is determined by the lexical placement of the mk-module expression that creates it. Figure 4.1 shows examples of module nesting. In Figure 4.1 (a), type1 and type2 are nested modules whose fill methods refer to the capacity attribute of the outer module. Individual vehicles are represented by instances of the nested modules.

(a)	(define vehicle-category
	    (mk-module ()
	        ((capacity 10)
	         (type1 (mk-module (...) ((fill (lambda... (env-ref capacity)...)))))
	        (type2 (mk-module (...) ((fill (lambda... (env-ref capacity)...))))))))
	(define mycategory (mk-instance vehicle-category))
	(define v1 (mk-instance (attr-ref mycategory type1)))

(b)	(define veh-type (mk-module (...)
	                ((fill (lambda ... (env-ref capacity) ...)))))
	(define new-vehicle-category
	    (nest type3 veh-type vehicle-category))

Figure 4.1. Examples of nested modules. (a) Lexical nesting, and (b) retroactive nesting.

The mk-module expressions for these nested modules are evaluated at the time vehicle-category is instantiated. Thus, nested modules have an instance of their surrounding module as their environment and are bound to their environment at the time of instantiation of the outer module (described more precisely in Section 4.4.2). Hence, lexical scoping is maintained regardless of whether nested modules are moved to and combined in other environments with other nested modules created in yet other environments. This is analogous to the creation and manipulation of first-class closures, i.e., procedures with environment references, in Scheme.

With static scoping, a module and its nested modules interact in nonobvious ways. Changes to a module's attributes, e.g., via rename, freeze (static binding), and hide, result in modifications to the environment of nested modules and must be tracked by environment references. For example, renaming an attribute of a module will cause the renaming of environment references in nested modules.

4.2.2 Retroactive Nesting

Module nesting can be done retroactively via the primitive nest.

(nest ⟨*attr-name*⟩ ⟨*nested-module-expr*⟩ ⟨*outer-module-expr*⟩)

This primitive returns a new module containing an attribute ⟨*attr-name*⟩ bound to the module ⟨*nested-module-expr*⟩ within the given outer module. An example is shown in Figure 4.1 (b).

In an interactive language such as *CMS*, modules that contain env-ref's can be specified in the "top-level" environment. However, since modules abstract over their environments, env-ref's in such modules are not automatically bound to names occurring in the top-level environment. Instead, when such a module is instantiated via mk-instance, its environment is bound to the Scheme environment at the point of instantiation.

The above semantics, however, is akin to dynamic scoping. To obtain the effect of static scoping for top-level modules, the following primitive is used:

(bind-env ⟨*module-expr*⟩ [⟨*environment*⟩])

This expression binds the environment of the module produced by ⟨*module-expr*⟩ to the optional argument ⟨*environment*⟩, which by default is the Scheme environment at that point. However, once fixed in this manner, env-ref bindings within the module cannot be changed; thus further retroactive nesting will have no effect on its environment references.

4.3 Applications of Nesting

4.3.1 Name-space Control and Sharing

A major use of block structure arises from a module providing a local name-space for nested modules. This helps control problems associated with flat global name-spaces, such as name pollution and accidental name conflicts.

Furthermore, a module can serve as a shared data repository for nested modules. In Figure 4.1 (a), the attribute capacity is shared among all instances of type1 and type2 modules. Similarly, a mutable attribute can hold state that is shared among instances of several modules. (An interesting, but unexplored, consequence of this is that nonlocal references that are not resolved within the "top-level" environment can be considered to be shared, persistent names.)

In addition, a module can serve as a "factory" that produces initialized instances of nested modules. That is, a method of an outer module can instantiate a (possibly encapsulated) nested module and initialize the instance with (possibly shared) state before returning it to the caller. A generalization of this idea is explored in Section 4.3.3.

4.3.2 Modeling

Some real-world modeling problems can be nicely solved via nested classes. One such problem is known as the *prototype abstraction relation problem* ([55], page 123). The problem is how to model a concept that can be viewed both as an instance of a more general concept (since it has state changing over time) and as a class (since it models a prototype of similar entities). For example, the concept of a flight from point A to point B can have changing state, e.g., its carrier and schedule. Given a particular flight, such as Delta 1415 at 9:30 am, there are instances of that flight each day.

One solution to this problem is to model the notion of a flight from A to B as an outer class which contains a nested class. A particular flight, such as Delta 1415 at 9:30 am, is an instance of this outer class. This instance is a prototype for individual flights of Delta 1415 at 9:30 am, which are themselves modeled as instances of the class nested within it.

Another solution to the above problem is to use meta-classes. That is, the flight from A to B is a meta-class that can be instantiated into a class representing Delta 1415 at 9:30 am, which can itself be instantiated into individual flights.

However, an important point to be made here is that it is not always necessary to pay the price of complexity of meta-classes; block structure can solve the same problems in many cases. This is explored in more detail in the following section.

4.3.3 Manager Modules

Reflection is a means by which programs can access and manipulate themselves. *CMS* supports a form of reflective programming on modules with the introspective primitives given earlier in Section 3.1.6, in conjunction with *manager modules*.

A manager module consists of a nested module along with methods that manipulate some extensible functionality to be supported on the nested module. In a sense, a manager module can be used to simulate a meta-class in more conventional designs — this has already been shown by using block structure in the programming language Beta ([55], page 124). For example, a generic manager module can be specified as follows:

```
(define manager
  (mk-module ()
   ((new  (lambda () (mk-instance (self-ref class))))
    (ref   (lambda (inst attr args)
             (apply attr-ref inst attr args))))))
```

This module specifies a method new that returns an instance of an undefined attribute called class and a method ref that accesses the attribute attr of inst. Basically, the new and ref methods act as surrogates for mk-instance and attr-ref for modules bound to the attribute class. (The new and ref methods could actually be named mk-module and attr-ref.) The idea is that any module can be bound to the attribute class and the new and ref methods can be specialized appropriately for that module via manipulation of the manager module.

For instance, a car-manager module that counts the number of instances of the car-class module can be created by binding the car-class module to class, adding a counter attribute and wrapping the new method with code that increments the counter. Attribute references on the instances of car-class can also be specialized by wrapping the ref method and by using the convention of accessing car-class attributes only via the specialized ref method (although a policy such as this cannot be enforced).

4.3.4 Hierarchy Combination

An inheritance "hierarchy" in OO programming is usually thought of as a graph of inheriting classes. In some languages, hierarchies are indeed represented internally as graphs. In *CMS*, an inheritance hierarchy is represented simply by a collection of module expressions, some of which are mk-module expressions and others which adapt and combine these modules. Such a hierarchy of modules can be nested within another module. That is, the base class of the hierarchy can be a nested module, and other modules that inherit from it can be computed via module expressions within methods of the outer module (since modules are first-class). Examples of hierarchy combination are shown in Figure 4.2 and pictorially in Figure 4.3. A hierarchy veh consisting of a vehicle module and its "subclass" car can be written as shown in Figure 4.2(a), and pictorially in 4.3(a).

Entire hierarchies such as the above can be "combined" with other hierarchies by manipulating the outer modules. Consider a hierarchy cap with a module vehicle with a single attribute capacity as shown in Figures 4.2(b) and 4.3(b). Suppose we wish to extend the hierarchy veh with the hierarchy cap, so that an attribute capacity is added to the vehicle module (i.e., the *superclass*), which will be automatically inherited by car (i.e., the *subclass*). This results in the hierarchy veh-cap shown in Figure 4.3(c) and can be produced by the expression shown in Figure 4.2(c). This expression has considerable similarity with those given in the section on multiple inheritance (Section 3.4).

(a)
```
(define veh (mk-module ()
              ((vehicle  (mk-module ((fuel 0)) ()))
               (car      (lambda () (override (self-ref vehicle)
                                      (mk-module ((color 'white)) ()))))))))
```

(b)
```
(define cap (mk-module ()
               ((vehicle (mk-module () ((capacity 10)))))))
```

(c)
```
(define veh-cap
  (override (override (copy-as veh '(vehicle) '(veh-vehicle))
                      (copy-as cap '(vehicle) '(cap-vehicle)))
            (mk-module ()
               (vehicle (lambda ()
                          (override (self-ref veh-vehicle) (self-ref cap-vehicle)))))))
```

(d)
```
(define disp
  (mk-module ()
   ((vehicle  (mk-module () ((display (lambda () ...)))))
    (car      (mk-module () ((display (lambda () ...))))))))
```

(e)
```
(define veh-disp
  (override (override (copy-as veh '(vehicle car) '(veh-vehicle veh-car))
                      (copy-as disp '(vehicle draw) '(disp-vehicle disp-car)))
            (mk-module ()
              ((vehicle (lambda ()
                          (override (self-ref veh-vehicle) (self-ref disp-vehicle))))
               (car (lambda ()
                      (override (self-ref veh-car) (self-ref disp-car))))))))
```

(f)
```
(define cap-disp
  (merge (merge (rename cap '(vehicle) '(cap-vehicle))
                (rename disp '(vehicle) '(disp-vehicle)))
         (mk-module ()
           ((vehicle (lambda ()
                       (merge (self-ref cap-vehicle)
                              (self-ref disp-vehicle))))))))
```

Figure 4.2. Example of hierarchy combination. (a) The veh hierarchy, (b) the cap extension hierarchy, (c) combining veh and cap, (d) the disp extension hierarchy, (e) combining veh and disp, and (f) combining the parallel extensions cap and disp.

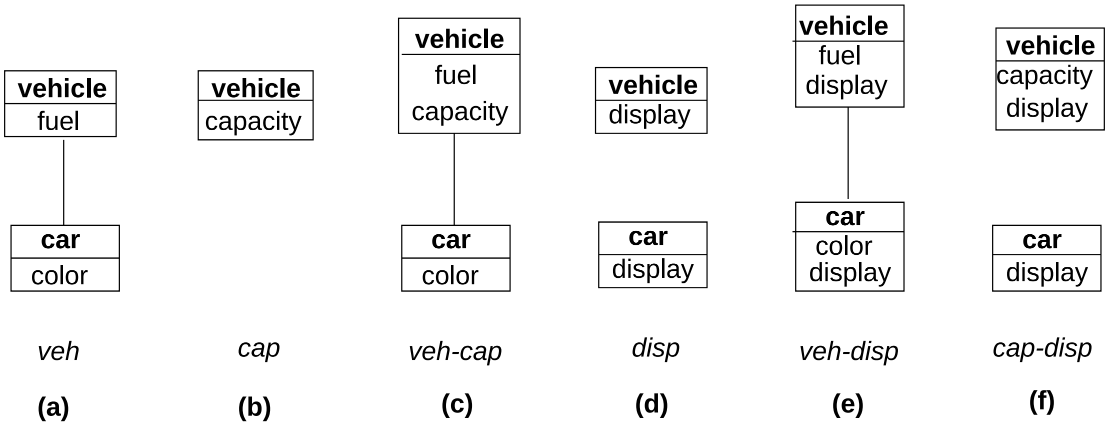


Figure 4.3. Hierarchy combination example shown pictorially.

Similarly, a disp hierarchy can be used to extend the veh hierarchy. In disp, the two modules shown provide their own display methods. The disp hierarchy can be combined with the veh hierarchy to produce the veh-disp hierarchy with the expression shown in Figure 4.2 (d) and (e).

In the above examples, the hierarchies cap and disp can be considered "parallel" extensions of the hierarchy veh. An interesting and often useful operation on parallel extensions is that they can themselves be combined if there are no conflicts between them, as given below and shown in Figure 4.2(f). Note the use of merge in the above expression (as opposed to override) since we want to disallow conflicts.

Effects similar to the above were first given by Ossher and Harrison [65]. Hierarchy combination is a significant but consistent extension to OO programming since it supports the development of modifications to entire hierarchies as well as separated extensions which can themselves be integrated.

4.4 Semantics

In this section, we provide a denotational description of the notion of compositional nesting described earlier in the context of *CMS*. The reader who is not interested in the formal underpinnings of compositional nesting may safely skip this section.

Recall that nested modules in generator semantics, presented in Section 2.2.1, can contain free references which implicitly refer to names in a surrounding lexical scope and are found in the module's *environment*. For an imperative language with stores, this semantics is modeled by passing the environment as an argument to the semantic function that creates constructors. The constructor retains the environment in which it was created. Upon instantiation, i.e., when applied to a store, it extends its retained environment with bindings arising from the attribute definitions in the module.

With respect to nested modules, there are at least two problems with the model of generator semantics. The major problem is that modules can have free references, which is at odds with goal (1) in Section 4.1. Free references compromise an important requirement of modularity: the ability to develop and maintain a module independent of its context. Second, modules in this model must be a priori designed to be nested or not; they cannot be nested after the fact. This is at odds with goal (2) in Section 4.1.

In this section, we attempt to remedy these problems first via a notion of *importation* in Section 4.4.1, and more cleanly and elegantly with an extended form of generators called *closed* generators in Section 4.4.2.

4.4.1 Importation

Let us first attempt to retrofit support for nested modules into the original framework of generators. We do this by disallowing free references and modeling nonlocal references via the already existing mechanism of self-referencing. Thus, a nonlocal reference is modeled as a self-reference to an undefined attribute. With this approach, self-referenced attributes (that really stand for non-local references) can be imported from bindings in an environment by using an import operator:

$$\mathsf{import}\ a = \lambda g.\ \lambda e.\ \mathsf{hide}\ a\ \lambda s.\ \{a = e._r a\}\ \|_r\ \ g(s)$$

This operator takes a module (generator) and an environment and produces a new module with the given attribute bound to its value in the environment. The type rule must make sure that a is not already defined in the incoming module.

The import operation hides the imported attribute in the resultant module. This is because multiple modules that import an attribute with the same name from the same environment should be mergeable without conflicts arising only from importation. To see why, consider sibling nested modules that import the same name from a surrounding environment (generated from a module). If such modules are otherwise mergeable, importation should not result in a merge conflict.

The above semantics satisfies the goal of independent development. However, the goal of composability is only partially met. This semantics presumes module combination at run-time, since environments are run-time entities. This is not always desirable or possible. Furthermore, the goal of local/nonlocal access is not satisfied, since the two name-spaces are undesirably combined together. An additional problem is that of verbosity: one must explicitly import from each instance of a surrounding module as opposed to specifying a permanent nesting relationship between a module and its surrounding module.

4.4.2 Closed Generators

In this section, we describe an alternative formulation of modules that meets all the goals set out in Section 4.1. In this formulation, a module is modeled as a *closed-generator* — a generator that abstracts over its environment. The domain equation for closed-generators, and an example are shown below:

$$\mathit{Closed\text{-}generator} = \mathit{Environment} \rightarrow \mathit{Instance} \rightarrow \mathit{Instance}$$

$$g_c = \lambda e.\ \lambda s.\ \{a_1 = v_1, \ldots, a_n = v_n\}$$

Closed-generators do not have any free references. Self-references are modeled as explicit references to the s parameter (e.g., $s.a_x$), and environment references are modeled as explicit references to the e parameter (e.g., $e.a_y$).

A closed-generator is instantiated by applying it to an environment before taking its fixpoint.

$$\mathsf{instantiate} = \lambda g_c.\ \lambda e.\ Y(g_c(e))$$

Within this framework, a nested module is modeled as a closed-generator that is applied to an environment $(e \leftarrow_r s)$, where s and e are the self and the environment of the outer module. The (nonclosed) generator thus obtained is closed again by abstracting it over a "dummy" environment d. For example, the attribute a_x is bound to a nested module within module g_c below:

$$g_c = \lambda e.\ \lambda s.\ \{\ldots, a_x = \lambda d.\ g_{c_{nest}}(e \leftarrow_r s), \ldots\}$$

The environment $(e \leftarrow_r s)$ of $g_{c_{nest}}$ is bound at the time the outer module g_c is instantiated. Subsequently, the nested module bound to a_x can be selected and passed around. Regardless of the environment it is supplied during its own instantiation, its nonlocal references will always refer to its lexically surrounding scope g_c. This is the desired effect of lexical scoping.

Nested modules, by virtue of being part of the implementation of the outer module, have access to the encapsulated attributes of the outer module. This fact is modeled accurately by the above model, since the self s of the outer module is supplied as the environment to the nested module.

A closed-generator can be nested within another after the fact via the operator nest. The operator creates a new closed-generator by embedding the incoming closed-generator as an attribute named n within the resultant, as shown below:

$$\mathsf{nest}\ n = \lambda g_{c_{in}}.\ \lambda g_{c_{out}}.\ \lambda e.\ \lambda s.\ \{n = \lambda d.\ g_{c_{in}}(e \leftarrow_r s)\}\ \ \|_r\ \ g_{c_{out}}(e)(s)$$

Closed-generator semantics can be easily integrated with ordinary generator semantics. Closed-generator versions (subscripted with g_c) of unary and binary generator operators (subscripted with g) can be specified as given below:

$$\mathcal{U}_{g_c} = \lambda g_c.\ \lambda e.\ \mathcal{U}_g\ g_c(e)$$

$$\mathcal{B}_{g_c} = \lambda g_{c_1}.\ \lambda g_{c_2}.\ \lambda e.\ g_{c_1}(e)\ \mathcal{B}_g\ g_{c_2}(e)$$

4.4.3 Imperative Closed Generators

This section provides a semantics for the mk-module and nest primitives of *CMS*, using closed generators with stores. For this, the notion of closed generators presented in the previous section is augmented with the notion of constructors given in Section 2.2.1. Thus, an imperative closed generator takes an environment and produces an ordinary imperative generator (as given in Section 2.2.1), which when applied to a store returns an instance and an updated store.

$$\textit{Closed-generator} = \textit{Environment} \rightarrow \textit{Constructor} \rightarrow \textit{Constructor}$$

$$\textit{Constructor} = \textit{Store} \rightarrow (\textit{Instance} \times \textit{Store})$$

Given below is the semantic function $\mathcal{M}$, which takes in a syntactic module expression M_{syn} and return an imperative closed generator. A semantic function $\mathcal{B}$ that generates an environment of names bound to values and locations from a syntactic list of attributes, B_{syn}, is assumed.

$\mathcal{B} : B_{syn} \rightarrow \textit{Environment} \rightarrow \textit{Store} \rightarrow (\textit{Environment} \times \textit{Store})$

$\mathcal{M} : M_{syn} \rightarrow \textit{Closed-generator}$

$$\begin{array}{l}
\mathcal{M}[\![\ (\textsf{mk-module}\ \langle \textit{attributes} \rangle)\ ]\!] = \\
\quad \lambda e.\ \lambda c_{self}.\ \lambda s_{create}. \\
\qquad \textit{let}\ (e_{self}, _) = c_{self}\ s_{create}\ \textit{in} \\
\qquad\quad \mathcal{B}[\![\langle \textit{attributes} \rangle]\!]\ (e \leftarrow_r e_{self})\ s_{create}
\end{array}$$

$$\begin{array}{l}
\mathcal{M}[\![\ (\textsf{nest n}\ \langle \textit{nested-module} \rangle\ \langle \textit{outer-module} \rangle)\ ]\!] = \\
\quad \lambda e.\ \lambda c_{self}.\ \lambda s_{create}. \\
\qquad \textit{let}\ (e_{self}, _) = c_{self}\ s_{create}\ \textit{in} \\
\qquad\quad \textit{let}\ g_{in} = \mathcal{M}[\![\langle \textit{nested-module} \rangle]\!]\ (e \leftarrow_r e_{self})\ \textit{in} \\
\qquad\qquad \textit{let}\ (r_{out}, s_1) = \mathcal{M}[\![\langle \textit{outer-module} \rangle]\!]\ e\ c_{self}\ s_{create}\ \textit{in} \\
\qquad\qquad\quad (\{n = \lambda d.\ g_{in}\} \parallel_r r_{out},\ s_1)
\end{array}$$

4.4.4 Static Typing

In order to satisfy the static safety requirement of Section 4.1, we outline static typing for closed generators and nesting in this section. Note that the augmentation of ordinary generators into closed generators requires augmentation of static type rules defined on ordinary generators in ref. [7]; however these augmentations will not be given here.

The first question is whether we can treat imported names simply as declared names, thus not distinguishing between declared self-references and nonlocal references. The problem with such an approach arises when trying to nest one module within another — we cannot determine which declared names of the nested module should be imported from the names of the outer module and which correspond to self-referenced names and thus should not be imported. Thus, modules corresponding to closed generators must be given a type consisting of declared, defined, and imported attributes.

If a module defines unique attributes $a_1 \ldots a_m$ with bindings $v_1 \ldots v_m$ with valid type signatures $\alpha_1 \ldots \alpha_m$, declares unique attributes $d_1 \ldots d_k$ with valid type signatures $\delta_1 \ldots \delta_k$, and imports unique names $e_1 \ldots e_l$ with valid type signatures $\epsilon_1 \ldots \epsilon_l$, then the module has the type:

$$\begin{array}{lll} \{ & \textbf{define} & a_1 : \alpha_1, \ldots, a_m : \alpha_m \\ & \textbf{declare} & d_1 : \delta_1, \ldots, d_k : \delta_k \\ & \textbf{import} & e_1 : \epsilon_1, \ldots, e_l : \epsilon_l \quad \} \end{array}$$

The operator nest imports those attributes that are specified as **import** in an "inner" module from declared, defined, or import attributes in an "outer" module. Those import attributes in the inner module that are not already declared, defined, or import in the outer module become import attributes of the resultant module. Attributes actually imported from an outer module can be subtypes of the types specified for import attributes in the inner module. Furthermore, the nested module in the resultant module does not have any remaining import attributes; all of them have been "pushed up" into the outer module. This corresponds to the notion that the environment of the inner module is fixed after the nest operation. These notions are captured in the type rule given in Figure 4.4 (omitting clauses for uniqueness conditions, etc.).

This concludes our treatment of the semantics of nested modules.

4.5 Discussion and Related Work

In C++ [28], class nesting is merely a name-space structuring mechanism. Nonlocal references within nested classes are disallowed (except to statics, which are globally referable via qualified names) .

A more conventional semantics is to have nonlocal references access bindings of names in a surrounding scope. A local binding essentially *overrides* a binding

$$
\begin{array}{rll}
\Gamma \vdash m_{out} : \{ & \textbf{define} & a_1 : \alpha_{11}, \ldots, a_m : \alpha_{1m}, \\
 & & b_{11} : \beta_{11}, \ldots, b_{1p} : \beta_{1p} \\
 & \textbf{declare} & c_1 : \gamma_{11}, \ldots, c_q : \gamma_{1q}, \\
 & & d_{11} : \delta_{11}, \ldots, d_{1r} : \delta_{1r} \\
 & \textbf{import} & x_1 : \phi_{11}, \ldots, x_s : \phi_{1s}, \\
 & & y_{11} : \psi_{11}, \ldots, y_{1t} : \psi_{1t} \ \ \} \\
\Gamma \vdash m_{in} : \{ & \textbf{define} & b_{21} : \beta_{21}, \ldots, b_{2u} : \beta_{2u} \\
 & \textbf{declare} & d_{21} : \delta_{21}, \ldots, d_{2v} : \delta_{2v} \\
 & \textbf{import} & a_1 : \alpha_{21}, \ldots, a_m : \alpha_{2m}, \\
 & & c_1 : \gamma_{21}, \ldots, c_q : \gamma_{2q}, \\
 & & x_1 : \phi_{21}, \ldots, x_s : \phi_{2s}, \\
 & & y_{21} : \psi_{21}, \ldots, y_{2w} : \psi_{2w} \ \ \} \\
\forall i \in 1 \ldots m & \alpha_{1i} \leq \alpha 2i & \\
\forall j \in 1 \ldots q & \gamma_{1j} \leq \gamma 2j & \\
\forall k \in 1 \ldots s & \phi_{1k} \leq \phi 2k & \\
\hline
\Gamma \vdash \mathsf{nest}\ n\ m_{in}\ \ m_{out} : \{ & \textbf{define} & a_1 : \alpha_{11}, \ldots, a_m : \alpha_{1m}, \\
 & & b_{11} : \beta_{11}, \ldots, b_{1p} : \beta_{1p}, \\
 & & n : \{ \ \textbf{define} \quad b_{21} : \beta_{21}, \ldots, b_{2u} : \beta_{2u} \\
 & & \qquad \textbf{declare} \ \ d_{21} : \delta_{21}, \ldots, d_{2v} : \delta_{2v} \\
 & & \qquad \textbf{import} \quad \} \\
 & \textbf{declare} & c_1 : \gamma_{11}, \ldots, c_q : \gamma_{1q}, \\
 & & d_{11} : \delta_{11}, \ldots, d_{1r} : \delta_{1r} \\
 & \textbf{import} & x_1 : \phi_{11}, \ldots, x_s : \phi_{1s}, \\
 & & y_{11} : \psi_{11}, \ldots, y_{1t} : \psi_{1t}, \\
 & & y_{21} : \psi_{21}, \ldots, y_{2w} : \psi_{2w} \ \ \}
\end{array}
$$

Figure 4.4. Type rule for the nest operator.

to the same name in a surrounding scope. This is equivalent to the conventional semantics of nested functions in most languages. Examples are ML substructures [76], Scheme modules [25], and Beta subpatterns. However, one can only directly lexically nest modules in these systems; there is no notion of compositional nesting.

The most experience with nested modules to date is with subpatterns supported by the Beta language [51, 55]. Nested patterns are idiomatically used in Beta for obtaining several effects, one of which is the mixin style of programming. For example, a mixin such as AMixin

```
AMixin: (# In: virtual class InType;
```

```
        Out: class In (# ... #)
#)
```

can be used by first binding the virtual class In by deriving from AMixin, then extracting the nested class Out (which is "mixed in" with In). In BETA, this programming idiom is useful since it permits one to bind a virtual class such as In with a class that is a *subtype* of the declared type InType. However, such a construct cannot be statically typed. Idioms such as this are obtainable by other means in *CMS*, as shown in the previous chapter.

Finally, it should be pointed out that separate compilation of nested modules can be performed by using conventional link-editing techniques. Environmental references can be stored as relocation information in the compiled object file and patched up with the proper nonlocal access code when combined with an outer module.

4.6 Summary

This chapter showed how to achieve the goal that a language that supports compositional modularity must also support nested modules as independent composable abstractions. We develop a denotational semantic formulation of nested modules as closed-generators that meets the above goal.

An important point of this chapter is that block structure is not orthogonal to modularity. Madsen[55] has argued that block structure is not a mechanism for programming in the large and that a language must contain other facilities for modularizing a program into smaller parts. Here, we show that nesting is itself a form of composition, and nested modules can indeed be used for programming in the large.

We further go on to illustrate several applications of nested modules such as sharing, name-space control, modeling, manager modules, and inheritance hierarchy combination. We show that it is not always necessary to pay the price of complexity of meta-class support; compositional nested modules can solve the same problems in many cases.

CHAPTER 5

THE ETYMA FRAMEWORK

et.y.mon (pl. et.y.ma also etymons) [L, fr. Gk] . . . 2: a word or morpheme from which words are formed by composition or derivation.
— Webster Dictionary

Earlier, it was argued that the model of compositional modularity can be expressed independent of particular realizations. In this chapter, we show that the notions fundamental to compositional modularity can be formulated as a simple software architecture, an OO framework named ETYMA, that can be effectively *reused* to build tools for a wide variety of systems.

The framework was designed to be simple, extensible, and reusable and to support introspective reflection. The abstract and concrete classes that constitute the framework are presented in Section 5.2. These classes cover the domain of values of interest within compositional modularity as well as their types.

Systems based on this framework benefit not only from the power and flexibility that the architecture offers but also from significant design and code reuse. The implementation architecture of an interpreter for the language *CMS* and the extent to which it benefits from reuse are given in Section 5.3.

Finally, reuse characteristics of the framework, such as its design, documentation, and its evolution over reuse iterations, are discussed in Section 5.4.

5.1 OO Frameworks and Design Patterns

The challenge to broadly applying the abstract model of compositional modularity is to realize it in a practical, coherent, and reasonably complete manner. For this, we exploit the idea that generic linguistic notions such as "module," "record," "instance," etc. can be organized as a taxonomy of concepts with relationships such as *is-a, has-a, aggregates,* etc. Furthermore, such a space of concepts can itself be specified using an OO language, thus constituting what is known as an OO *framework.*

In essence, an OO framework is an OO model that captures the essential abstractions in a particular application domain [42]. It expresses the architecture of applications in the domain in terms of objects and interactions between them. Frameworks allow developers to build applications effectively by concretizing abstract classes in the framework via inheritance and by configuring, i.e., connecting instances of, predefined concrete classes in the framework. As a result, a framework can be thought of as being parameterized on a completion that provides *call back* code — a sort of bidirectional function abstraction. Applications are built by *completing* a framework for specific purposes, while preserving the overall architecture of the framework. Frameworks thus promote design and code reuse through OO concepts such as inheritance and polymorphism.

We use the notion of *design patterns* to describe parts of the ETYMA framework. Design patterns are somewhat smaller architectural units than frameworks and are more general. Design patterns systematically name, explain, and evaluate important and recurring solutions to specific problems in OO software design. Thus, a pattern "catalog," such as the one given by Gamma et al. [30], helps an OO designer to identify and apply simple and elegant solutions to commonly occurring problems in OO design to achieve greater reuse and flexibility. Table 5.1 gives brief descriptions of the patterns used in describing ETYMA; the reader is referred to ref. [30] for complete descriptions.

Furthermore, we use OO diagrams to describe the framework. Our diagramming conventions are based on that given in [30], and are given in Figure 5.1, These diagramming conventions have been adapted to describe a framework realized using the C++ language [28] such as ETYMA.

Table 5.1. Design patterns used to describe ETYMA.

Name	Description
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Iterator	Provide a way for accessing the elements of an aggregate object sequentially without exposing its underlying representation.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

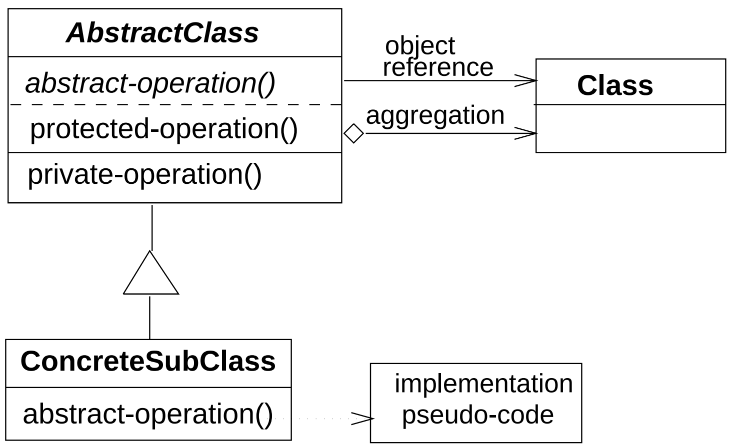


Figure 5.1. OO diagramming conventions. A class' interface is divided into three parts: public, protected (methods below dashed line), and private (methods below solid line). Slanted font indicates abstract classes and methods, and regular font stands for concrete ones. Lines with triangles stand for inheritance *and* subtyping (*is-a*).

5.2 The Design of ETYMA

In this section, we present the design of our OO framework for compositionally modular systems. We first outline some of the requirements and issues that governed the design of ETYMA.

1. *Scope.* The framework must encompass abstractions spanning module systems and their associated type systems. Thus, the focus of ETYMA is to facilitate building the processing engines of tools for module systems, rather than for building the front- and back-ends to such systems.

2. *Simplicity.* The framework must consist of a small number of orthogonal abstractions.

3. *Extensibility.* It must be possible to easily add new abstractions and new features to abstractions within the model without overhauling the entire design for each extension. This property is strengthened with more reuse.

4. *Reusability.* The framework must be designed and documented to facilitate design and code reuse. That is, it must include as much functionality of module systems as possible, without committing to specific base languages. Also, every abstraction in the framework must be reused within at least two completions. Furthermore, design decisions and dependencies must be documented to aid reuse clients.

5. *Reflection.* The model must support introspective reflection. That is, each class must support methods to query (but not modify) the significant aspects of its internal representation.

A model of a model, i.e., a meta-model, of a software system is usually referred to as a *meta-level architecture* (or meta-architecture). For instance, the OO programming model supported by Smalltalk [32] is itself captured as a set of interacting meta-objects, such as Object, Class and CompiledMethod, which constitutes the Smalltalk meta-architecture. Thus, a generic OO realization of the essential concepts of compositional modularity, such as ETYMA, can be termed as an OO meta-architecture for compositional modularity.

After introducing the central notions in ETYMA and their relationships in the following section, we present the abstract classes of ETYMA in Section 5.2.2, and some concrete classes in Section 5.2.3. These sections together describe ETYMA.

5.2.1 Concepts and Their Relationships

Compositional modularity deals with modules, their instances, the attributes they are composed of, and the types of all the above. Thus, the primary concepts that must be captured by a meta-architecture are those of *modules*, *instances*, *methods*, *variables*, and their corresponding types.

However, ETYMA is a linguistic framework, i.e., a framework from which programming languages will be designed. Thus, while modeling the above concepts, we must not inadvertently limit their generality. For example, a *method* is a specialization of the general concept of a *function*. Similarly, the concept of a *record* is closely related to that of a *module* and an *instance*, although the precise relationship between these concepts may not be immediately obvious. The first order of business, then, is to clarify the relationships, specifically *is-a* relationships, between these three concepts.

The concept of records models the classical notion: finite functions from labels to values, with no notion of self-reference. Records support operations such as merge, override, rename, restrict, and copy-as, similar to the ones found in refs. [16, 36]. In addition, the select operation on records models attribute selection.

Modules support all the above operations except select, plus the operations freeze and hide. Thus, the concept of a module is clearly not a subtype (*is-a*) of the concept of a record.

Consider the idea that a record *is-a* module. This seems feasible, since we can regard the operation freeze on records to be the identity function, and hide on records to be the same as restrict. Furthermore, if a record *is-a* module, a record can be viewed as a module. However, if the restrict operation is applied to such a record (viewing it as a module), the given attribute will be entirely removed from the interface of the record. Restrict semantics on modules require that the type of the restrict'ed attribute remain in its interface. Consequently, a record cannot be viewed as a module, and hence a record is not a subtype (*is-a*) of module.

Formally, modules are modeled as functions over records — record generators. Thus it is natural that records and modules are not related to each other via the *is-a* relationship. On the other hand, an instance *is-a* record. This is because the fixpoint of a record generator — an instance — is a record with its self-references bound.

In the following sections, we show how to model the abstractions of compositional modularity in an OO manner.

5.2.2 The Abstract Classes

In this section, we give an overview of the abstract classes in the ETYMA framework, as shown in Figure 5.2. These classes are abstract in the conceptual sense — they represent abstract concepts that must be concretized in the setting of a particular completion. In most cases, they also have at least one abstract method (pure virtual in C++) or one template method (pattern, not the C++ concept, see Figure 5.1) that is specified in terms of one or more abstract methods of the same class or of another class.

All classes in ETYMA are subclasses of the root superclass *Etymon*, which simply consists of support methods for debugging (not shown).

Classes *Type* and *TypedValue* are abstract superclasses that model the linguistic domains of types and values respectively. These two classes form the roots of parallel hierarchies. (The hierarchies are almost parallel, with the exceptions noted below.) These two hierarchies are described in the following two subsections.

5.2.2.1 The Value Classes

Subclasses of class *TypedValue* are called the value classes. ETYMA models strong typing; hence concrete subclasses of *TypedValue* are expected to return their concrete type object (an object of a subclass of class *Type*, see Section 5.2.2.2) when queried via *type-of()*.

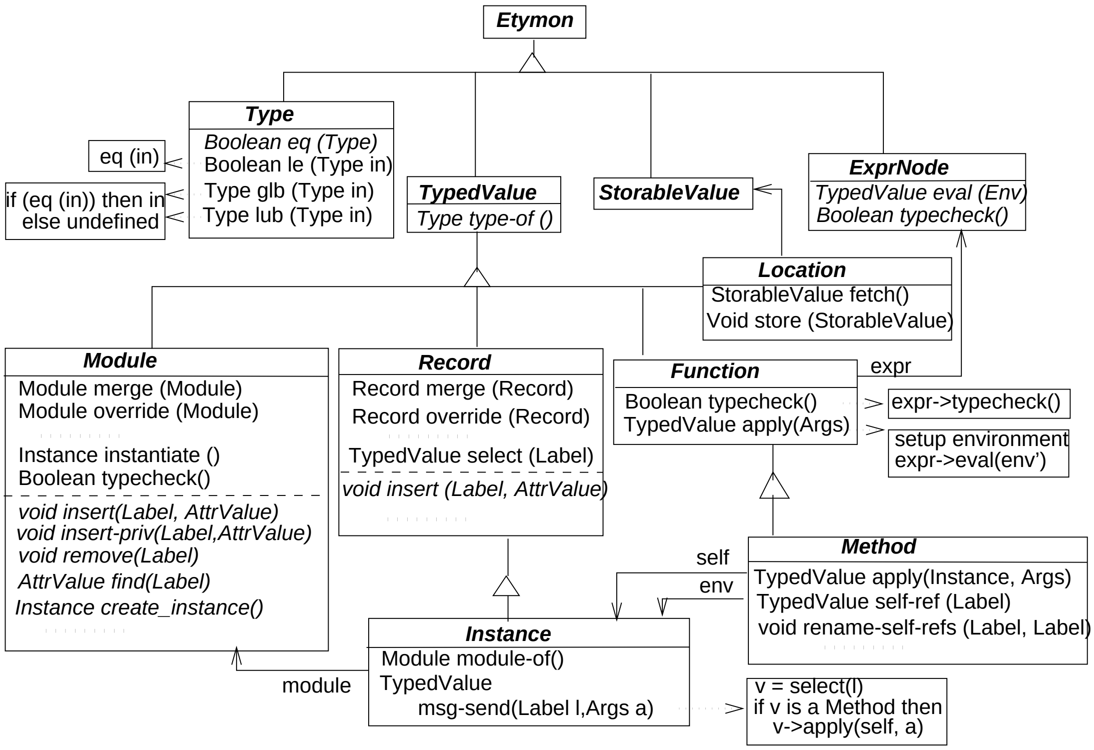


Figure 5.2. Overview of abstract classes.

5.2.2.1.1 Class *Module*. The abstract class *Module*, a *TypedValue*, captures the notion of a compositional module in its broadest conception. The public methods of class *Module* are central to this model; they correspond to the module operators introduced in Chapters 3 and 4: merge, override, hide, copy-as, rename, restrict, freeze, and nest. Additionally there are methods that support instantiation and typechecking.

Within the abstract class *Module*, no concrete representation for module attributes is assumed. Instead, a set of protected abstract methods such as *insert()*, *remove()*, etc. (see Figure 5.2) are defined to manage module attributes. The public module operations are implemented as template method patterns in terms of these abstract methods. Concrete subclasses of *Module* are expected to provide implementations for these abstract protected methods.

To illustrate this design technique, Figure 5.3 shows the pseudo-code for the template method merge in terms of several abstract methods (shown in *slanted* font). The abstract method *create_iter()* is a factory method pattern and is expected

```
Module::merge (Module in)
  if (interfaces of self and in are not mergeable) then
    fail
  out = self.clone()
  i = in.create_iter()
  i.first()
  while ( not i.at_end() ) do
     in_label = i.curr_label()
     in_attr = i.curr_attrval()
     self_attr = self.find(in_label)
     if (self_attr not found) then
        out.insert(in_label.clone(), in_attr.clone())
     else if (self_attr and in_attr are both only declared) then
          glb = greatest common subtype of the types of self_attr and
in_attr
        out.replace(in_label, glb)
     else if (self_attr is declared and in_attr is defined) then
        out.replace(in_label, in_attr.clone())
     i.next()
  merge private attributes
  return out
```

Figure 5.3. Pseudo-code for template method merge. The method is completely specified in terms of several abstract methods, shown in *slanted* font.

to provide an instance of a concrete subclass of an abstract iterator pattern class *AttrIter*. The attribute iterator returned by a call to *create_iter()* is expected to iterate over label-binding pairs, which can be individually extracted via *curr_label()* and *curr_attrval()* respectively.

Module objects have public and private (hidden) attributes. Thus, concrete subclasses of *Module* are expected to maintain two attribute lists corresponding to public and private attributes. Accordingly, there is a set of protected abstract methods in class *Module* to manage private attributes, such as *insert_priv* shown in Figure 5.2.

The abstract method *create_instance()* is a factory method pattern that generates instance objects of module objects. That is, this method is expected to return an object of a concrete subclass of class *Instance*. The method instantiate()

of *Module* then "fills-in" the newly allocated instance with attributes from the module, by sharing nonlocation attributes and allocating new location attributes.

5.2.2.1.2 Classes *Record* and *Instance*. As mentioned earlier, class *Record* supports operations such as merge, restrict, etc. as well as select(Label). All of these methods are implemented as template methods pattern in terms of abstract protected methods, similar to class *Module*.

Class *Instance* is a subclass of *Record*, and it supports operations similar to *Record*. In addition, class *Instance* models the traditional OO notion of sending a message (dynamic dispatch) to an object as *select*'ing a function-valued attribute followed by invoking *apply* on the returned function object. This functionality is encapsulated by the template method pattern msg-send(Label,Args). Furthermore, class *Instance* has access to its generating module with its module member.

5.2.2.1.3 Classes Function, Method, and *ExprNode*. Function-values are modeled via class Function. A function's body is modeled as an object reference to an *ExprNode* object. Class *ExprNode* is a composite pattern, with concrete subclasses representing particular kinds of expressions.

The apply() method of class Function sets up the environment with the current values of the arguments and calls the method *eval* with the environment as the parameter. The appropriate subclass of *ExprNode* can then access the values in the environment as needed.

The concept of a method differs from that of a function in that a method "belongs" to an object; a method has a notion of *self*. Accordingly, class Method, a subclass of Function, requires that the first argument to its apply method be an object of a subclass of class *Instance*, corresponding to its notion of *self*. The apply method of class Method saves the value of its first argument as the self attribute of the Method object, for future access by its self-ref method (explained below).

Self-references within the body of the method can access the attributes (both public and private) of the instance within which the method is executing via the self_ref method of class Method. The self_ref method accesses the previously saved self instance, saved upon entry into the apply method.

Consider what happens if the method body applies the same method object on a different instance of the same module. The apply method is recursively called with the different instance. Upon entry, it saves self — but this overwrites the old value of self ! To solve this, the apply method saves away the old value of self upon entry, overwrites it, and restores the old value before exiting.

Additionally, class Method provides methods for accessing (not shown) and renaming self-references in its body.

5.2.2.1.4 Classes *Location*, *StorableValue* and *PrimValue*. Mutable state (e.g., instance variables) is modeled with class *Location*. Location objects store objects of class *StorableValue*, the exact definition of which depends on a particular completion. For example, storable values typically include at least the primitive values in a language, but often include pointers, which can be modeled as locations containing other locations. Primitive values are modeled as a class *PrimValue* (not shown in Figure 5.2) that multiply inherits from classes *TypedValue* and *StorableValue*.

Some of the other classes referred to, but not described in Figure 5.2, are: class Label which implements the notion of program identifiers (names), class AttrValue which represents label bindings, class Env which represents environments, and class Args which is an aggregation of values. Other support classes for iteration (class Iterator) and lists (class List) are not shown.

The role and use of these abstractions will become clearer with the description of their concrete subclasses in Section 5.2.3 and a framework completion in Section 5.3.

5.2.2.2 The Type Classes

Type classes corresponding to the value classes of the previous section, as well as other commonly found types in modular programming languages, are modeled as subclasses of class *Type* (see Figure 5.4). Class *Type* defines an abstract method *eq* that is expected to check if two types are equal in concrete subclasses. In addition, it defines template method patterns le that checks for subtype (defaults to eq) and glb that computes the greatest lower bound (or greatest common subtype,

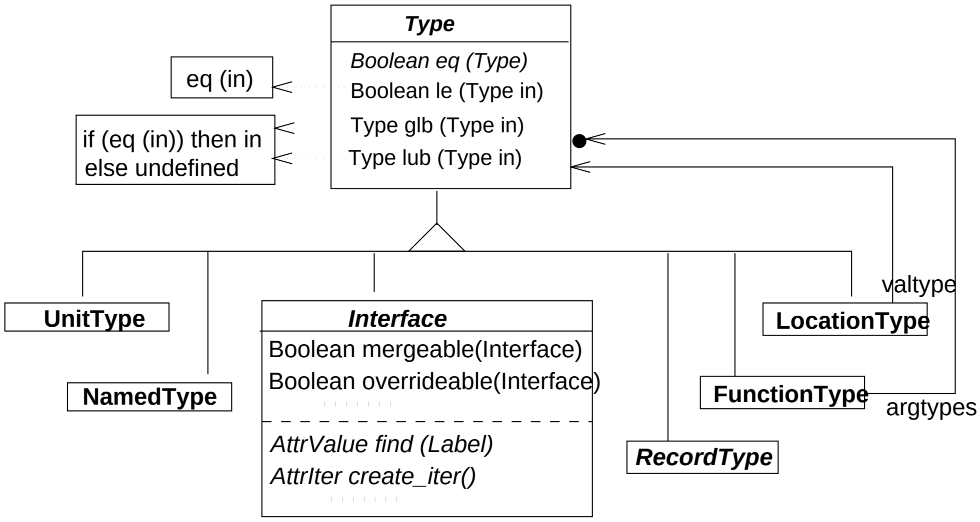


Figure 5.4. Overview of type classes.

required for merge and override semantics). However, in order to compute the greatest common subtype of two function types, it is necessary to compute the lowest common supertype of their input argument types, due to contravariance. As a result, class *Type* also defines a method lub that computes the least upper bound (or least common supertype) of a pair of types. Concrete subclasses of *Type* are expected to appropriately redefine this semantics.

5.2.2.2.1 Classes UnitType and NamedType. Class UnitType represents a type that contains a single member. It is implemented as a singleton pattern.

Class NamedType models types that have identity. For named types, equality is determined by string equality of their names. Subtyping is given by type equality if there is no explicit assertion of subtyping relationships between named types. The class provides a method for explicit assertion of subtyping relationships.

5.2.2.2.2 Classes *Interface* and *RecordType*. Class *Interface*, a subclass of *Type*, models the type of modules. Class *Interface* implements template methods that typecheck individual module operations. Methods of class *Module* call methods of class Interface such as mergeable, overrideable, etc., which implement the type rules for merge, override, etc. These methods are based on the type rules given in

Section 2.2.2 and are defined in terms of abstract protected methods such as *find()*, and the type comparison predicates such as eq and le inherited from class *Type*.

Class *Interface* supports structural typing, in which the names and types of attributes are considered without order significance. For example, its eq method checks, for each attribute, that an attribute with the same name and type exists in the incoming interface. Thus, the eq method relies on the eq method of class Label, which implements identifiers. Subtyping on interfaces is the same as type equality (see Section 2.2.2).

Some applications, e.g., document modules as given in Chapter 8, may require structural typing of interfaces *with* order significance. This can of course be implemented by deriving a concrete subclass of *Interface* that implements its attributes as an ordered collection of label-attrvalue pairs.

Name-based typing of interfaces can be supported by creating a concrete subclass that inherits from classes *Interface* and NamedType and inheriting the eq and le methods from NamedType. Such an interface is referred to as a "branded" interface [59]. Again, the le method may take into account explicit specification of subtyping relationships between branded interfaces.

Class RecordType represents the type of records [36], as well as the type of instances. It supports template methods for typechecking individual record operations, including select. These methods are implemented in a manner similar to those of class *Interface*. This class also implements structural typing, but can be subclassed to implement ordered and branded record types (such as C structs).

5.2.2.2.3 Class FunctionType. Class FunctionType models function types with the standard notions of equality and subtyping, taking into account contravariance. Methods to compare function types are used in the combination of modules that contain function-valued attributes. This should be distinguished from typechecking the *implementation* of a function, which is done by calling the typecheck() method of the function object, which typechecks the expressions comprising the function body.

5.2.2.2.4 Class LocationType. Type equality in class LocationType is given by type equality of its contained type, and subtyping is the same as type equality. Location-bound attributes (variables) can be used as evaluators (i.e., expressions that return values) and as acceptors (i.e., expressions that receive values) in different contexts. Expressions that are evaluators can only be replaced with expressions whose types are subtypes of the original, whereas expressions that are acceptors can only be replaced by expressions whose types are supertypes of the original [10]. As a result, subtyping of variables is always restricted to type equivalence.

5.2.2.2.5 Recursive types. ETYMA supports recursive types (not shown in Figure 5.4). A constituent of a composite type may be a recursive type, represented as an object of class RecType. Class *Type* requires that subclasses provide separate methods for finding equality (eq_rec) and subtyping (le_rec) of recursive types. Subclasses of class *Type* shown in Figure 5.4 directly implement the recursive subtyping algorithm given by Amadio and Cardelli [2], which uses a trail of pairs of known recursive subtypes to avoid diverging on cyclic structures. See Chapter 7 for examples of recursive types.

5.2.3 Concrete Classes

As described thus far, the framework provides a rather generic object model, abstracting over notions such as primitive values and expressions in potential language completions. This basic architecture itself can be used for constructing some kinds of modular systems, such as that explained in Chapter 6.

However, in order to be more directly useful, e.g., for constructing a language interpreter or compiler, concrete subclasses of generic notions must be provided as part of the framework. The idea is that an implementor of a compositionally modular system should be able to find as much reusable design and code in the framework as possible. Consequently, we provide "standard" concrete subclasses of *Module*, *Instance*, *Interface*, and the attribute iterator *AttrIter*, as part of ETYMA, as shown in Figure 5.5. In addition to these, some concrete subclasses of the composite pattern *ExprNode* are provided as well but are not shown in the figure.

Class StdModule is a concrete subclass of *Module* that represents its attributes

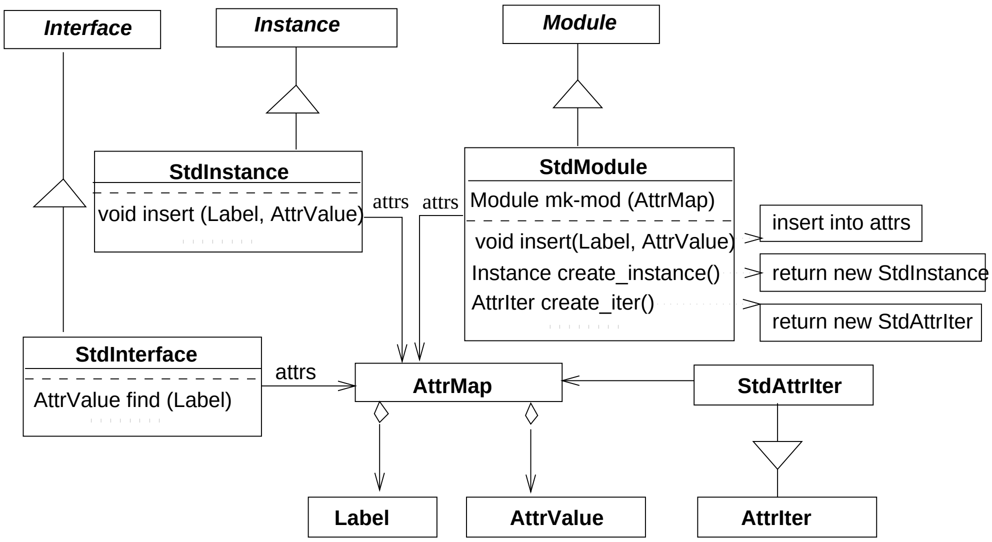


Figure 5.5. Overview of some concrete classes.

as a map (object of class AttrMap). An attribute map is a collection of individual attributes, each of which maps an object of class Label to one of class AttrValue. An AttrValue object encapsulates an object of class *TypedValue*. This design pattern, a variation of the bridge pattern, makes it possible for completions to reuse much of the implementation of class *StdModule* by simply implementing classes corresponding to attribute bindings as subclasses of *TypedValue*.

Each of StdModule's attribute management functions is implemented as the corresponding operation on the map. Its factory method create_iter returns an object of a concrete subclass of class *AttrIter*, class StdAttrIter, which iterates over attribute maps. Similarly, the factory method create_instance returns an object of the concrete subclass of class *Instance*, class StdInstance. Class StdInstance itself is also implemented using attribute maps.

A concrete subclass of class *Interface*, class StdInterface, represents the type of StdModule objects. StdInterface is also implemented as an attribute map. Thus, AttrValue holds an object of a concrete subclass of either *TypedValue* (definitions) or *Type* (declarations).

This concludes the description of the ETYMA framework proper. ETYMA is implemented in the C++ language. It is continually evolving (see Section 5.4.3) but currently consists of about 45 reusable classes and approximately 7,000 lines of C++ code.

A distinguishing feature of ETYMA is that its design has been guided mainly by a formal description (i.e., denotational semantics and type rules) of the corresponding linguistic concepts. The reader might have noted the correspondence between the above framework abstraction design and denotational models of programming languages [33]. Denotational semantics applies functional programming to abstract over language functionality. Here, we apply a denotational description of modularity to abstract over language modularity. Furthermore, the framework approach is intended to provide the language developer a modular means by which to design *and implement* a language's value domain, type system, etc. relatively independently of each other. Once the basic elements of the language are designed, the classes in the framework are directly available for incorporation into the language processor.

5.3 Implementing *CMS* as a Completion

As mentioned earlier, one uses the ETYMA framework by writing concrete subclasses of the framework classes and by instantiating them within a C/C++ program. To give a concrete example of the precise mechanics of completing the framework, this section presents in some detail the design and implementation of an interpreter for the language *CMS* which was presented in Chapters 3 and 4.

The *CMS* interpreter consists of two parts: a basic Scheme interpreter written in the C language, and the module system implemented as a completion of ETYMA. The basic Scheme interpreter itself was derived from a publicly available scriptable windowing toolkit called *STk* [29]. The interpreter implementation exports many of the functions implementing Scheme semantics, thus making it easy to access its internals. Furthermore, the interpreter was originally designed to be extensible, i.e., new Scheme primitives can be implemented in C/C++ and easily incorporated into the interpreter. For example, say we want to implement a new Scheme primitive `new-prim` that takes two input parameters and is implemented by a C function `new_prim_in_C (...)`. This can be done by including the following C function call in the appropriate initialization routine of the interpreter and recompiling it:

```
add_new_primitive ("new-prim", tc_subr_2, new_prim_in_C);
```

Thus, in order to implement *CMS*, Scheme primitives implementing concepts of compositional modularity such as mk-module, mk-instance, self-ref, merge, etc. were implemented in C++ and incorporated into the interpreter. More specifically, once the new *CMS* primitives were identified, subclasses of Etyma classes that support the primitives were designed and implemented. Furthermore, "glue" functions that extend the basic interpreter by calling the appropriate methods of the new subclasses were also implemented.

The overall architecture of the *CMS* interpreter is shown in Figure 5.6. The Scheme library shown on the left includes the macros for OO programming such as define-class and define-prefix presented in Chapter 3, as well as other support functions written in *CMS*. The subclasses of framework classes comprising the completion are shown in Figure 5.7 and described in the following sections.

In order to extend Scheme with a new datatype corresponding to modules that itself can contain other Scheme values, we must first model the Scheme value and type domains in the framework. Scheme consists of a uniform domain of first-class values that includes primitive values and functions. (It is sufficient to consider these Scheme values for the purposes of this discussion.) Scheme variables are identifiers

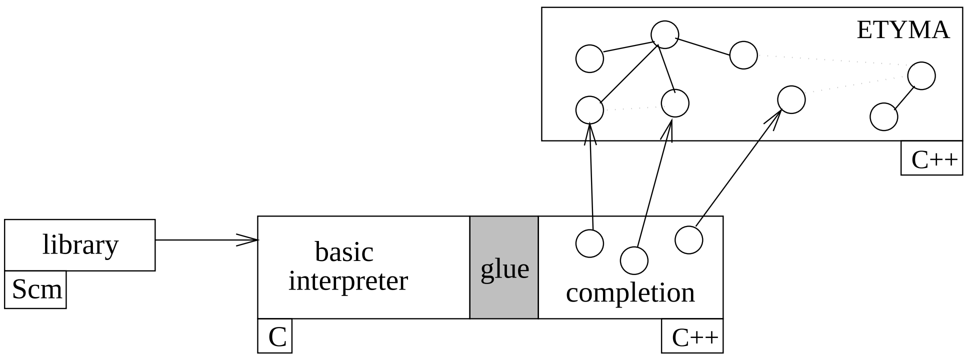


Figure 5.6. Architecture of *CMS* interpreter. The basic Scheme interpreter written in C is extended with C++ code consisting of subclasses of generic classes in the Etyma framework. The Scheme library on the left consists of *CMS* macros and support functions.

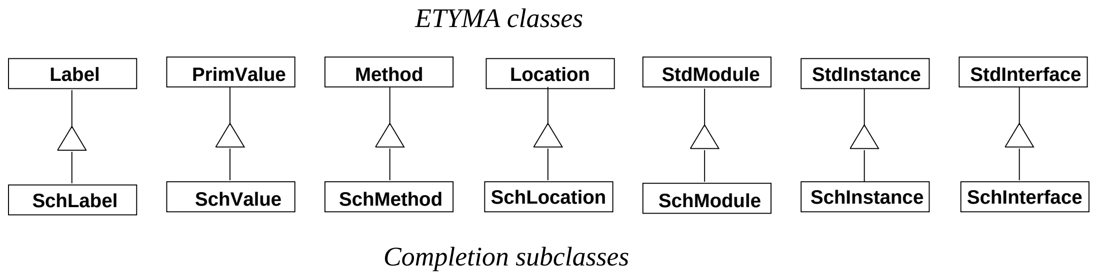


Figure 5.7. Subclasses of framework classes to implement *CMS*.

that are bound to locations, which in turn can contain any Scheme value. Scheme values are typed, but variables are not.

An extension to Scheme must preserve Scheme's essential flavor and pragmatics. Thus, module attributes should be designed to be untyped names. Furthermore, attributes can be bound to values, locations, or methods (functions). Hence, it must be possible to model these notions, i.e., names, values, locations, and functions, using the framework.

The untyped nature of variables can be modeled using the singleton type represented by class UnitType. Scheme identifiers can be modeled with a subclass of the framework class Label, with the notion of equality implemented using the corresponding method for Scheme names exported by the interpreter. Location bindings can be modeled with a subclass of class Location and method bindings with a subclass of class Method. The method subclass need not store the method body as a subclass of *ExprNode*; instead, it can simply store the internal representation of Scheme expression as exported by the interpreter implementation. Self-references and environment references can be managed by searching and modifying the internal representation of the method body, as necessary.

As described in Section 5.2.3, the concrete class StdModule is implemented using class AttrMap which maps Label objects to AttrValue objects. AttrValue objects encapsulate *TypedValue* objects. Thus, the implementation of class StdModule can be almost completely reused for implementing Scheme modules simply by creating the Scheme subclasses mentioned in the previous paragraph. The only modification that needs to be done in the subclass of StdModule is to redefine create_instance to return an object of an appropriate subclass of class StdInstance. This subclass needs to also implement code for *CMS* primitives such as self-ref, self-set!, attr-ref, etc. A subclass of StdInterface also needs to redefine certain typechecking routines to reflect the fact that attributes in *CMS* are not explicity declared, but simply left undefined.

This concludes the description of the *CMS* implementation. The reusability of the framework design, in conjunction with the extensibility of the basic Scheme interpreter, made the degree of reuse so high in this case that most of *CMS* was designed and implemented in a short period of time. Table 5.2 shows several measures of reuse for this completion. The first three rows represent the module system alone implemented as a completion of ETYMA whereas the last row represents the entire system. The percentages for class and method reuse give an indication of *design* reuse, since classes and their methods represent the functional decomposition and interface design of the framework. Similarly, the percentages for lines of code give a measure of *code* reuse. Reuse issues are explored in the following section.

5.4 Reuse Issues

OO frameworks are built to be reused. Although the traditional notion of reuse is that of code reuse, design reuse is generally acknowledged to be equally or more important. According to Peter Deutsch [26], interface design and functional factoring constitutes the key intellectual content of software and is far more difficult to create or recreate than code. In fact, it would not be wrong to say that the primary benefit of OO frameworks is design reuse.

Table 5.2. Reuse of design and code *CMS*.

Reuse parameter		New	Reused	% reuse
ETYMA	Classes	7	25	78
	Methods	67	275	80.4
	Lines of Code	1550	5000	76.3
ETYMA + STk	Lines of Code	1800	20000	91.7

However, it is well known that OO programs are not reusable unless they are specifically designed to be reused. Moreover, once they are designed to be reusable, an appropriate method or "protocol" for effectively reusing the framework must be documented [48]. Some reuse techniques used in ETYMA are given below.

5.4.1 Designing for Reuse

The best known OO reuse mechanism is that of abstract base classes. Abstract classes are partially defined classes that specify the essential characteristics of an abstraction, so that concrete subclasses provide the incomplete parts. For example, consider the abstract class *Module* of ETYMA. One way to design it would be to simply specify all the public methods as abstract methods (pure virtuals in C++). This will certainly enable design reuse; however, there would be no code reuse.

Alternatively, public methods can be designed as template methods to increase code reuse. In class *Module*, the public methods are implemented in terms of very simple protected attribute management methods specified as abstract methods or factory methods. This technique also results in a *layered* design, where a subclass can redefine inherited functionality at a fine level by redefining the protected methods, or at a coarse level by redefining the public methods themselves.

Designers of reusable software must take special care not to preclude efficient implementation. In the case of class *Module*, the public methods implement *applicative* operators; i.e., they return a copy of the module with the requested operation performed on it. Thus, for example, if a concrete subclass desires to compose inherited public methods, there might be undesirable copies made, compromising both time and space efficiency. To avoid this, the public methods of *Module* are implemented in terms of a protected *clone()* method in conjunction with protected, *imperative* versions of the operators. Thus, the concrete subclass can compose the imperative versions of operators without making any more copies than necessary.

5.4.2 Documenting for Reuse

In contrast to a code library, it is not sufficient to merely publish one interface in order for clients to reuse an OO framework. *Two* interfaces must be published:

a *usage* interface for external clients (clients that use the public interface), and an *extension* interface for internal clients (clients that complete the framework via inheritance). Furthermore, in order for the extension interface to be utilized effectively by internal clients, it must be accompanied by information about the structure and dependencies between the abstractions modeled in the framework.

The use of OO diagrams as given in this chapter is very useful. The diagrams greatly enhance a client's understanding of the overall architecture of a framework. Identification of design patterns is also helpful, since these patterns have been studied and described extensively.

While trying to implement methods more efficiently, one can end up introducing dependencies between methods that needs to be explicitly documented. For example, consider the implementation of the project operation, the dual of the restrict operation, in a subclass of class *Module*. Project simply restricts all the attributes of the module that are not given to it; thus it can be implemented in terms of the restrict method. If this is done, however, the typechecking code in the implementation of restrict will be executed several times more than is necessary. To avoid that, the project method can duplicate some of the restrict code. In doing so, we have introduced a dependency between the project and restrict methods: every time a subclass redefines the restrict method, the project method must be appropriately redefined as well. Exposing such dependencies has been dubbed "consistent protocols" in the literature [48].

5.4.3 Framework Evolution

A fundamental phenomenon of OO framework construction is evolution over iterative reuse cycles. It is commonly said that the reusability of a framework increases over reuse iterations. Reuse begets reuse.

The design cycle of frameworks is usually characterized as follows: Start by building one application in the domain of interest and identify those abstractions that can be generalized. For $n > 1$, build the n^{th} application by reusing the generalized abstractions identified thus far, simultaneously trying to identify other generalizable abstractions from all n applications. Although the value of n to produce a truly "reusable" framework varies widely from case to case, it is generally believed that most of the reusable value in a framework can be obtained by $n = 3$.

The C++ realization of the ETYMA framework has undergone several iterations over almost two years. We outline below the major evolutionary stages:

1. The very first version of ETYMA was almost fully concrete and was designed to experiment with a module extension of the C language. It consisted only of the notions of modules, instances, primitive values, and locations, along with a few support classes. C++ programs that instantiate these classes and thus "program with modules" could be written; however, an appropriate front-end and back-end were not constructed. The experiment provided a glimpse of a modular C but was not completed (since it was judged that such a project would demand vastly more effort on front-end and back-end considerations, rather than the framework itself). However, it helped identify some of the basic implementational characteristics of the framework.

2. The next incarnation of ETYMA was used to build a typechecking mechanism for C language object modules, described in Section 6.4. This experiment solidified many of the type classes of ETYMA. At this point, ETYMA was still primarily a set of concrete classes.

3. The third incarnation was used to direct the design of a programmable linker and loader called OMOS, described in Chapter 6. In this iteration, the framework was not directly used in the construction of OMOS, but it evolved in parallel with the actual class hierarchy of OMOS, as described in Section 6.5. Throughout, several revisions were continually made to ETYMA. In this iteration, most of the abstract classes of ETYMA were introduced and much of the template method pattern implementations within abstract classes were first written. Also, the standard concrete classes given in Section 5.2.3 were developed, and this required the introduction of several new support classes (implemented as C++ templates).

4. The fourth iteration over ETYMA was the construction of *CMS*, described in Section 5.3. By the time of this completion, the framework design was essentially in place. The only changes made to the framework were in the solidifying of class `Method`. Code relating to module nesting was added. Nonetheless, the *CMS* interpreter was constructed within a very short period of time and resulted in a high degree of reuse.

5. The next iteration was to design and implement an IDL compiler front-end, described in Chapter 7. Most of the design developed in iteration (2) above was reused here. There were almost no changes to the framework. Code relating to recursive types was added.

6. The most recent iteration over ETYMA has been to build the document composition system described in Chapter 8. There were no changes to the framework.

As can be seen from the above iterations, the first three iterations essentially evolved the framework from a set of concrete classes to a reusable set of abstract and concrete classes. Thus, these iterations crystallized the reusable functionality of the framework. From the fourth iteration onwards, the framework was mostly reused, with the completions themselves hardly changing the structure of the framework.

As the observed reusability of the framework increased, measurements were taken to record the reuse achieved, starting from iteration (4) above. Reuse measurements for iteration (4) were given in Table 5.2; similar measurements are given for iterations (5) and (6) later. For all three iterations, both design and code reuse were found to be significant, between 73.3% and 91.7%.

5.5 Related Work

Several OO frameworks have been developed, initially for user interfaces and subsequently for many other domains as well [26, 78, 79, 13]. ETYMA bears a close relationship to compiler frameworks [21], which comprise classes usually for generating an internal representation of programs. Compiler frameworks fall into two categories: those that represent programs syntactically such as Sage++ [80] and those that represent programs semantically, such as ours. Compiler frameworks are designed with various objectives, such as for representing abstract syntax, constructing tools for programming environments, or for structuring the compiler itself, e.g., with objects representing phases of the compiler [32], or for enabling compile-time reflection via a meta-object protocol [49]. ETYMA, although supporting many of the above, is unique in that it is intended to be a reusable architecture for constructing a variety of modular systems.

As mentioned earlier, ETYMA represents a *meta-architecture* for modular systems. OO meta-architectures have been employed previously to enable reflective, flexible, and extensible language designs. Many of these advantages stem from the fact that reified (i.e., concretely realized) meta-classes are candidates for systematic reflective access. That is, a system that has a well-designed meta-architecture can essentially provide users not only with its standard interface but with an alternative interface — a "side door" to the internal architecture, which is typically a subset of the meta-architecture interface. Information access and refinement via this alternative interface can enable applications to fine-tune a language implementation to suit its particular needs. Meta-classes can be specialized to suit specific tasks using standard OO techniques such as inheritance. In a compilation setting, meta-classes can even be specialized to statically optimize run-time data layout or generate optimized code for particular special cases.

It is important to clarify the relationship between the concepts of *meta-architecture, reflection*, and *metaobject protocol* (MOP). A meta-architecture models, systematically implements, and documents the fundamental concepts of a system. A meta-architecture is OO if the concepts are modeled as collaborating classes. A system is reflective if its users have introspective (i.e., read) and/or intercessory (i.e., modification) access to the reified meta-architecture of the system. Finally, a MOP documents and illustrates a disciplined method of reflective access to a carefully chosen subset of a system's OO meta-architecture.

ETYMA is primarily an application framework for modular systems. An important point of this dissertation is that the design discipline encouraged by object-oriented methods can be fruitfully applied to the design of programming languages themselves. However, being a meta-architecture, ETYMA enables the construction of reflective systems, and the design of suitable MOPs, and thus could potentially bring the advantages mentioned above.

As points of comparison, we now briefly present two widely known language meta-architectures: Smalltalk-80 [32] and the Common Lisp Object System (CLOS) [47].

5.5.1 Smalltalk

Smalltalk is based on a uniform model of communicating objects. It has a small number of concepts — object, class, instance, message, and method. Every concept in the system is modeled as an object, either instantiable (class object) or not (instance object). The most primitive low-level operations in the system are delegated to a virtual machine. Objects communicate via messages; the semantics of messages are implemented by receivers as methods.

Smalltalk's notion of objects is captured by class Object which provides the basic semantics, including message handling, of all objects in the system. The semantics of classes is captured by class Class along with its superclass Behavior which defines the state required by classes, such as for instance variables and a method dictionary. Further, the class CompiledMethod embodies the notion of a class' method; this class defines a method valueWithReceiver: to evaluate itself.

Smalltalk is a "dual hierarchy" language, as are most object-oriented languages. That is, it has a cleanly articulated class-subclass hierarchy as well as a class-instance hierarchy. In most languages, however, the class-instance hierarchy is not interesting since it comprises only two levels — that of all classes and all instances. In Smalltalk, this hierarchy is deeper and is recursive, as described below.

Every object in Smalltalk is an instance of some class. Since classes themselves are objects, each class object is an instance of yet another class, usually referred to as a *metaclass* object. For example, a class Foo is an instance of its metaclass, given by the expression Foo class. Such metaclass objects are themselves instances of an ordinary class called Metaclass. The metaclass of class Metaclass itself is given by Metaclass class, which is also an instance of class Metaclass, just as Foo class is. The above recursion puts an end to the infinite regression of metaclasses.

Consider the class-subclass hierarchy of metaclasses. Every class in Smalltalk inherits from class Object; hence the subclass hierarchy is a singly rooted tree. The class Metaclass mentioned above is also a subclass of Object. The instances of class Metaclass, such as Foo class, Metaclass class, and even Object class, are all (meta)classes. These metaclasses are subclasses of class Class, which is a subclass of class Object. (The actual subclass hierarchy of Smalltalk is slightly more involved than what is described here, due to the desirability of symmetric class and metaclass hierarchies, but the given description will suffice for this discussion.)

5.5.2 CLOS

The CLOS object system supports the standard concept of *classes*, which can be instantiated into *instances* [45]. Class attributes are called *slots*. A distinguishing feature of the CLOS model is the notion of *generic functions* which are defined independent of any class and can be specialized into *methods* that are applicable to specific classes. Generic functions can be dispatched based on *multiple* arguments (multimethods).

The CLOS meta-architecture specifies the following basic meta-object classes corresponding to the basic concepts of the language: class, slot-definition, generic-function, and method. All user-defined metaobjects must be designed to be subclasses of one of the above meta-object classes. The specified default semantics of the CLOS language are embodied by specializations of the above classes, with names beginning with standard-.., e.g., standard-class, and standard-method.

The class-subclass hierarchy of the CLOS meta-architecture is as follows. At the root is class t which has one subclass standard-object capturing the semantics of all objects in the system. Every class created in the system must have standard-object as its superclass. One subclass of standard-object is the class metaobject, of which the basic meta-object classes mentioned above are subclasses.

The class-instance hierarchy of CLOS essentially has four levels. Individual

CLOS classes are instances of class `class` or one of its subclasses. Class `class` is an instance of (its own subclass) `standard-class`, as are most other meta-object classes.

5.6 Summary

We argued that the simplicity of compositional modularity enables it to be applied to a wide variety of systems such as modular and nonmodular programming languages, linkers, interface definition processors, and document manipulation systems. Thus, it is desirable to build a generic, reusable realization of the model — a meta-architecture — that abstracts over the particular characteristics of the system it is embedded within. We defined the scope of such a meta-architecture and required that it be simple, general, extensible, and reusable, and that it support reflection.

The model of compositional modularity can be expressed independent of base languages. It can also be expressed in an OO manner, i.e., as a set of interacting partially specified (abstract) classes — also known as an OO framework. In this chapter, we presented an OO framework for compositional modularity known as Etyma.

Etyma can be reused to build a variety of compositionally modular systems. The implementation of one such system, an interpreter for the language *CMS*, is described in this chapter. Other systems are described in the following chapters.

Etyma consists of a set of highly intertwined abstract classes that model the value and type domains of modular systems, as well as a set of generally useful concrete subclasses. The interfaces of and relationships between Etyma's abstract value and type classes as well as its concrete classes are explained.

A framework becomes reusable only if it is designed and documented with reusability in mind. Furthermore, a framework becomes more reusable with more reuse. Etyma's design considerations and its repeated reuse have helped it evolve into a reusable framework.

> *As the painter needs his framework of parchment, the improvising musical group needs its framework in time. Miles Davis presents here frameworks which are exquisite in their simplicity and yet contain all*

that is necessary to stimulate performance with a sure reference to the primary conception.

— Bill Evans (liner notes for Miles Davis' "Kind of Blue")

CHAPTER 6

COMPOSITION OF OBJECT MODULES

This chapter develops a second application of compositional modularity: composition of object modules, i.e., programmed linkage of separately compiled files.

An object module is an abstracted namespace. That is, it has a set of file-level symbols (represented as a symbol table) and a set of symbol self-references (represented as relocation information), which can be modeled as an abstracted namespace. Manipulation of abstracted namespaces is precisely the province of compositional modularity; hence it can be used to adapt and combine (link) object files.

The central idea of this chapter is that the *physical* modularity of application components, i.e., separately compiled files, can in essence be viewed as *logical* modularity, i.e., first-class compositional modules, which in turn can support effective application development. A software architecture which can take advantage of this idea is presented in Section 6.2.

Section 6.3 proceeds to describe the mechanics of module management within this architecture. It also shows how this viewpoint solves some longstanding problems with the management and binding of components in present-day application development environments.

It is possible to perform type-safe composition of object modules in the above architecture. The details of such a facility are given in Section 6.4.

6.1 Motivation

6.1.1 Application Composition via Linkage

In a traditional application development environment such as UNIX, application components ultimately take the form of *files* of various kinds — source, object, executable, and library files. Entire applications are typically built by putting together these components using inflexible, and sometimes ad-hoc, techniques such as preprocessor directives and external linkage, all often managed via makefile directives.

It is also natural for developers to generate components corresponding to incremental changes to already existing application components, especially if they subscribe to the software engineering principle known as "extension by addition." This principle holds that it is better to extend software not by direct modification but by controlled addition of incremental units of software. Advantages of "extension by addition" include better tracking of changes and more reliable semantic conformance by software increments. Most importantly, the increments themselves have the potential to be reused in other similar settings.

The model of compositional modularity supports the effective management of all the above kinds of application components. It supports encapsulation which helps to enhance abstraction. It supports several forms of inheritance, which is a mechanism for managing incremental changes to software units. It supports various mechanisms for adapting components to enable their reuse in a broad range of situations. Hence there is much to gain from supporting compositional modularity within the infrastructure of an application development environment, beyond whatever support is provided by the languages in which application components are written.

In this chapter, an architecture for applying compositional modularity to OO application development is presented. This architecture demonstrates a principled, yet flexible, way in which to construct applications from components. This facility is orthogonal to makefiles and does not impose new techniques for building individual application components. Instead, it relies on the idea that the *physical* modularity of traditional application components (i.e., files) can be endowed the power and flexibility of *logical* modularity. Such logical modules can then be manipulated using the concepts of compositional modularity, where first-class modules are viewed as building blocks that can be transformed and composed in various ways to construct entire application programs. Individual modules, or entire applications, can then be instantiated into the address spaces of particular client processes.

This approach has other advantages besides making application construction more principled and flexible. First, it enables a form of OO programming with components written in non-OO languages such as C and Fortran. Second, it enables *adaptive* composition, where the system that manages the logical layer can perform various composition-time, "exec"-time, and possibly run-time optimizing transformations to components. For example, system services (such as libraries) can be abstracted over their actual implementations, adding a level of indirection between a service and its actual implementation. This permits optimizations of the service implementation based on clients' disclosed behavioral characteristics. Such system-level support is explored in several studies [61, 64, 62] and is summarized in Section 6.2.3. However, for the most part, this chapter focuses on application level support.

Indeed, much of this chapter is based on work performed by Douglas Orr and others on OMOS, the Object Meta-Object Server [63]. OMOS was primarily conceived as an "object server" — a server process that generates implementations of programs based on user and system requirements. Although OMOS was not originally constructed for the specific purpose of illustrating the concepts of compositional modularity, it provides an infrastructure that is perfectly suitable for doing so.

It is important to make it clear that compositional modularity supported by a logical layer is not in conflict with object-orientation supported by component-level languages. For example, C++ programmers deal with two distinct notions of modularity: classes, fundamental to logical modularity, and source files, which deal with physical modularity. These two modularity dimensions share many characteristics but have very different senses of composability, i.e., inheritance for classes, and linkage for files. Indeed, they are rather orthogonal in the minds of C++ programmers, because class definitions and source files do not always bear 1-1 relationships, and linkage is performed in a "class-less" universal namespace flattened by name mangling. In essence, they manage programs at two levels: classes with their semantic relationships and files with their linkage relationships. The approach presented here supports a similar degree of manageability for physical artifacts (i.e., files) as for logical artifacts (i.e., classes).

In the following section, we present an application scenario that motivates the architecture presented in this chapter.

6.1.2 A Scenario

Consider a scenario in which a team of developers is building an image processing application using a vendor supplied (shrink-wrapped) library. Say the team completes building an initial version of the application (which is large-scale, say, greater than 100K lines of code) and is now ready for system testing. We can imagine common problems deriving from this scenario:

(i) Call wrapping. Suppose that the team finds that the application malfunctions because it calls a library function edge_detect() on an image data structure, consistently with an incorrect storage format, say with pixels represented as type FLOAT when BYTE was expected. Using traditional tools, this problem is rectified by inserting another library function call to the routine floattobyte() before each site in the application where edge_detect() was being called. This approach not only requires extensive modification of the application source code but also expensive recompilation. Moreover, if two separate shrink-wrapped libraries are to be put together in this manner, sources might not even be available. Instead, it is more desirable to "wrap," at binding time, calls to edge_detect() with an adaptor that calls floattobyte(), all without recompiling the large application. However, such a facility is not usually supported in conventional OS environments.

(ii) Library extension management. Suppose further that the team decides that the application could work much better with an image format slightly different from the format expected by the library, but one which is easy to convert to and from the old format. If the new format is to be supported for future projects, it is best to change all library functions to accept the new format. However, sources for the library are not available; hence it cannot be directly modified. Thus, this would

require developing and integrating a separate extension to the library. Furthermore, there could be several other independent extensions to the library that need to be integrated and supported for future applications. Developing such incremental extensions is much like subclassing in OO programming, but there is usually no support for effectively managing such incremental software units.

(iii) Static constructors and destructors. Imagine that the team wants to make sure that all statically defined images are properly allocated and initialized from disk before the program starts and flushed back to disk before the program terminates. Currently available techniques for doing this are difficult and cumbersome.

(iv) Flat namespace. Say the IP library uses the Motif library, which is in turn implemented in terms of the lower-level X library. Thus, in the traditional scenario, all the symbols imported from the Motif and X libraries become part of the interface exported by the IP library. There is no way to prevent clients of the IP library from obtaining access to the lower level library interface or possibly suffer name collisions with that interface.

The system architecture we present in the following sections offers an effective solution to the above problems. Specific solutions to these problems are given in Section 6.3.3.

6.2 Architecture

6.2.1 Conceptual Layering

The first step in presenting an architecture for managing object modules is to clarify the conceptual layering of application components.

Conceptually, artifacts of physical modularity, i.e., files of various kinds, form a *physical* layer. These modules may be written as components in conventional languages that have no notion of objects. For example, in the case of C, there is no support for manipulating physical modules, much less for generating and accessing instances of them at run-time — files are simply a design-time structuring mechanism.

The physical layer is managed with the help of traditional programming language environments. For example, the C language preprocessor, compiler, the make utility, the debugger, and library construction utilities help the programmer to develop application components of various kinds.

In the architecture presented here, each physical module can be manipulated as a first-class compositional module in what we shall conceptualize as the *logical* layer. In this layer, construction of entire applications is directed by scripts written by application programmers describing the composition of logical modules. Scripts are written in a module manipulation language that supports not only a simple merge of modules in the manner of conventional linking but also many others including attribute encapsulation, overriding, and renaming. Most importantly, since modules are first-class entities in this language, individual operations can be composed in an expression-oriented fashion to produce composite effects such as inheritance in OO programming.

The logical layer is managed by a special tool, in the design of which the following requirements were laid out. First, the tool must provide a language processing system for the module manipulation language. Second, it must perform essential operating system services: that of linking modules and loading them into client address spaces. Third, since these services are in the critical path of all applications, it must be able to perform optimizations such as caching. Finally, it must be continually available. For these reasons, the logical module layer in the prototype described here is managed by a server process — a second generation implementation of a server named OMOS [63].

The module manipulation language supported by OMOS is derived from the programming language Scheme[22] and is similar in flavor to *CMS*. The most important distinction, of course, is that its notion of modules is that of object files. Modules are created by reading in physical object files. Conventional linking is accomplished using the merge operator of the module language. Given the other module manipulation operators, the language enables powerful and flexible application composition indeed — this is the subject of Section 6.3.

6.2.2 Application Construction

In this section, we describe the steps in constructing an application, based on the architecture shown graphically in Figure 6.1.

The first step is to build individual application components (physical modules) using a conventional programming language. (In this discussion, we shall consider only C language components, although the same ideas can be applied to another language such as Fortran.) Individual components, such as c1.c, c2.c, and c3.c in Figure 6.1 can be designed as traditional program files with no knowledge of the logical layer. Alternatively a component can be designed to be reused via suitable programming in the logical layer, such as a "wrapper" module described in Section 6.3.

Application components may be owned and managed by the user or by OMOS.

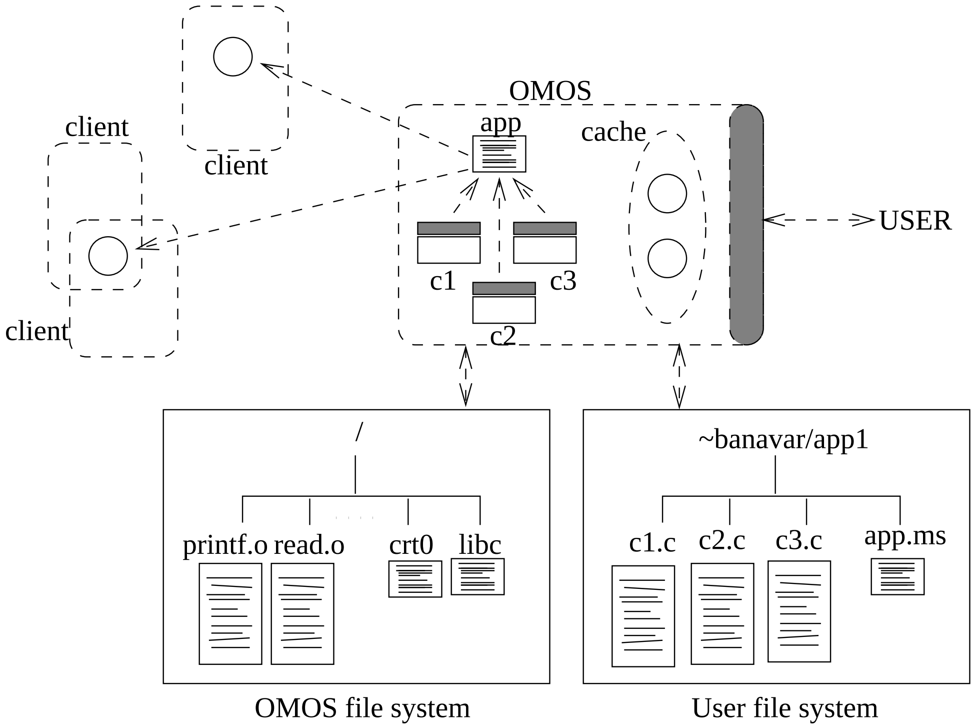


Figure 6.1. Overall architecture of object file composition. c1.c, c2.c, etc. are user application components to be composed as described in the user module spec app.ms. Printf.o, etc., are system components to be composed as given in module specs crt0, etc. These components are composed by OMOS, possibly cached, and instantiated into client address spaces. The user can directly interact with OMOS via a command line interface to effect module composition and instantiation.

In Figure 6.1, c1.c, c2.c, and c3.c are user provided application components. System provided components, such as libraries, are owned and managed by the OMOS server and accessed via service requests to OMOS.

The second step is to create a *module spec*, a file that describes the creation and composition of logical modules from application components. This is written in a *module language* described in Section 6.3. In Figure 6.1, app.ms is a user module spec that describes how to put the components of the application together. Module specs can themselves be modular; they can refer to other module specs. For example, app.ms may refer to libc, a system provided module spec that describes how to put together the components of a standard system library with a client module.

The final step is to request the module server to execute the module spec and instantiate (i.e., load) the result into a client address space. Module specs may be executed by calling a stand-alone version of OMOS from within a makefile and the loading step performed interactively.

6.2.3 The OMOS Server

The most important component of the architecture given above is briefly described in this section. This is the system that manages the logical layer, OMOS [63]. The term "meta-object" was originally intended to be indicative of the intensional nature of module specs and the fact that module specs are really programs that generate other programs. However this name should be considered historical and will not be further justified here.

As mentioned earlier, OMOS is a continuously running process that is designed to provide a linking and loading facility for client programs via the use of module combination and instantiation. OMOS supports three main functions: execution of module specs to compose applications, caching of intermediate results, and program loading.

Clients request OMOS to construct and map a program by providing a module spec that describes how to construct the program. Module specs can refer to object files and other module specs from either the user file system area or the system area.

System provided components, such as the standard C library libc shown in Figure 6.1, are themselves module specs. Application programs that need to use libc load in the module spec and combine the library with other application components in any desired manner.

In fact, system provided libraries are actually represented as function abstractions that receive an application module as a parameter and produce a composed module. Thus, for example, the function body of libc can examine its incoming module parameter using a set of meta-level primitives and custom generate an implementation of libc that is best-suited to the particular application at hand. This mechanism, in effect, provides a level of indirection between a system service and its actual implementation, thus permitting optimizations of the service implementation based on clients' disclosed behavioral characteristics. The details of this mechanism can be found in ref. [62].

Since OMOS loads programs into client address spaces, it can be used as the basis for system program execution and shared libraries [61], as well as dynamic loading of modules. OMOS module specs have also been used to implement program monitoring and reordering [64].

Since OMOS is an active entity (a server), it is capable of performing sophisticated module manipulations on each instantiation of a module. Evaluation of a module expression will often produce the same results each time. As a result, OMOS caches module results in order to avoid re-doing unnecessary work. Combining a caching linker with the system object loader gives OMOS the flexibility to change implementations as it deems necessary, e.g., to reflect an updated implementation of a shared module across all its clients [61].

Backward compatibility is a crucial issue to be addressed in the context of the architecture presented here. Two issues seem to be important: (i) support for old-style linking specs (e.g., ld), which should be automatically translatable to the module language, and (ii) support for old style system services such as libraries (as opposed to module specs). (These issues are currently being worked out as part of ongoing related projects, and will not be further dealt with here.)

There are several other interesting system issues related to OMOS. However, the purpose here is not to explore OMOS but to explore the application of compositional modularity to flexible application development using the infrastructure supported by OMOS. Thus, we conclude the general description of the architecture and proceed to describe the functionality exported by the module language in the following section.

6.3 Object Module Management

As argued in Sections 6.1, an infrastructure that aims to support effective application development must support the flexible management of application components. It was further argued that the management of components, their extensions, and their bindings is essentially similar to the management of classes and subclasses via inheritance in OO programming. This argument behooves us to demonstrate that the architecture presented above does indeed support the essential concepts of OO programming, viz. classes and inheritance, which is shown below in Sections 6.3.1 and 6.3.2 respectively.

Given the facilities described in this section, it is in fact possible to consider doing OO programming with a non-OO language (such as C). However, it is not possible to do full-fledged OO programming in such a manner, because of the reasons given in Section 6.3.1.3. Neither is it desirable, since OO language support (such as C++) might be directly available. Thus, the facility described here is intended mainly for enhancing application component management rather than for actual application programming.

6.3.1 Classes and Instances

6.3.1.1 Modules

An object (".o", or dot-o) file, generated by compiling a C source file, corresponds directly to a compositional module. A dot-o consists of a set of attributes with no order significance. An attribute is either a file-level definition (a name with a data, storage or function binding), or a file-level declaration (a name with an associated type, e.g., `extern int i;`). (Type definitions, e.g., `struct` definitions and typedef's in C, are not considered attributes.) Such a file can be treated just like a class if we consider its file-level functions as the methods of the class, its file-level data and storage definitions as member data of the class, its declarations as undefined (abstract) attributes, and its static (file internal linkage) data and functions as encapsulated attributes. Furthermore, a dot-o typically contains unresolved self-references to attributes, represented in the form of relocation entries.

Symbols, both defined and merely declared, of physical modules make up the interface of logical modules. (For simplicity of presentation, we consider interfaces to comprise only the symbol names, without their programming language types; see Section 6.4 for a study of typed interfaces.) Compiled code and data in the actual object file represent the module implementation.

A physical dot-o is brought into the purview of the logical layer by using the module language primitive called open-module, the syntax of which is given in Figure 6.2. Once it is thus read in as a logical module, it can be manipulated using other primitives of compositional modularity.

6.3.1.2 Encapsulation

Module attributes can be encapsulated using the operator hide (see Figure 6.2). However, in the case of C language components, encapsulation partly comes for free, since C supports the internal linkage directive, static. However, attributes can be hidden after the fact, i.e., nonstatic C attributes can be made static retroactively, with hide. This is a very useful operation as demonstrated in Section 6.3.3.

Many OO systems support the notion of a *class* consisting of public and private

(open-module ⟨*path-string-expr*⟩)
(fix ⟨*section-locn-list*⟩ ⟨*module-expr*⟩)
(hide ⟨*module-expr*⟩ ⟨*sym-name-list-expr*⟩)
(merge ⟨*module-expr1*⟩ ⟨*module-expr2*⟩ ...)
(override ⟨*module-expr1*⟩ ⟨*module-expr2*⟩ ...)
(copy-as ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩ ⟨*to-name-list-expr*⟩)
(rename ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩ ⟨*to-name-list-expr*⟩)

Figure 6.2. Syntax of module primitives.

(encapsulated) attributes. In our system, a similar concept of classes is supported by a Scheme macro define-class, with the following syntax:

(define-class ⟨*name*⟩
  ⟨*dot-o-file*⟩ ⟨*superclass-exprs*⟩ ⟨*encap-attrs*⟩)

For example, given a dot-o vehicle.o that contains, among other attributes, a global integer named fuel and a global method display that displays the value of the fuel attribute, one can write the following expression (within a module spec) to create a class named vehicle by encapsulating the attribute named fuel:

```
(define-class vehicle "vehicle.o" () ("fuel"))
```

This macro expands into the following simple module expression:

```
(define vehicle
   (hide (open-module "vehicle.o") '("fuel")))
```

6.3.1.3 Instances

Instantiating a module amounts to fixing self-references within the module and allocating storage for variables. In the case of instantiation of dot-o modules, fixing self-references involves fixing relocations in the dot-o, and storage allocation amounts to binding addresses. These two steps are usually performed simultaneously. Thus, an object file can be instantiated into an executable that is bound ("fixed") to particular addresses and is ready to be mapped into the address space of a process. Dot-o's can be instantiated multiple times, bound to different addresses. Hence, fixed executables are modeled as instances of dot-o's. A module is instantiated using the primitive fix (see Figure 6.2). The argument ⟨*section-locn-list*⟩ specifies constraints for fixing the module to desired sections of the client address space.

A concept closely associated with first-class objects in conventional OO languages is *message sending*. However, as mentioned earlier, there is no notion of first-class objects at the physical layer, which is where physical modules are implemented using component-level languages. Thus, message sending is not directly supportable in our framework. However, an approach to supporting a form of message sending via inter-process communication can be envisioned, as described in Section 9.2.4.

6.3.2 Inheritance

In this section, we introduce the inheritance related primitives supported by the module language and describe the manner in which they can be composed. We start by introducing the following four primitives whose syntax is given in Figure 6.2. These primitives have essentially the same semantics as for *CMS*, except here, the computational sublanguage is quite different.

The primitive merge combines modules which do not have conflicting defined attributes, i.e., attributes with the same name. This semantics is analogous to traditional linking of object files. However, the idea here is to go beyond traditional linking and support other operations basic to inheritance in OO programming, such as the following.

The primitive override produces a new module by combining its arguments. If there are conflicting attributes, it chooses ⟨*module-expr2*⟩'s binding over that of ⟨*module-expr1*⟩ in the resulting module.

The primitive copy-as copies ⟨*from-name-list-expr*⟩ attributes to attributes with corresponding names in ⟨*to-name-list-expr*⟩. The *from* argument attributes must be defined.

The primitive rename changes the names of the definitions of, *and* self-references to, attributes in its argument ⟨*from-name-list-expr*⟩ to the corresponding ones in ⟨*to-name-list-expr*⟩.

To illustrate the use of the above primitives, the following section describes how to achieve several variations of a facility generally referred to as "wrapping."

6.3.2.1 Wrapping

Figure 6.3 shows a service providing module LIB with a function f() and its client module CLIENT that calls f(). Three varieties of wrapping can be illustrated with the modules shown in the figure.

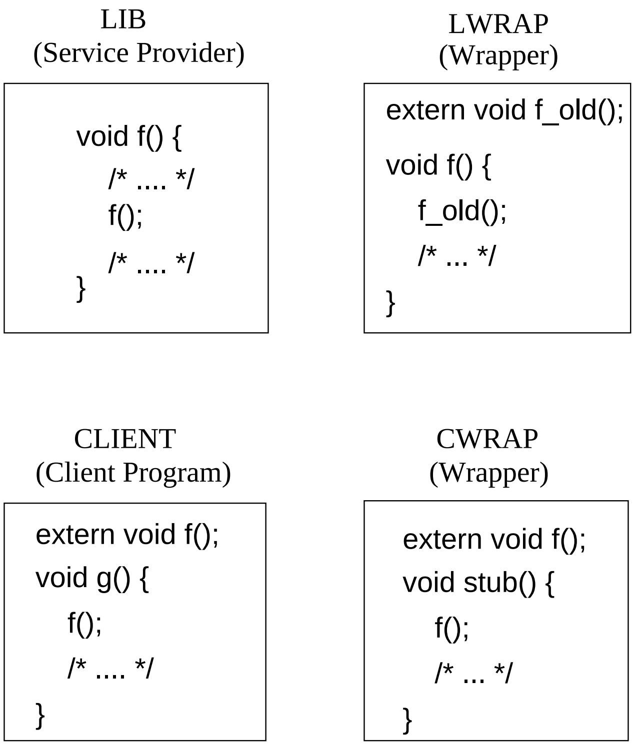


Figure 6.3. Examples of wrapping in OMOS.

(1) A version of LIB that is wrapped with the module LWRAP so that all accesses to f() are indirected through LWRAP's f() can be produced with the expression:

(hide (override (copy-as LIB f f_old) LWRAP) f_old)

By using copy-as instead of rename, this expression ensures that self-references to f() within LIB continue to refer to (the overridden) f() in the resultant and are not renamed to f_old.

(2) Alternatively, a wrapped version of LIB in which the definition of and self-references to f() are renamed can be produced using the expression:

(hide (merge (rename LIB f f_old) LWRAP) f_old)

This might be useful, for example, if we want to wrap LIB with a wrapper which counts only the number of external calls to LIB's f(), but does not count internal calls.

(3) If we want to wrap all calls to f() from CLIENT so that they are mediated via the stub() function of module CWRAP, we can use the following expression:

(hide (merge (rename CLIENT f stub) CWRAP) stub)

Note that only a particular client module is wrapped, without wrapping the service provider. In the example, renaming the client module's calls to f() produces the desired effect, since the declaration of f() as well as all self-references to it must be renamed.

Generalizing the above cases, the three varieties of wrapping possible in our model are shown pictorially in Figure 6.4. The leftmost column of the figure shows the given modules M1 and M2 and their wrappers W1 and W2. The top row shows a technique referred to as *method wrapping*, and the bottom row *call wrapping*. Figure 6.4(a) corresponds to example (1) above, Figure 6.4(b) to (2), and Figure 6.4(d) to (3) above.

The CLOS language supports a technique known as before-after methods to interpose calls to code before or after a particular method proper. The above

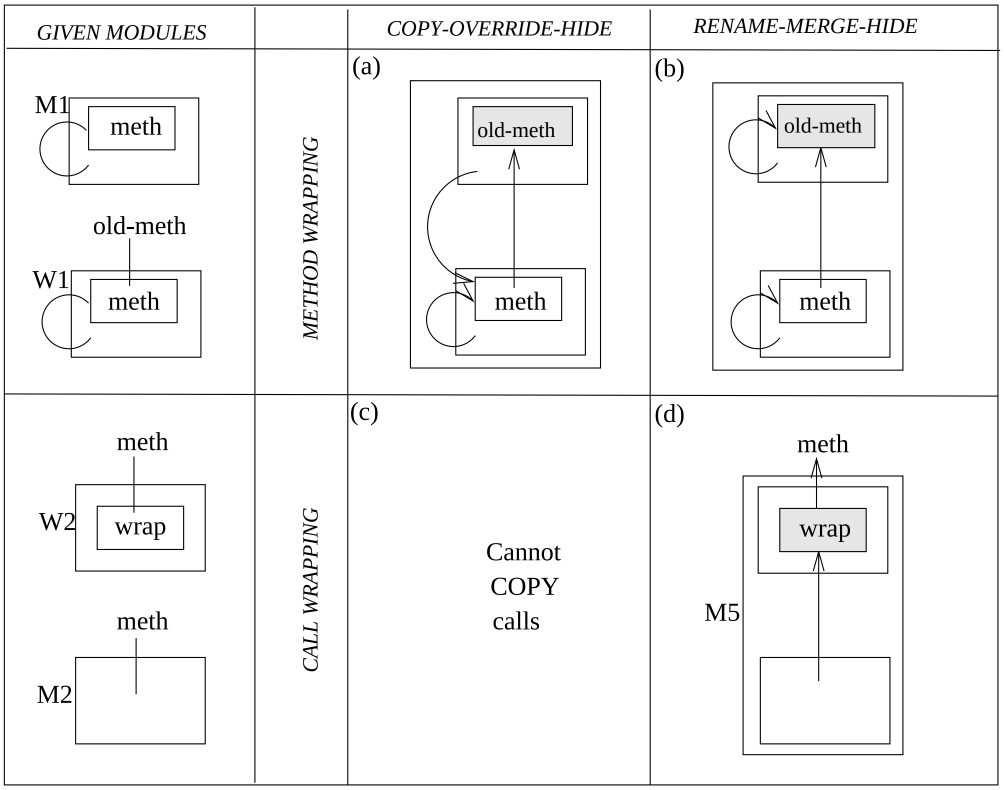


Figure 6.4. Wrapping scenarios. The leftmost column shows the given modules: M1 to be wrapped by W1, and M2 to be wrapped by W2. The top row shows the operations and effects of performing method wrapping, and the bottom row shows call wrapping.

notions of method wrapping and call wrapping can be extended to support calling of precompiled routines by generating and wrapping the appropriate adaptors. For example, to call a method bef in module B before a method meth in module M, we can generate a wrapper module W with a function meth that first calls bef and then calls the old definition of meth as old-meth. The modules M, W, and B can be combined in a manner similar to method wrapping to get the effect of a before-method:

```
(hide (override (copy-as M meth old-meth)
                (merge W B))
      old-meth)
```

6.3.2.2 Single and Multiple Inheritance

The idioms shown in Figure 6.4 are in fact the basis of inheritance in current day OO languages, as shown in Chapter 3. In this section, we give a brief idea of how these idioms can be used to achieve notions of inheritance.

Recall from Section 3.1.1 that a class can be defined using the macro define-class, which expands to a module expression that uses open-module and hide. A vehicle class was defined there. Using the same macro, a class can also inherit from another existing class.

For example, suppose a dot-o land_chars.o is created, which contains a global constant integer called wheels, and a function called display() that first calls a declared method called super-display(), then prints the value of wheels. Given such a module, a land-vehicle class can be created as a subclass of the previously defined vehicle module by writing:

```
(define-class land-vehicle "land_chars.o" (vehicle) ())
```

This macro expands to the module expression:

```
(define land-vehicle
  (hide (override (copy-as vehicle '("display")
                                   '("super_display"))
                  (open-module "land_chars.o"))
        '("super-display")))
```

In this expression, a module with attributes wheels and display is created and is used to override the superclass vehicle in which the display attribute is copied as super_display. The new display method can access the shadowed method as super_display. In general, all such conflicting attributes are determined by a meta-level primitive called conflicts-between and copied to a name with a super_ prefix. The copied super_display attribute is then hidden away to get a module with exactly one display method in the public interface, as desired. An important point here is that calls to display within the old vehicle module and the new land-vehicle module are both rebound to call the display method of the land-vehicle module.

The above idea of single inheritance can be generalized to multiple inheritance as found in languages such as CLOS [47]. In these languages, the graph of superclasses of a class is linearized into a single inheritance hierarchy by a language provided mechanism. A similar effect can be achieved with the define-class macro, except that the programmer must explicitly specify the order of the superclasses, as shown below:

```
(define-class land-chars "land_chars.o" () ())
(define-class sea-chars "sea_chars.o" () ())
(define-class amphibian
    "amphibian.o" (land-chars sea-chars vehicle) ())
```

In fact, explicit specification of linearization is more useful than an implicit, language provided mechanism. With the module operations supported by the module language, several other single and multiple inheritance styles can be expressed as well, as shown in Chapter 3.

6.3.3 Solving Old Problems

Using the operations defined on modules it is possible to conveniently solve long-standing problems in software engineering, encountered when using C, or C++. Several of these problems had solutions previously, but they were ad-hoc and/or required changes to source code. Module operations permit general solutions that impose no source code changes.

In this section, we delineate clean solutions to each of the problems enumerated in Section 6.1.2, in the same order.

(i) Wrapping calls. To solve the first problem of Section 6.1.2, the module spec for the image processing (IP) application can be written as given in Section

6.3.2.1, under call wrapping. Calls to edge_detect() can be wrapped with a wrapper method that first calls the function floattobyte() and then calls the edge_detect() library function.

(ii) Library extension management. The IP library can be thought of as an OO class, and incremental changes to it can be thought of as subclasses that modify the behavior of their superclasses. The subclasses can be integrated with the superclass by means of a module spec that uses the notions of inheritance illustrated in Section 6.3.2.

(iii) Static constructors and destructors. In C++, there is a need to generate calls to a set of static constructors and destructors before a program starts. Special code is added to the C++ front end to generate calls to the appropriate constructor and destructor routines. However, the order in which such static objects are constructed is poorly controlled in C++ and leads to vexing environment creation problems for large systems.

Under some variants of Unix, the C language has handled the need for destructors in an ad-hoc fashion, by allowing programs to dynamically specify the names of destructor routines by passing them to the atexit() routine. In other variants, the destructors for the standard I/O library are hard-coded into the standard exit routine. In neither case is there any provision for calling initialization routines (e.g., constructors) before program startup.

In both the cases of C and C++, module operations allow addressing the problem of generating calls to initialization or termination routines by using a general facility, rather than special-purpose mechanisms. As shown in Section 6.3.2.1 as before-after methods, module expressions can easily be programmed to generate a wrapper main() routine that calls all of the initialization routines found within that module, then call the real main() routine. Similarly, the exit() routine can be wrapped with an exit routine that calls all the destructors found in the module before calling the real exit().

(iv) Flat namespace. A longstanding naming problem with the C (and, to some extent C++) language has traditionally been the lack of depth in the program

namespace. C has a two-level namespace, where names can be either private to a module or known across all modules in an application. As a result, if an application uses library l1.a which imports symbols from another library l2.a, all symbols imported from l2.a are known by the application and become part of its exported interface.

With module operations, these problems can be avoided. Once a module that implements low-level functionality has been combined with a module that implements higher-level functionality, the functions in the former's interface can be subjected to the hide operation to avoid conflicts or accidental matches at higher levels.

6.4 Type-safe Composition

In this section, we present a technique to perform type checking of object modules within the architecture presented earlier in this chapter. The type system of specific languages can be incorporated into OMOS, and type information extracted from object modules can be used to ascertain the type safety of combining object modules. We describe in detail the realization of these steps for ANSI C, by utilizing standard debugging information generated by compilers.

6.4.1 Motivation

It is widely agreed that strong typing increases the reliability of software. However, compilers for statically typed languages such as C and C++ in traditional nonintegrated programming environments guarantee type-safety only within a compilation unit, but not *across* such units. (C++ style name-mangling does not accomplish complete type-safety across compilation units; see Section 6.6.) Long-standing and widely available linkers compose separately compiled units by matching symbols purely by name equivalence with no regard to their types. Such "common denominator" linkers accommodate object modules from various source languages by simply ignoring the static semantics of the language. (Commonly used object file formats are not even designed to incorporate source language type information in an easily accessible manner.)

Moreover, a programmable linkage facility such as OMOS enables the incorporation of automatic and user-defined type conversion routines for encapsulated data. For automatic conversion, we postulate safe adaptability rules for converting built-in data types using the language definition in conjunction with the characteristics of particular hardware platforms. We then utilize these rules to automatically generate data conversion adapters at link time. More importantly, programmer defined conversion stubs can also be easily incorporated at link time. This opens up the possibility of programmer-controlled data evolution and conversion across heterogeneous data formats, e.g., those arising from different languages, hardware architectures, etc.

Link-time type-checking helps one to adapt and utilize the full expressive power of language type systems to better suit modern persistent, distributed, and heterogeneous environments. For example, *structural* typing can be applied to languages such as ANSI C with *name-based* typing. Pure name-based typing becomes a problem in persistent and distributed environments, where data and types could migrate outside the program in which they were originally created [2] and lead to matching of names that may or may not have the same programmer-intended meaning. This argues for structural matching of aggregate types similar to Modula-3 [59], using member order and type significance along with names.

6.4.2 A Scenario

As before, an ANSI C program source or object file is considered a module consisting of a set of attributes with no order significance. The interface consists of the types of the attributes of the module, along with information as to whether each is defined or merely declared. For example, Figure 6.5 shows a module O1 whose interface consists of one attribute

$\{$ **define** $\mathsf{f} : \mathsf{void} \rightarrow \mathsf{struct\ S}$ $\}$

and a module O2 whose interface consists of two attributes

$\{$ **define** $\mathsf{g} : \mathsf{void} \rightarrow \mathsf{void}$,
declare $\mathsf{f} : \mathsf{void} \rightarrow \mathsf{int}$ $\}$

Consider the case where a programmer creates and compiles module O2 with the intention of using O1's f definition by performing O1 merge O2, but makes

Module O1	Module O2	Module O3

struct S {
int x;
/* ... */
}
struct S f () {
/* ... */
}
	extern int f ();
	void g () {
	int x = f ();
	}
		struct S {
		int x;
		/* ... */
		}
		extern struct S f
		();
		int f_stub () {
		return f().x;
		}

Figure 6.5. Linkage adaptation. Modules O1 and O2 are composed with the expression: (O2 rename f f_stub) merge O3 merge O1

the incorrect presumption that f returns an int. If merge were untyped (as it is in common linkage), O1 merge O2 would have been legal, with disastrous results. However, it should not typecheck, since the interfaces of O1 and O2 are not type compatible for a merge operation: the return types of f are not related.

Suppose that the programmer of O2 discovers during linkage that f returns the desired int value as a component of the returned structure. In the traditional scenario, in order to make O1 and O2 compatible, the programmer would have to modify the source code of either module extensively and recompile. This might not be possible for precompiled libraries. Even if it were possible, it could adversely affect combination of the modified module with yet other modules.

Programmability supported by the module management facility is crucial to alleviate this problem. In this example, O2 can be adapted by constructing an adapter module O3. O3 consists of a declaration of f that matches its definition in O1 and a stub function f_stub that extracts the desired value from the structure returned by f. With this, a modified version of O2 can be obtained with the module expression (merge (rename O2 f f_stub) O3), which can then be merge'ed with O1 to get the originally desired effect. (This effect is similar to call wrapping explained in Section 6.3.2.) If a type error cannot be corrected with such transformations on object modules, it might indicate a more serious error in the design of the modules involved.

Furthermore, the process of wrapping procedures as given above is enhanced by the availability of module type information. The wrapper procedure can sometimes be automatically constructed with a signature identical to that of the wrapped procedure, and simple language constructs can be used to propagate the caller's arguments to the wrapped routine. If type information was not available (or in cases such as `printf` where the routine is defined to take a variable number of arguments) it would be necessary to use a machine-dependent wrapper that preserves and passes along the call frame without knowledge of its contents.

6.4.3 C's Type System

In order to ascertain the type-safety of modules being combined, the module type rules (given in Section 2.2.2) built into the linker requires knowledge of the type system (type domain, type equivalence and subtyping) of the base language ANSI C. As before, attributes *match* if they have the same name. There cannot be matching attributes within a single interface, and attributes that match across interfaces must be type compatible.

This section describes the relevant type system of ANSI C (type domain and type equivalence) [46] and enhancements made to it for type-checking across compilation units (structural typing, and function subtyping).

The type domain of ANSI C consists of (i) basic types (primitive types (int, float, etc.), and enumerated types), (ii) derived types (function types, struct and union types, array and pointer types), and (iii) typedef'ed names. Specifiers for these types can be augmented with type qualifiers (const and volatile) and storage class specifiers (auto, register, static and extern).

The type qualifier volatile concerns optimization and is not relevant here. The type qualifier const is significant for typing since it distinguishes read-only variables from read-write variables; it will be explicitly dealt with later in this section. The storage class specifiers auto and register are not relevant since they may only be used within functions — we are interested in file-level declarations and definitions. The storage specifier extern indicates an attribute *declaration* whereas nonextern attributes are considered to be *defined*. The storage specifier static for a file-level attribute gives it internal linkage, i.e., the attribute can be viewed as having been subjected to a hide module operation. Similarly, attributes that are subjected to hide via link-time programming can be regarded as having been converted to the static storage class after the fact.

C permits calls to functions that have not been declared in a module. A call to an undeclared function f in a module results in an implicit file-level declaration of extern int f ().

6.4.3.1 Type Equivalence

Type equivalence in ANSI C within a single translation unit and our extensions for type-checking across translation units are given in Table 6.1. There are two changes to the defaults. As mentioned earlier, for aggregate types (struct's and union's), name equivalence is too weak when applied outside of a single translation unit, as argued earlier. Therefore, we adopt a conservative structural typing regimen in which the names, order and types of members are also significant. We also retain the significance of aggregate tags since there could be application-specific semantic content in them. Second, for typedef'ed names, again, there could be application-specific semantic content in them, so we adopt strict name equivalence.

Furthermore, some type specifiers are implied by others, e.g., short implies short int; therefore these types are equivalent. Also, the type qualifier const is significant for equivalence.

Table 6.1. Type equivalence in ANSI C.

Type	Within translation unit	Across translation units
Primitive	name equivalence	same
Function	structural, with in and out parameter types significant	same
Enum	name equivalence	same
Struct/union	name (tag) equivalence; tag-less types are unique	structural, with tag, member order & names significant
Pointer	equivalence of target types	same
Array	equivalence of element types & equality of array size	same
Typedef	typedef'ed type	typedef name equivalence

6.4.3.2 Subtyping

The module operators merge and override utilize subtyping rules for type-checking combination. Our base language, ANSI C, has no notion of subtypes; hence subtyping can be considered to be restricted to type equivalence. However, module composition would be more flexible if we could retroactively formulate subtyping rules consistent with the ANSI C language definition.

The ANSI C language specifies safe conversion rules for certain primitive arithmetic data types (e.g., float to double). A conversion is said to be safe if all values of one type can be represented as values of the other without loss of precision or change in numerical value. C compilers, however, can usually be expected to support many more safe conversions than those that are defined by the language, as governed by hardware characteristics. These safe conversion rules can be thought of as *subsumption* rules, which in turn provide the basis for formulating subtype rules for primitive arithmetic types. Figure 6.6 shows the data type sizes and a partial order of subtypes for the HP series 9000 machines (300s and 700s). For instance, a value of type short can be safely coerced into a value of type float on this platform without loss of precision or change in numerical value. We might ask if the above rules can be exploited during type-checking of attributes across translation units.

Consider file-level variable declarations. Variables can be used as evaluators (i.e., expressions that return values) and as acceptors (i.e., expressions that receive values) in different contexts. Expressions that are evaluators can only be replaced with expressions whose types are subtypes of the original, whereas expressions that are acceptors can only be replaced by expressions whose types are supertypes of the original [10]. As a result, subtyping of variables must always be restricted to type equivalence.

Consider file-level read-only (i.e., const) variables. Subtyping involving the type qualifier const can be described as follows: if a non-const type s is a subtype of a non-const type t, then const s is a subtype of const t, s is a subtype of const t, but const s is *not* a subtype of t. So, for example, is a declaration extern const float x in one translation unit be considered a subtype of a definition short x in another,

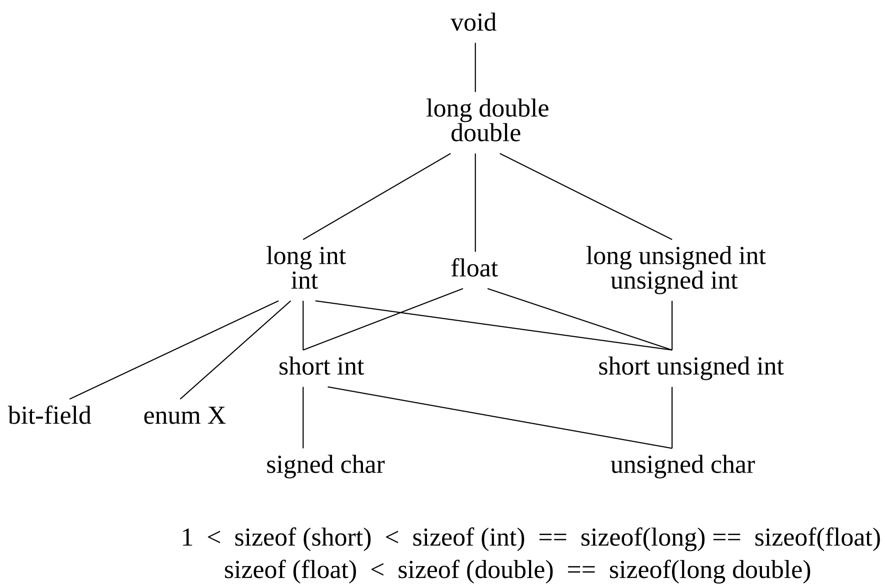


Figure 6.6. Subtyping of C primitive data types.

and thus safely combined?

Unfortunately, this is not the case, since size and layout formats for various primitive data types are usually incompatible, thus requiring active coercion. Moreover, in certain cases, e.g., enum types, compilers usually optimize layout by packing, hence the fact that an enum type is really an int cannot be utilized. Within the *same* translation unit, however, such subsumption rules can be applied since the compiler has complete knowledge of layout and usage and hence it can generate appropriate conversion and access code.

Similar arguments hold for subtyping constant user-defined aggregate data types (struct and union) across translation units. For example, a struct of two shorts cannot be considered to be a subtype of a const struct of two const floats even though short is a subtype of const float. Furthermore, C unions are not discriminated, and member access is not type-checked at run-time. For example, a union with one short component cannot be read-only accessed by a supertype, a union with a const short and a const float component, in another translation unit, since there is no way for the supertype accessor to know at run-time if the union actually contains a short value or a float value. As a result, subtyping on file-level read-only variables is also restricted to type equivalence.

Arguments such as the above can be formulated to show that subtyping on pointer types is also restricted to type equivalence.

Consider subtyping of function types. Subtyping of function types is by *contravariance* [10]. That is, a function type is a subtype of another with the same number of arguments if its return type is a subtype of the latter's, and its input argument types are *supertypes* of the corresponding ones in the latter. According to this rule, one can pass a function actual parameters that are *subtypes* of the formal parameters in the function definition. For subtyping function types with an unspecified (variable) number of arguments, we require that the subtype has at most the number of explicitly specified argument types in the supertype and that they are in the proper relationship.

However, we cannot use such a rule across translation units since in a compiled function, the amount of space allocated for the input parameters is exactly the size of the expected types, and the format is expected to be exactly as specified. All in all, and not surprisingly, *no* useful subtyping rules can be discovered in the existing C language for *direct* application in type-checking across translation units.

The crucial observation, however, is that several useful subsumption rules can be utilized for data that are *encapsulated* within functions, *if* adapters ("stubs") that perform the appropriate coercion between data-types can be inserted between combined modules at link time. This is feasible since such stub functions are themselves compiled and hence they can utilize data format conversion knowledge that a compiler uses within a translation unit. Applying this stub technique to global *data*, however, is not feasible since it involves initializing global variables with nonconstant values, which is illegal in ANSI C.

Function types lend themselves particularly well to this technique since the performance of function calls is affected much less by this indirection than the performance of data access. Moreover, it does not seem unreasonable to impose the requirement on users to encapsulate such data that they foresee will be accessed via supertypes.

In the architecture presented here, it is possible to automatically generate coercion stubs for functions using the primitive type conversions shown in Figure 6.6. For an example of type adaptation using language defined subtypes, consider Figure 6.7. As mentioned earlier, the type short is a subtype of float. Therefore, the definition of function f in module O1 is a subtype (by contravariance) of the declaration of the function f in module O2. However, O1 cannot be directly merged with O2, since in general the calling sequence for f might not be compatible; e.g. the definition of f might be expecting its input in a floating point register rather than an integer register. This is remedied by first combining O2 with the automatically generated stub module O3 that incorporates safe coercions and then performing the desired merge, as shown in the figure.

Structural record subtyping with member name, type, and order significance is also possible; an example is shown in Figure 6.8. It should be emphasized that the above technique applies only to input and output parameters of functions, since coercion stubs can be automatically generated to account for function subtyping only.

This technique of type conversion stubs can be generalized as illustrated in Figure 6.9 to provide a general facility to incorporate user defined stubs at link time for arbitrary data format conversion. In the figure, module O3 comprises user-defined stubs.

<table>
<tr><th>Module O1</th><th>Module O2</th><th>Module O3
(automatically generated)</th></tr>
<tr><td><pre>short f (float y) {
  /* ... */
}</pre></td><td><pre>extern float f (short);
void g () {
  float z = f (3);
}</pre></td><td><pre>extern short f (float);
float f_stub (short x) {
  return (float) f ((float) x);
}</pre></td></tr>
</table>

Figure 6.7. Automatic data coercion using language rules. Modules O1 and O2 are combined with the expression: (merge (hide (merge (rename O2 f f_stub) O3) f_stub) O1)

Module O1	Module O2	Module O3 (automatically generated)

struct S {
  short x;
  float y;
}
struct S f () {
  /* ... */
}
	struct S {
	  float x;
	}
	extern struct S f ();
	void g () {
	  /* ... */
	}
		struct S1 {
		  short x;
		  float y;
		}
		struct S {
		  float x;
		}
		extern struct S f ();
		struct S f_stub () {
		  struct S1 s1;
		  struct S* s = (struct S*) &s1;
		  struct S ret_s;
		  *s = f ();
		  ret_s.x = (float) s->x;
		  return ret_s;
		}

Figure 6.8. Automatic conversion of structs using structural subtyping. Modules O1 and O2 are combined with the expression: (merge (hide (merge (rename O2 f f_stub) O3) f_stub) O1)

Module O1	Module O2	Module O3

R1 f (T1 y) {
  /* ... */
}
	extern R2 f (T2);
	void g () {
	  R2 z = f (/*T2 value*/);
	}
		extern R1 f (T1);
		R2 f_stub (T2 x) {
		  return R1_to_R2 (f (T2_to_T1(x));
		}
		R2 R1_to_R2 (R1 r) {
		  /* ... */
		}
		T1 T2_to_T1 (T2 t) {
		  /* ... */
		}

Figure 6.9. Programmer-defined data conversion. Modules O1 and O2 are combined using: (merge (merge (rename O2 f f_stub) O3) O1)

6.5 Implementation

6.5.1 OMOS

Just like the interpreter for *CMS*, it is possible to design and implement OMOS as a completion of the Etyma framework. However, we shall refer to the OMOS implementation presented here as a "parallel completion" of Etyma, for the following reasons.

The implementation of OMOS described here is not, strictly speaking, a completion of Etyma in the same way that the *CMS* interpreter is, since it does not *directly* reuse the code in Etyma. Nevertheless, the *design* of OMOS closely *follows* the class design of Etyma. Moreover, concepts of compositional modularity first developed with respect to Etyma and *CMS* were directly reused in the context of OMOS.

The first generation implementation of OMOS [63] existed much before Etyma was born. Later in its lifecycle, OMOS was reengineered to incorporate several major design enhancements. During this, its "upper" class hierarchy was made to follow the Etyma class design. However, it was not designed to be a direct completion of Etyma, since the two pieces of software had already significantly diverged in terms of, for example, conventions and management infrastructure. Nonetheless, there is conceptually no reason why OMOS could not be implemented as a first-class completion of Etyma.

The OMOS class hierarchy consists of classes corresponding to the Etyma classes *Module* and *Instance*. The notion of an object file is implemented as a concrete subclass DotO of *Module*. The methods of class DotO are implemented using a code library that provides relative independence from particular object file formats, the Binary File Descriptor (BFD) library [18] from Cygnus Corporation.

As mentioned earlier, a dot-o is instantiated into fixed executables, modeled by class FixedExe, a subclass of *Instance*. A fixed executable is internally represented as an address map. An address map is a collection of entries that specify the address in the virtual memory of a process that a block in an object file is mapped to.

The front-end to OMOS is a Scheme interpreter derived from the STk package (similar to *CMS*). New primitives added to the Scheme interpreter instantiate the appropriate classes such as DotO and FixedExe above and invoke their appropriate methods.

6.5.2 Type-safe Linkage

The object module type-checker given in Section 6.4 was built by reusing both the design and the code of an early version of the ETYMA framework.

Ideally, we would like compilers that generate object modules in a self-describing format, with information about the source language, the machine architecture, and the interface, all packaged within the object module in a readily accessible format. However, this is far from reality — the closest approximation is an object file that has been compiled with the debugging option -g, which instructs the compiler to generate type information in a standard encoded format. (Object files compiled without the debugging option contain no type information, and those compiled with the debugging option contain more information than is necessary for type-checking linkage, e.g., types of local variables, line numbers, etc.)

Extracting type information from the generated debugging information involved the following steps in our prototype. The GNU C compiler, gcc, does not generate debugging information for C extern symbols, since debugging is normally performed on executable files in which all external references have been resolved. To solve this, we modified the back end of gcc to generate debugging information for all symbols. For accessing the sections of the object file that contain debugging information (.stab and .stabstr), we again use the BFD library and parse the "stabs" format debug strings [56] using a yacc/lex generated parser.

The parser instantiates the appropriate subclasses of ETYMA classes to create the interface of the object module. For instance, the subclass CPrimType of the framework class *PrimType* implements the partial order of primitive types introduced before. A subclass DotOInterface of the framework class StdInterface implements the concrete notion of the interfaces of dot-o files. Similar subclasses exist for representing function types, struct types, etc.

For using the type-checking facility, source programs must be written in ANSI

C, and function declarations specified using "new-style" prototypes. Furthermore, usage of header files can be minimized; explicit declarations of external functions can be provided instead. Programs that are to be type-checked at link time must be (re)compiled with our (modified) compiler using the debug (-g) option.

One legitimate concern is the size of object files as a result of the inclusion of debugging information. The size of object files does increase significantly due to debugging information, but this problem is exacerbated by the inclusion of huge library header files. Our solution to this problem is that given type-checking at link-time, it is not necessary to include header files in the traditional way. Instead, programs can explicitly declare prototypes for those external (library) functions that are called. A discussion of the disadvantages of header files used in the traditional manner is found below in Section 6.6.

6.6 Related Work

This work is in essence a general and concrete realization of a vision due to Donn Seeley [71].

6.6.1 OMOS

Traditionally linkers support little control over name conflicts and the semantics of combination. As mentioned earlier, traditional linking essentially amounts to what is supported by merge.

Although programmable linkers exist, they do not offer the generality and flexibility of our system. Typically, some notion of hiding is supported. However, it is novel to support an expressive suite of combination operators and the ability to use a full-featured programming language to compose object modules.

From the systems point of view, a user-space loader such as OMOS is considered no longer unusual [69, 31]. Also, OMOS has some similarity to utilities such as dld [38] that aid programmers in the dynamic loading of code and data. Finally, the Apollo DSEE [3] system was a server-based system which managed sources and objects, taking advantage of caching to avoid recompilation. However, DSEE was primarily a CASE tool and did not take part in the execution phase of program development.

The module language of OMOS is somewhat similar to architecture description languages, such as Rapide [44], the POLYLITH Module Interconnection Language (MIL) [12, 66], and OMG's Interface Definition Language (IDL) [60]. These languages all share the characteristic that they support the flexible specification of high-level components and interconnections. Our approach offers the important advantage that OO like program adaptation and reuse techniques (inheritance, in all its meanings) can be applied to legacy components written in non-OO languages.

An environment for flexible application development has been pursued in the line of research leading to the so-called subject-oriented programming (SOP) [37]. In this research, a "subject" is in essence an OO component, i.e., a component built around an OO class hierarchy. Subjects can be separately compiled and composed using tools know as "compositors" (similar to OMOS). Compositors use various operators similar to the ones presented here. The primary difference between SOP and our research is that SOP is broadly conceived around the OO nature of individual components, and aims to build a toolset and object file formats specifically tailored for SOP. On the other hand, our research has focussed on layered evolutionary support.

6.6.2 Type Safety

Integrated Development Environments (IDEs) for strongly typed languages, e.g., Eiffel [58], utilize mechanisms for type-checking separately compiled modules, since they have complete knowledge and control over source and object modules. However, our work differs from IDEs in that we provide a systemwide linkage facility that attempts to typecheck combined modules independent of language processors. Furthermore, the programmability of our linker enables "fine tuning" the compatibility of (possibly heterogeneous) object modules at link time.

The Berkeley Pascal Compiler pc [43] is similar to our effort in that it employs debugging information to check type consistency across separately compiled modules. The compiler routinely generates stab-format type information into object modules, which is used by a binding phase of the compiler to check consistency before delegating the actual linking to ld. However, the crucial advantage with our approach is that we perform type-checking as a controlled and programmable link-time activity.

Use of header files has been a longstanding attempt at type-safety of separate compilation. The *Annotated C++ Reference Manual* [28, page 122] explains the inadequacy of header files as follows:

> ... C tried to ensure the consistency of separately compiled programs by controlling the information given to the compiler in header files. This approach works fine up to a point, but does involve extra-linguistic mechanisms, is usually error-prone, and can be costly because of the need to have other programs (in addition to the linker and the compiler) know about the detailed structure of a program.

Instead of including header files, it is clearly more modular and less error-prone to explicitly declare the expected external functionality (e.g., library functions), let the linker check consistency at link time and correct inconsistencies via programming.

With the objective of enabling type-safe linkage within the constraints of existing linkers, Stroustrup [74, 28] describes a mechanism for encoding functions with the types of input arguments. However, this mechanism is inadequate for our purposes since (i) certain classes of type errors cannot be detected (page 126 of [28]) since variable types and function return types are not encoded; (ii) although it could be extended to deal with structural typing of C aggregate types, it does not scale well to arbitrarily large types, e.g., large structs; and (iii) we want to do not only type-checking, but also useful adaptation during link-time; hence we must utilize sophisticated linker technology.

There is a plethora of literature related to stub generation[6, 5, 75]. The Polygen system [12] is representative of automatic stub generation for programming in a heterogeneous environment. Polygen packages heterogeneous modules by utilizing a programmer-defined specification of their interfaces and execution environments specified in a common module language. The packaging process involves generation of client and server stubs that handle module interconnection and data type coercion dynamically. Our technique differs from Polygen in that we enable the combination of precompiled object modules by automatic extraction of interfaces and via link-time programming.

6.7 Summary

We have argued that conventional application development environments lack support for the effective management of application components. We illustrate that the problems faced by application builders are similar to those that are solved by the concepts of OO programming. We thus conclude that it is beneficial to support OO functionality within the component manipulation and binding environment.

We show that support for OO development can be achieved by elevating the physical modularity (i.e., separately compiled files) of application components to the level of logical modularity, managed by a systemwide server process. The server supports a module language based on Scheme, using which first-class modules can be manipulated via a powerful suite of operators. Expressions over modules are used to achieve various OO effects, such as encapsulation and inheritance, thus directly supporting application development in an OO manner. In this manner, we enable a superior application development environment within a conventional infrastructure.

In addition, we have described a programmable linkage facility for separately compiled ANSI C object modules. We design the type system of ANSI C into our linker and typecheck composition by extracting the interfaces of object modules compiled with debugging information. Furthermore, we automatically generate conversion stubs for compatible encapsulated types and permit easy incorporation of arbitrary user-defined type conversion stubs at link time. We have thus demonstrated a powerful, flexible, and type-safe linkage facility.

CHAPTER 7

INTERFACE COMPOSITION

It is widely believed that the complexity of modern distributed systems requires a sophisticated language (or sublanguage) to explicitly specify the interfaces through which application components interact. Service providing components are said to *offer* interfaces, while client components *invoke* them, possibly remotely.

The requirements of modern distributed systems necessitate support not only for the inheritance of component implementations but also for the explicit specification, adaptation, and combination of interface specifications themselves. This suggests that interfaces can be regarded as compositional entities, making it possible to apply composition operators to interface composition.

This chapter explores this third application of compositional concepts. Section 7.1 develops the idea that interfaces are compositional entities and presents an IDL compiler front-end as a completion of ETYMA. Section 7.2 sketches how existing IDLs can be extended with compositional concepts.

7.1 Software Architecture Description

Developing large, complex, and concurrent systems requires significant effort spent on describing their *architecture*, both as an iterative design activity as well as for documentation purposes. Informal architecture description via pictures is a frequent practice in system design. Nonetheless, formal and machine-processable descriptions are becoming necessary and widespread, especially in the context of distributed systems.

An architecture aims to describe the components of a system and their relationships, such as "is-a," "has-a," and "communicates-with." These properties are usually described via architecture description languages (ADLs) [54, 1] of varying degrees of expressive power. A description of the conventional typed interfaces of components is a simple example of an architecture description. A more sophisticated description, with its concomitant benefits and complexities, might specify the typed interfaces along with their structural (e.g., "component-of"), behavioral (e.g., invariant), and interaction (e.g., event description) relationships.

A family of languages known as interface definition languages (IDLs) are becoming increasingly popular in the arena of distributed systems and is described in the following subsection. Although there is a wide variety of IDL designs, an IDL typically provides a subset of the functionality of an ADL.

7.1.1 Interface Definition Languages

Components of modern distributed systems are usually written in conventional programming languages such as C and Fortran, which we shall call component programming languages, or CPLs. These languages usually support some form (however weak) of specifying the interfaces of components, e.g., header files in C. However, many distributed systems, e.g., refs. [60, 4, 44], support a separate IDL distinct from CPLs.

Explicit interface descriptions are useful for various purposes:

(1) Specification. To specify the contractual obligations between a service providing component and its clients, and to ascertain that they are met. Minimally, IDLs employ conventional programming language types for the specification of interfaces, although some IDLs, e.g., ref. [70], support much stronger forms of behavior specification than others.

(2) Interoperability. As an intermediate language to facilitate interoperability among components in heterogeneous, e.g., hardware or programming language, environments. There are usually several "language mappings" from an IDL to individual CPLs, or vice versa (see below).

(3) Implementation. As a source language for the automatic generation of communication code between components. An IDL compiler usually reads in an IDL specification and generates "stub" code that performs packaging ("marshal-

ing"), communication, and unpackaging ("unmarshaling") of arguments for remote procedures calls (RPC).

If the specificational power of an IDL does not exceed that of conventional programming language type systems, the question arises as to the need for supporting a separate IDL that is human-readable. It can be argued, as in ref. [4], that instead, the type systems of individual CPLs can be augmented with constructs to specify interoperability requirements and used to generate specifications in a purely machine-readable IDL. This could be beneficial in not burdening the programmer with yet another programming language and perhaps for achieving a high degree of interoperability. However, designing extensions to each existing CPL is considerably harder than introducing a new, CPL-neutral IDL. Moreover, eliciting the acceptance of new language extensions from a community of users of an existing language is usually problematic.

In the rest of this chapter, we will consider only that aspect of IDLs that supports typed interfaces and ignore interoperability and adapter generation aspects.

7.1.2 Compositional Interfaces

Having established the role and purpose of a separate, human-processable IDL, consider the expressive power desired of such a language.

It may often be useful to specify an interface by *reusing*, i.e., inheriting from, existing interfaces. Reuse facilitates the evolution of interfaces [34] by ascertaining that inheriting interfaces evolve in step with the inherited interfaces. It also simplifies maintenance by reducing redundant code. Most importantly, an IDL should be able to express the types of components generated via implementation inheritance in CPLs. In fact, it has been shown that inheritance of interfaces generates exactly those types, known as *inherited types*, that correspond to the types of inherited objects [23]. (Inherited types are distinct from subtypes, see below.)

These reasons point to a need for flexible interface inheritance mechanisms in IDLs. Several existing IDLs, e.g., refs. [60, 44], support some form of interface inheritance. However, some IDLs, e.g., CORBAs IDL [60] described in Section

7.2, unnecessarily limit the expressiveness of their interface inheritance mechanism. Thus, support for compositionality of interfaces can prove to be useful, especially since it can be layered on top of existing IDLs, including those that do not already support inheritance. In order to apply compositional concepts to interfaces, however, we must first establish that an interface can be regarded as an abstracted namespace.

An interface represents the type of a component. An interface thus comprises a set of names associated with the types of subcomponents. The particular domain of base types (i.e., non-interface types) used in an IDL is of course chosen by the IDL designer. Furthermore, interfaces may be recursive; i.e., a type constituent may refer back to the entire interface itself.

For example, consider a Point module that contains attributes corresponding to rectangular coordinates x and y, a method move for changing the position of the point and an equality predicate equal. Its interface may be expressed as follows, where recursion is expressed using the selftype keyword.

```
interface PointType {
  float x, y;
  selftype move (float, float);
  boolean equal (selftype);
};
```

In general, an IDL may support interface type constituents that refer to sibling type constituents, similar to the notion of sibling member access via *self* in objects. For example, the point interface above may be specified as follows:

```
interface FloatPointType {
  float x, y;
  selftype move (selftype.x, selftype.y);
  boolean equal (selftype);
}
```

In this manner, the type of the move operation can track changes to the x and y constituents. As a convenience, selftype.x may be abbreviated to x.

Inheritance is an operation on recursive structures; thus it can be applied to interfaces as well. For instance, the interface PointType above can be extended to have a color attribute using the merge operation, as follows:

```
interface ColorType {
  color_type color;
};
interface ColorPointType = PointType merge ColorType;
```

Although it inherits from PointType, the ColorPointType interface is not a subtype of PointType, due to the contravariance of the equal method. That is, in order for ColorPointType to be a subtype of PointType, the incoming argument type of equal must be a *supertype* of the correspoding one in PointType. This is not so. However, ColorPointType shares the same structure as PointType; hence it is known as an *inherited type* of PointType.

An important point to note here is that the merge operation on interfaces generates types that correspond to the types of inherited module implementations generated via both the merge *and* override operations in CPLs.

An override operation is defined on interfaces as well, but it does not produce types directly corresponding to the override operation on CPL module implementations. With the interface override operation, type constituents of interfaces may be arbitrarily rebound. The primary motivation for including such an operator is to support a high degree of reuse of existing interface specifications.

In the following example, the x and y constituents of FloatPointType are rebound to complex_type; note that this will automatically result in the proper type for the move constituent, due to self-reference.

```
interface ComplexPointType =
  FloatPointType override
    interface {
      complex_type x, y;
    };
```

Type constituents may be rename'd, which results in self-references to get renamed as well. This is useful for resolving name conflicts while performing operations equivalent to multiple inheritance. Furthermore, particular interface constituents may be project'ed. This operation is analogous to the notion in relational algebra and is the dual of the restrict operator presented earlier.

The operator copy-as does not seem very useful in the context of interfaces. Also, the operators freeze and hide do not apply, since interfaces by definition represent the public types of modules. In the following section, we provide a formal characterization of interface inheritance in order to clarify the above notions.

7.1.3 Type Generators

An interface is a recursive structure [23, 10] corresponding to what we shall call a *type generator*. Type generators are analogous to ordinary generators given in Section 2.2.1, except that they are functions over records in which labels are bound to types viewed as values. In this sense, types are first-class within IDLs. Our description of type generators and their manipulation in this section is independent of particular type domains.

We shall notate a type generator as $\Lambda s.\ \{a_1 : \tau_1, \ldots, a_n : \tau_n\}$, where $a_1 \ldots a_n$ are attribute names bound to the types $\tau_1 \ldots \tau_n$ respectively. Note that s here stands for the notion of *selftype*, by which a type constituent τ_x can refer back to the entire interface.

Thus, the interface PointType given in the previous section can be written as:

$$PointType = \Lambda s.\{x : Real, y : Real, move : Real \times Real \rightarrow s, equal : s \rightarrow Bool\}$$

A CPL may support a type system in which the type of the Point module may be directly given by the above recursive type. In this case, instances of the Point module have a type given by the fixpoint of the above type generator. In the case of a CPL that does not support recursive types, the type of modules themselves may be given by the fixpoint of type generators such as the above.

Compositional inheritance can be used to adapt and combine type generators. The merge operator combines type generators that do not have any conflicting attributes, as described by the following deduction rule:

$$\frac{\begin{array}{rl} \Gamma \vdash\ g_{t_1} : \Lambda s.\ \{ & a_1 : \alpha_1, \ldots, a_n : \alpha_n\ \}, \\ \Gamma \vdash\ g_{t_2} : \Lambda s.\ \{ & c_1 : \gamma_1, \ldots, c_p : \gamma_p\ \}, \\ \forall i \in 1 \ldots n,\ \forall j \in 1 \ldots p, & a_i \neq c_j \end{array}}{\begin{array}{rl} \Gamma \vdash\ g_{t_1}\ \mathsf{merge}\, g_{t_2} : \Lambda s.\ \{ & a_1 : \alpha_1, \ldots, a_n : \alpha_n \\ & c_1 : \gamma_1, \ldots, c_p : \gamma_p\ \} \end{array}}$$

Equality and subtyping of base types are given by the base language and are not specified here. For interfaces, which are recursive types, subtyping is determined structurally, as described by the following rule (algorithms for subtyping recursive types are given by Amadio and Cardelli [2]):

$$\frac{\begin{array}{rl}\Gamma \vdash & g_{t_1} : \Lambda s.\ \{a_1 : \alpha_{11}, \ldots, a_k : \alpha_{1k}, \ldots, a_n : \alpha_{1n}\ \}, \\ \Gamma \vdash & g_{t_2} : \Lambda s.\ \{a_1 : \alpha_{21}, \ldots, a_k : \alpha_{2k}\}, \\ \Gamma \vdash & \forall i \in 1 \ldots k,\ \alpha_{1i} \leq \alpha_{2i} \\ \Gamma, \tau_1 \leq \tau_2 \vdash & g_{t_1}(\tau_1) \leq g_{t_2}(\tau_2)\end{array}}{\Gamma \vdash \quad g_{t_1} \leq g_{t_2}}$$

As an example of the merge operation, merging the type generator $PointType$ above with

$$ColorType = \Lambda s.\ \{c : Color\}$$

produces the type generator

$$\Lambda s.\ \{x : Real, y : Real, move : Real \times Real \rightarrow s, equal : s \rightarrow Bool, c : Color\}$$

As mentioned earlier, this type is not a subtype of $PointType$ due to the contravariance of the *equal* method. Instead it is an *inherited type*. It is important to understand that type inheritance does not necessarily produce subtypes.

One of the main motivations for the merge operator is to be able to generate interfaces that correspond to inheritance in the module implementation language. The $\mathsf{override}$ operator for type generators, defined exactly as for object generators, permits labels to be rebound to arbitrary types:

$$\Gamma \vdash g_{t_1}\ \mathsf{override}\ g_{t_2} : \Lambda s.\ g_{t_1}(s) \leftarrow_r g_{t_2}(s)$$

The rename and $\mathsf{restrict}$ operators have the usual semantics and can be specified entirely analogously to what is given in Figure 2.2. As mentioned before, the operator $\mathsf{project}$, the dual of $\mathsf{restrict}$, can be quite useful in the context of interface reuse, and the copy, freeze, and hide operators are not defined.

Compositional nesting can be supported for type generators as well, by developing closed type generators. However, it is doubtful that such expressiveness would be very useful in the context of IDLs.

Finally, it is worth mentioning that types are applicative structures in the context of IDLs. That is to say, the manipulation of type generators can be viewed as applicative operations, i.e., with copy semantics, and there is no notion of assignable type variables.

7.1.4 An Experimental IDL

In this section, we describe the implementation of an experimental compositional IDL based on the concepts described in the previous two sections.

The base type domain of the language consists of primitive types, function types, and record types. Interfaces in this language follow a structural type discipline. Furthermore, interfaces can be recursive, in that a component type of the interface can use the keyword selftype to refer to its own interface. Interfaces can be composed using the operators merge, override, rename, and project with the semantics given in the previous two sections. Examples of specifications were also given in Section 7.1.2.

A compiler front-end for this language was designed as a completion of the Etyma framework. The implementation architecture is shown in Figure 7.1. The front-end parses the IDL source and builds up an internal representation that can be used to further process the interface description as desired, e.g., to produce communication stubs.

Briefly, the class design for this completion is as follows. Define class IDLInterface as a subclass of class StdInterface, and define methods merge (IDLInterface), rename (Label,Label), etc. to return new interface objects after performing the appropriate operations. The type equality and subtyping methods of StdInterface can be reused directly in the IDLInterface class.

For implementing the base types, create a subclass IDLFunctionType of Func-

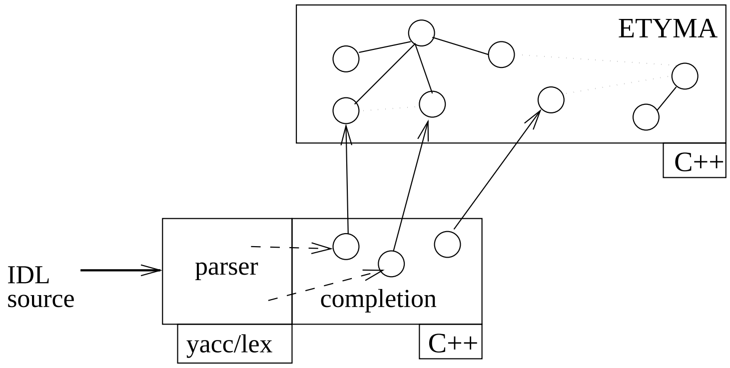


Figure 7.1. Architecture of the IDL front-end.

tionType to model function-valued attributes (operations) of interfaces. Subtyping is by contravariance for operations. To keep the language design simple, we can define subtyping to be type equality for all the other types. Subclass NamedType to IDLPrimType to model the primitive types of the IDL. For struct types, which have type identity in IDL, subclass IDLStructType from BrandedRecType, itself a subclass of RecordType.

Design and code reuse numbers for this completion are given in Table 7.1.

7.2 Making CORBAs IDL Compositional

As an illustration of how an existing IDL can be enhanced with compositional concepts, we present an extension of the IDL specified as part of the Common Object Request Broker Architecture (CORBA) [60] in this section.

With CORBA IDL, one can specify interfaces comprising data attributes (constant or variable) and operations (functions), as well as type definitions and exceptions. The type domain consists of basic data types (e.g., short, float, char, boolean), constructed types (e.g., struct, union, enum), template types (e.g., sequence, string), arrays, and functions. Interfaces and other constructed types use name-based typing; i.e., a function declared to accept an interface type A can only accept objects who have the type *named* by A.

An interface can inherit from another with the inheritance operator ":", in which case all members of the inherited interface become members of the inheriting interface, provided there are no conflicts. This is quite a rudimentary notion of interface inheritance, since it does not permit redefinition of operation (method) types, or breaking larger interfaces into smaller ones, or even resolution of name

Table 7.1. Reuse of design and code for IDL.

Reuse parameter	New	Reused	% reuse
Classes	5	22	81.5
Methods	20	155	88.6
Lines of Code	1300	3600	73.5

conflicts in the case of multiple inheritance. As motivated in Section 7.1, it is beneficial to support a richer notion of interfaces and interface inheritance.

We will extend the CORBA IDL in the following specific directions:

1. We shall support an operator merge that lets one generate interfaces that result from implementation inheritance in CPLs. This requires support for rebinding of operations. The valid types that merge can rebind to a constituent name are those that makes the resultant interface an *inherited type* of the original interface. That is, the resultant interface must share the same recursive structure as the original interface, but need not necessarily be a *subtype*. (Thus, interface inheritance is a weaker notion than subtyping.)

   Integrating the notion of inherited types with name-based typing deserves some consideration. In CORBAs IDL, an interface *names* a type. Consequently, a derived interface, in our extension, names an inherited type (not a subtype) of a base type.

2. Instead of integrating inherited types with name-based typing, we can introduce structural typing of interfaces. Pure name-based typing could become a problem in distributed environments, where data and types could migrate outside the program in which they were originally created [2] and lead to matching of names that may or may not have the same programmer-intended meaning. With structural typing, the names and types of individual constituents of interfaces are significant.

3. We shall support arbitrary rebinding of operations with the override operator. Naturally, override could result in totally unrelated types.

4. Currently, name conflicts in the case of multiple inheritance are illegal. Support for attribute renaming can solve this problem.

5. Just as it is desirable to build up larger interfaces from smaller ones, it is equally desirable to break up larger interfaces into smaller ones. We introduce

an operator project on interfaces to support this.

We can add the above features without introducing the notion of selftype described earlier. Consider the example interfaces shown in Figure 7.2. Interfaces B and C inherit from A, redefining the op1 operation's return value in each case. B and C are subtypes of A, although interfaces that inherit from B need not be subtypes of B (due to the recursion in the input argument to op3). Interface D combines B and C, resolving a conflicting name via the rename operator, since the conflicting name, op1, has unrelated types in B and C. Nonetheless, B and C can be combined without renaming via the override operator as in the expression for E, in which case op1 in the resultant gets its type from the right operand. Interface F inherits from an interface derived from (a subset of) D via the project operator.

7.3 Related Work

The TOOPL language [10] extensively treats the foundations of type systems in relation to inheritance. The notions of selftype and inherited types given here are taken from there.

The type sublanguage of the RAPIDE [44] programming language framework supports much the same functionality that is given in this chapter. It supports

```
interface A {
  A op1 ();
  long op2(in long arg1);
};
interface B = A merge {
  B op1 ();
  boolean op3(in B arg1);
};
interface C = A merge {
  C op1 ();
 long op4(inout long arg1);
};
```

```
interface D =
  (B rename op1 b_op1) merge C;

interface E = B override C;

interface F =
  (D project op1 op4) merge
  { void op5(); };
```

Figure 7.2. Example specifications in extended IDL.

structurally typed interfaces with subtyping, recursion via several forms of parameterized types, interface inheritance, renaming, and projection. In addition, RAPIDE supports a notion of private interfaces which facilitates ADT-style implementation of n-ary methods. However, it does not permit arbitrary overriding as with the override operator. Also, there is no notion of self-reference of type constituents of interfaces.

Concert [4] is a multilanguage distributed programming system in which interface specification is the responsibility of individual programming languages, not a separate IDL. However, a machine-readable IDL is used to define equivalence between declarations in different languages and to support a single intermediate representation.

The use of interface inheritance to address a variety of problems in the evolution of distributed systems is explored nicely in ref. [34], in the context of the Spring distributed system.

7.4 Summary

It is argued in this chapter that complex distributed systems need an explicit notion of interface specification and an expressive language to support such specifications. The expressiveness required of such a language typically includes a notion of interface inheritance.

It is then shown that since interfaces are recursive namespaces, compositional concepts can be used to support reuse of interface specifications. A structure known as *type generators* is introduced to describe notions of interface composition.

An experimental IDL that supports interface composition operators such as merge, override, rename, and project was presented. Also, a way to extend the CORBA IDL with such concepts was discussed.

CHAPTER 8

DOCUMENT COMPOSITION

A large and complex document is often broken down into and composed from smaller pieces. For example, the LaTeX source for this thesis was broken down into files corresponding to each of the chapters, which in turn were composed of files corresponding to sections.

Sometimes, documents developed for one purpose can be reused for other purposes. For example, this thesis can be regarded as a composition of previously written technical reports and papers, with modifications and much added material. Perhaps a more persuasive example is that of report generation — a user manual can be composed from several design document fragments.

A structured document can be modeled as a compositional module. Sections within the document correspond to module attributes, with each section comprising a label, associated section heading, and some textual body. Cross references within text to other section labels correspond to self-references. Thus, the document can be regarded as an abstracted namespace.

The model of compositional modularity can be used to enhance the composability and reusability of documents. This chapter explores applications of compositional document processing (Section 8.1) and presents a system for composing document modules (Section 8.2).

8.1 Applications of Document Composition

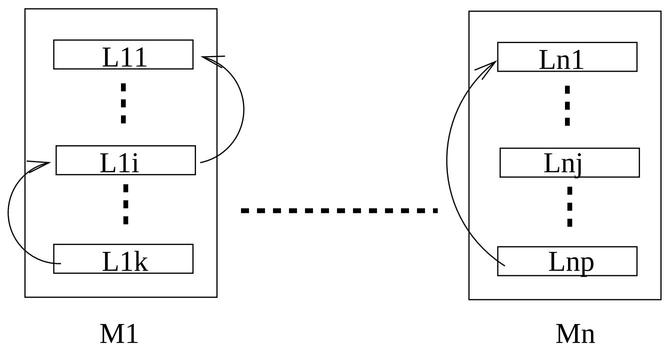
In general, document reuse can be useful in scenarios where several document fragments are generated, edited, composed, revised, maintained, and delivered in various ways. Some envisioned applications are described in the following sections.

8.1.1 Report Generation

There are many environments in which numerous document fragments are generated, with the ultimate goal of putting them together as coherent reports for human consumption. For example, design documents that are generated during application software development form the basis for producing a user manual or a reference manual for the application. Using concepts of compositionality, routine report generation activities can be automated.

Consider the example in Figure 8.1. At the top of the figure is shown a set of document fragments labeled M1 through Mn. Each of these fragments has several sections, where section Lij is the jth section in fragment Mi. A section may contain cross references to other defined or undefined sections within the document fragment.

If each of the fragments can be considered a compositional module, two ways in which they can be usefully put together are described in boxes (a) and (b) in the figure, using a Scheme-like language. (The examples use a function named cl-project which projects sections corresponding to the closure of self-references within those



(a)

```
(merge (cl-project M1 '(L11 L12 ...))
        (merge (cl-project M2 '(...))
           ...
              (cl-project Mn '(...))...))
```

(b)

```
(let (m (mk-module))
  (nest "m1" "M1-hdng" (cl-project M1 '(L11 L12 ...)))
  ...
  (nest "mn" "Mn-hdng" (cl-project Mn '(Ln1 Ln2 ...))))
```

Figure 8.1. Example of report generation.

sections. This function can be written using a module primitive project and an introspective primitive self-refs-in, see Chapter 3.) The expression in Figure 8.1 (a) merges (closures of) particular sections projected from each of the modules, producing a document containing several sections at the same level. The expression in Figure 8.1 (b) creates a new document module and nests within it one subsection per original module that contains (closures of) particular sections projected from each of the original modules.

Document composition such as the above is required routinely within, for example, document centered industrial manufacturing processes, where document fragments are generated at all stages of the process. Imagine that in a car manufacturing plant, individual document fragments are generated at materials receiving, at the parts warehouse inventory, at parts checkout, at various points on the assembly line, at product storage, and at delivery. Using compositionality, these fragments can be automatically composed in various ways to produce a number of extremely useful reports, such as an inventory statement, parts catalog, assembly reports, process monitoring, and quality control documents, etc.

8.1.2 Architectural Specifications

There is a tremendous amount of document generation and consumption in the activity of building construction. In fact, specialized document processing systems for architectural activities have been devised [68].

Typically, a building architect obtains several large textbases of materials specifications, which we shall call MasterSpecs, from a MasterSpec developer. The architect then carefully extracts those parts of the various MasterSpecs that are pertinent to the project in hand. These parts are then integrated with various other project related documents and client specifications, which could themselves already exist in various document fragments. The whole process could entail a significant amount of editing and composition; a document composition system would clearly be very useful here. The integrated document is then delivered to the building contractor.

There are other problems in this scenario that could potentially be solved by a compositional module system. First, MasterSpecs are generally delivered as a linear document, although there is much richer structure and modularity within it, which can be taken advantage of by an architect. Second, maintenance of the numerous documents generated by an architect can be very cumbersome. For example, there are usually multiple copies made of document fragments made for various projects, making their evolution very hard. Instead, with a document composition system which supports document reuse, a single copy of each document module can be maintained and composed with others as necessary.

8.1.3 Revision Control

Revisions are essentially incremental additions and modifications that need to be integrated with a base artifact. Compositional modularity can be useful for the integration of revisions.

In the architectural scenario given in the previous subsection, a MasterSpec developer could periodically release new revisions of MasterSpecs to an architect. The architect is now faced with the problem of integrating the information in the new release into already existing document fragments as well as architectural specifications provided to contractors.

A document composition system could provide effective mechanisms for revision control. For example, reexecuting a composition script on the new revision can extract the updated sections from the new revision. Alternatively, the original document based on the earlier revisions of the MasterSpec can be regarded as an inheritance hierarchy, and the revisions to be made to it (based on the revised MasterSpec) can be specified as a new hierarchy. These two hierarchies can be combined as given in Section 4.3.4. More about the relationship between compositional modularity and revision control is given in Section 9.2.3.

8.2 A System for Document Composition

Having motivated the utility of compositional document processing, this section presents a system for the same. A modular document processing system is a programmable facility that helps a document preparer to adapt and compose documents effectively. In this section, a compositionally modular document processing system called MTEX is described.

As mentioned earlier, the model of compositional modularity can be layered on top of a variety of computational sublanguages. The MTEX system presented in this chapter is built on top of a restricted version of the LaTeX document preparation system [53]. LaTeX is a typesetting program that takes in an ASCII text file annotated with typesetting commands (such as `\section` and `\ref`) and produces a high quality document as a device independent (dvi) file. With LaTeX, authors can concentrate more on the content of their document than on formatting details.

8.2.1 Documents as Compositional Modules

An MTEX module has the syntax specified by the grammar in Figure 8.2. Semantically, an MTEX module is modeled as a generator of an *ordered* set of sections, each of which is a label bound either to a section body or to a nested module. The section label is a symbolic name that can be referenced from other sections. The section body is a tuple (H, B) where H is text corresponding to the section heading and B corresponds to the actual text body, which consists of textual segments interspersed with self-references to labels. An example module and its semantic representation are shown in Figure 8.3. The module has three attributes: `sec:intro` bound to a section body, `sec:model` bound to a nested module, and `sec:concl` bound to a section body.

Given the above model of document modules, consider the meaning of the operations of compositional modularity. The binary operator merge produces a new document module with the sections of its right module operand concatenated to its left module operand, if there are no conflicting labels between the two module operands. Since the order of sections is significant, merge is associative, but not commutative.

The binary operator override concatenates two modules in the presence of conflicting section labels. Conflicting sections in the right operand replace corresponding ones in the left operand. Nonconflicting sections in the right operand are appended to the left operand in the same order that they occur in the right operand.

module	::=	*whitespace section-list*	an MTEX module
section-list	::=	*empty*	
		| *section-list section*	
section	::=	*def-label section-body*	
def-label	::=	`\section{` *heading* `}` *whitespace*	section heading
		`\label{` *label* `}`	symbolic section label
section-body	::=	*empty*	
		| *section-body segment*	list of segments
segment	::=	*whitespace*	
		| *text*	arbitrary text
		| `\ref{` *label* `}`	self-reference

Figure 8.2. MTEX module syntax.

```
\section{Introduction}
\label{sec:intro}

We propose ...

\section{Model}
\label{sec:model}

\subsection{Structure}
\label{sec:struct}

 ... \ref{sec:intro} ...

\subsection{Behavior}
\label{sec:behav}

 ... \ref{sec:struct} ...

\section{Conclusion}
\label{sec:concl}

We have shown that ...
```

$$\begin{array}{lll} \{ & \text{sec:intro} \mapsto & (\text{“Introduction”}, \text{“We propose a model to ...”}), \\ & \text{sec:model} \mapsto & \{ \; \text{sec:struct} \mapsto \; (\text{“Structure”}, \text{“...}\mathtt{\backslash ref}\{\text{sec:intro}\}\text{ ...”}), \\ & & \quad\; \text{sec:behav} \mapsto \; (\text{“Behavior”}, \text{“...}\mathtt{\backslash ref}\{\text{sec:struct}\}\text{ ...”}) \; \}, \\ & \text{sec:concl} \mapsto & (\text{“Conclusion”}, \text{“We have shown that ...”}) \; \} \end{array}$$

Figure 8.3. An example MTEX document module (top) and its semantic representation as a compositional module (bottom).

The restrict operator has the usual meaning of removing sections. However, its dual operator project is potentially more useful in the context of document composition. The operators rename and copy-as have the usual meaning.

In the MTEX system, we have chosen not to support encapsulation, i.e., the hide operator, and its related concept of static binding, i.e., the freeze operator. However, encapsulation could conceivably have some fairly natural meanings for some applications of document processing. For example, hidden sections could be appended to a special appendix that is suitably titled. Alternatively, a new document consisting of hidden sections could be created, and citations to the new document could be inserted in place of references to the hidden sections.

Hierarchical nesting is a very important and useful notion in document structuring. The nest operator supports retroactive nesting of document modules, as described in Chapter 4. However, two issues arise. First, since the LATEX system does not support nested namespaces, labels within nested modules must not conflict with any other labels in the module. This can be checked by the type rule of nest if we model the interface of a document as comprising a *flat* set of all the labels within the module. Second, self-references and environment references are indistinguishable in MTEX modules. Thus, the nest operator considers all unresolved self-references in its nested module argument to stand for environment references. Also, due to the nature of the generator model as it stands, self-references within nested modules can refer only to labels in surrounding scopes, but not to labels within other nested modules.

Naturally, there is no notion of typing in MTEX other than the notion of module interfaces consisting of label names mentioned above.

The notion of instantiation of documents is interesting. For one thing, instantiation involves binding of self-references. Additionally, instantiation results in allocation of space and the layout of instances. Considering these, instantiation of MTEX document modules is conceptually equivalent to running the latex program on a document to produce a dvi file. In fact, latex would normally need to be run multiple times on a file in order to resolve references — this iteration directly corresponds to taking the fixpoint of the document module.

8.2.2 MTEX Architecture

The architecture of the MTEX document processing system is two-tiered, similar to the architecture presented in Chapter 6.

Physical modules are created with the syntax described in the previous section. Section bodies can have arbitrary text in them, including arbitrary LaTeX command source.

A Scheme-derived module language is then used to transform the physical files into first-class compositional modules, then compose them in various ways. For example, a mk-module primitive is used to read in a physical document module. Operators such as project, nest and override are then used to adapt and combine these first-class modules, which can be written back as files using write-module. The syntax of some sample primitives is shown in Figure 8.4.

Meta-level introspective primitives, such as self-refs-in (see Section 3.1.6), are also useful while manipulating documents. For example, project and self-refs-in can be used to write a Scheme function that projects the closure of self-references from a set of sections. In fact, such a function, called cl-project, is included in the standard Scheme library associated with MTEX. As in Chapter 6, MTEX script files can contain arbitrary Scheme code.

Since section order is crucial in documents, the user might wish to reorder sections using a primitive reorder.

(mk-module [⟨*mtex-file-name*⟩])
(write-module ⟨*module-expr*⟩ ⟨*file-name*⟩)
(merge ⟨*module-expr1*⟩ ⟨*module-expr2*⟩)
(override ⟨*module-expr1*⟩ ⟨*module-expr2*⟩)
(project ⟨*module-expr*⟩ ⟨*label-list-expr*⟩)
(nest ⟨*outer-module-expr*⟩ ⟨*label*⟩ ⟨*heading*⟩ ⟨*nested-module-expr*⟩)
(self-refs-in ⟨*module-expr*⟩ ⟨*label-list-expr*⟩)
(reorder ⟨*module-expr*⟩ ⟨*label-list-expr*⟩)

Figure 8.4. Sample MTEX primitives.

Below, a simple example of MTEX document composition is shown. Say the document shown in Figure 8.3 is stored in a file named body.tex. Imagine that another file oo.tex contains sections corresponding to the various concepts in OO programming. The following expression can be used to compose the two files:

```
(override (mk-module "body.tex")
          (nest (mk-module) "sec:model" "OO Concepts" (mk-module "oo.tex")))
```

The above expression nests the module oo.tex with the section label sec:model into a freshly created empty module, which overrides the other module. The resultant module has a section labeled sec:model bound to a nested module, consisting of sections describing OO concepts. This module can be written out as a LATEX file and instantiated into a dvi file. Other examples, such as the ones shown in Figure 8.1, can be easily encoded in MTEX.

8.2.3 Implementation

The implementation architecture of MTEX is shown in Figure 8.5. A yacc/lex parser reads in document modules, creates the appropriate framework classes, and returns a first-class MTEX module, an object of class TexModule (below). These modules can be manipulated using Scheme primitives via the interpreter.

The subclasses of ETYMA created to construct MTEX are TexLabel of Label, Section of Method, TexModule of StdModule, SecMap of AttrMap, and TexInterface of StdInterface. Also, a new class Segment was created. Approximate design and code reuse numbers for the MTEX implementation are shown in Table 8.1.

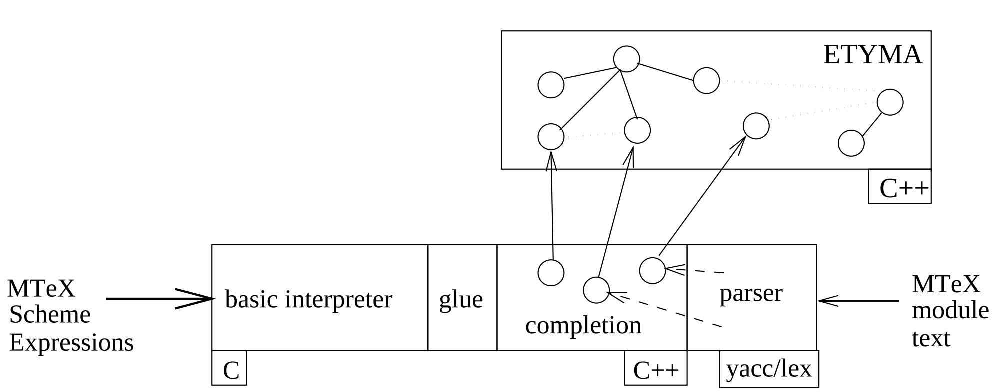


Figure 8.5. Architecture of the MTEX system.

Table 8.1. Reuse of design and code for MTEX.

Reuse parameter		New	Reused	% reuse
ETYMA	Classes	6	20	77
	Methods	36	231	86.5
	Lines of Code	1600	4400	73.3
ETYMA + STk	Lines of Code	1800	19400	91.5

8.3 Summary

This chapter characterizes documents as abstracted namespaces, thus making it possible to apply notions of compositional modularity. Many module operators introduced previously are shown to be sensible and beneficial to document reuse.

Inheritance is shown to reap benefits in the context of documents. For example, document fragments generated for one purpose can be reused for other similar, related, or even unrelated purposes. Evolution and maintenance are made easier.

A document processing system called MTEX, which supports composition of modules written in a variant of LATEX, was introduced. Examples in MTEX and applications such as report generation, architectural specification, and revision control were discussed.

This chapter has shown that the model of compositional modularity is beneficial to nontraditional computer applications such as document processing.

CHAPTER 9

CONCLUSIONS

Having demonstrated the expressive power and broad applicability of compositional modularity in detail, we will, in this chapter, recapitulate the significant lessons learned from this research and point to some unexplored directions.

9.1 Summary of Contributions

This dissertation has shown that the model of compositional modularity facilitates a high degree of reuse by supporting effective recomposition mechanisms for highly decomposed software. In particular, it is shown that compositional modularity supports a stronger and more flexible notion of reuse than traditional class-based inheritance in object-oriented programming.

Compositional modularity defines a comprehensive suite of operations to manipulate self-referential namespaces. Owing to the pervasiveness of namespace manipulation in software systems, compositional modularity can be applied within an unusually wide range of systems. To demonstrate, four such systems are described and it is shown that they derive significant benefits from incorporating compositional modularity. To facilitate the efficient construction of tools for such systems, a reusable software architecture has been developed and successfully reused.

Specifically, this dissertation has made the following advances over previous research:

1. A programming style based on compositional modularity has been illustrated via examples. The model has been shown to be unifying in scope by emulating the important idioms of object-oriented programming, including several

forms of single and multiple inheritance, wrapping, and inheritance hierarchy combination.

2. Hierarchical nesting has been characterized as a composition operation and shown to be an expressive feature by demonstrating its many applications.

3. A realistic imperative language, called *CMS*, embodying the major aspects of the model has been designed and implemented. *CMS* is an augmentation of the programming language Scheme.

4. Compiled object files have been characterized as compositional modules. An architecture for managing such modules and for building entire applications by composing them has been described. The primary component in this architecture is a programmable linker. Some longstanding problems with component management are shown to be alleviated within this architecture.

5. Interfaces, by virtue of being recursive structures, are shown to benefit from compositional concepts. An experimental compositional interface definition language is presented. Ways to augment existing IDLs are suggested.

6. It has been shown that document fragments can be viewed as compositional modules and, thus, may be reused in applicable situations. A modular document processing system layered on the LaTeX document preparation system and its applications are described.

7. A generic, reusable, object-oriented framework, known as Etyma, for constructing tools for systems based on compositional modularity has been designed and implemented. The current version of Etyma, comprising more than 45 classes in 7000 lines of C++ that evolved over six iterations, has been documented using design patterns. Etyma has been reused to realize the four significantly differing tools above as completions; three of them were direct completions and the fourth was a parallel one. Reuse of both design and code for all measured completions was found to be significant (between

73.3% and 91.7%).

There are some limitations to the model of compositional modularity. The model as presented here is not entirely unifying of previous models of modularity. For example, the model would benefit from supporting a slightly weaker notion of encapsulation such as that supported by ADT's. Implementation of binary (and more generally, n-ary) methods would be facilitated by this. In particular, a framework for compositionality can then be specified using a strongly typed compositionally modular language. This is not possible currently due to binary operations such as merge and override. Additionally, it must be noted that the binary module operators merge and override cannot be specified polymorphically, since they require complete knowledge of their incoming parameters. (The problem of integrating the notion of ADT's with inheritance in the presence of static typing was noted in ref. [7].)

There are several interesting and useful directions in connection with compositional modularity that are as yet unexplored. These are sketched in the following section.

9.2 Future Directions

9.2.1 Framework Enhancements

The design of ETYMA can be enhanced to encompass more functionality than presented here. The following specific areas come to mind.

The type system supported by ETYMA can be improved. First, typechecking method implementations when input or output parameters are specified to be selftype should be supported. However, associating a type with selftype can be tricky due to the hide operator. Hiding shrinks rather than expands an interface via inheritance. Thus, a bound on interfaces, which essentially specifies attributes that cannot be hidden after the fact, will be required to support this feature. Second, support for overloading, or *ad-hoc* polymorphism, may be found to be necessary in attempting to apply compositional modularity to some systems. This can, however, be supported by augmenting the notion of attribute matching across modules by having the name as well as the type of attributes significant.

As mentioned earlier, ETYMA does not support front-end and back-end related abstractions for tool construction. For supporting front-ends, a modular parser can be associated with each abstraction that parses by calling parsers associated other abstractions in turn. Object-oriented parser technology such as yacc++ [21] can be utilized for this.

Back-end compilation is more interesting. Say we want to support translation into the C language. As usual, the front-end can first build up an internal representation of the program comprising instances of specialized ETYMA classes. Class Method can support a method translate that translates its body into C code. This method can be implemented in terms of calls to translate methods of other abstractions, many of which would be specified by completion classes. Furthermore, for each occurrence of module instantiation in the source, class Method's translate can call class *Module*'s method gen_instance() which would generate C code that allocates an instance of that module with the appropriate layout (an object layout mechanism such as that suggested in ref. [7] can be used). Compilation in the presence of first-class modules will be slightly more involved, since code to represent modules and perform module operations at run-time must be generated as well.

9.2.2 Other Completions

Processors for many other programming languages can be built by reusing ETYMA. These programming languages can either be non-OO languages (similar to Scheme) or already OO. Of course, if they are already OO, it will be much harder to reconcile the semantics and pragmatics of the preexisting OO model with that of compositional modularity. (A brief treatment of extending Modula-3 to incorporate compositional operators is given in ref. [7].) However, in my opinion, much more can be gained by layering compositional modularity on top of *non-OO* languages and systems, such as the four examples presented in this work.

In general, modularity can be layered in three ways: (i) no first-class run-time modules or instances, in which case namespaces are manipulated prior to run-time only; (ii) first-class instances but no first-class modules, similar to most conventional

OO languages; and (iii) first-class modules and instances, similar to *CMS*. For instance, the programming language C can be extended in any of the three ways above.

It is interesting to consider other potential applications of compositional modularity that do not have to do with conventional programming languages. For instance, hypertext documents on the world wide web (WWW) correspond to namespaces with internal URL self-references as well as external (nonlocal) URL references. Such documents can be statically composed as shown in Chapter 8. Individual document fragments can be considered nested within the universal namespace of URLs such that nonlocal references are simply interpreted by the hypertext tool to find remotely located documents.

Another application is the management of visual entities (such as windows, buttons, menus) in a graphical user-interface environment. In this case, a container (such as a canvas) can be considered to be a namespace with various individual elements such as buttons, panel displays, and other canvases as attributes, each associated with a name as well as its two-dimensional position. Furthermore, some attributes (such as panel displays) can refer to other attributes (such as slide-bars). Given this notion of namespaces, adaptation operations such as renaming, copying, and removing, as well as composition operations such as merging, overriding, and nesting, can be performed on them, e.g., via some kind of a shell language. (This idea actually corresponds to many existing characterizations of GUI entities as objects in OO programming.)

A third application could be file system directories viewed as self-referential namespaces. A directory consists of file names bound to file contents that can potentially refer back to file names, e.g., a C source code directory with `#include` directives. It might be desirable to construct a specialized compositional file system to manage such source code repositories.

9.2.3 Version Management

In general, version management and compositional modularity have an inverse relationship. In compositional modularity, given a module *Old*, an increment *Delta*, and a specification $\oplus$ of how to combine them, we generate a new module New.

$$Old \;\oplus\; Delta = New$$

On the other hand, in version management, given a new module New and an old module Old, we generate the difference $Delta$ and a specification (the inverse of $\ominus$ below, say $\oplus$) of how to obtain New from Old.

$$New \;\ominus\; Old = Delta$$

A version management system then stores Old, $Delta$, as well as $\oplus$, so that either the old version or the new version can be generated. In effect, the revision history of a module comprises incremental modifications ($Delta$'s) as well as the composition history ($\oplus$'s).

This raises the interesting question of whether a relation such as $\ominus$ (and its inverse $\oplus$) between compositional modules can be automatically computed. Traditional tools such as diff compute $Delta$ at the granularity of text lines, but it would be interesting to study mechanisms to compute $Delta$ at the granularity of module attributes.

In any event, the above analysis suggests one, albeit not completely satisfactory, way to perform version management using compositional modularity, as sketched in Section 8.1.3. New revisions of modules are not specified by giving module New, but rather by giving $Delta$ and $\oplus$, which are stored in the newly generated module New. In this manner, every module carries along its own composition history, so that some form of meta-level primitives can extract older versions of the module.

9.2.4 Distributed Programming

In the treatment so far, composed modules are tightly bound to each other, in a centralized manner. Thus, in the case of program modules, a resultant composed module is expected to inhabit a single program address space.

Alternatively, program modules can be composed in a loosely coupled manner, so that they can reside geographically separated but still be logically composed.

This can be done by composing program modules so that remote procedure calls (RPC's) are made between parts of a composed module instead of regular procedure calls. (This involves generating and integrating RPC adapters during composition.)

In an architecture as given in Chapter 6, there is no notion of first-class instances or message-sending in the base language (compiled C object files). However, one can still support a limited form of message-sending. The basic idea is that a module instance corresponds to a thread in an address space. (Thus one can have many module instances within the same address space.) With this, message sending between instances is modeled as interprocess communication (IPC) by converting static calls to IPC calls. For example, the expression

```
(msg-send m1 foo m2 bar)
```

wraps the static call to foo() within m1 with an IPC stub that calls the bar() routine within an instance of m2, which is itself wrapped with a receiving IPC stub. The crucial question here is that of determining the identity of the receiving instance of m2, since there is no notion of first-class instances. One answer to this question is to have the msg-send primitive also generate a constructor function that establishes the IPC environment between m1 and m2. For example, the constructor routine for m2 registers instances of m2 with a name service, and invocations of m1's foo() look up the identity of an m2 instance and establishe an IPC handle using that name. The particular instance of m2 that the name service returns can either be constant for the duration of the program or be programmatically controlled from within base language modules.

Finally, it should be mentioned that a compositional IDL (such as that presented in Chapter 7) and a distributed programming infrastructure (as presented in Chapter 6) can be fruitfully integrated with each other, since they both support a similar view of compositionality. For instance, the IDL can be used to specify and compose the interfaces of components and generate adapters for their interactions, whereas the actual components can be implemented using conventional languages and composed using a compositional module language.

REFERENCES

[1] Allen, R., and Garlan, D. Beyond definition/use: Architectural interconnection. In *Proc. of Workshop on Interface Definition Languages* (January 1994), J. Wing, Ed., pp. 35 – 45. Available as August 1994 issue of ACM SIGPLAN Notices.

[2] Amadio, R. M., and Cardelli, L. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems 15*, 4 (September 1993).

[3] Apollo Computer, Inc. *DOMAIN Software Engineering Environment (DSEE) Call Reference.* Chelmsford, MA, 1987.

[4] Auerbach, J., and Russel, J. The Concert signature representation: IDL as intermediate language. In *Proc. of Workshop on Interface Definition Languages* (January 1994), J. Wing, Ed., pp. 1 – 12. Available as August 1994 issue of ACM SIGPLAN Notices.

[5] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. Lightweight remote procedure call. *Association for Computing Machinery Transactions on Computer Systems 8*, 1 (February 1990), 37–55.

[6] Birrell, A., and Nelson, B. Implementing remote procedure calls. *Association for Computing Machinery Transactions on Computer Systems 2*, 1 (February 1984), 39–59.

[7] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, March 1992. Technical report UUCS-92-007; 143 pp.

[8] Bracha, G., and Cook, W. Mixin-based inheritance. In *Proc. OOPSLA Conference* (Ottawa, October 1990), ACM.

[9] Bracha, G., and Lindstrom, G. Modularity meets inheritance. In *Proc. International Conference on Computer Languages* (San Francisco, CA, April 20–23, 1992), IEEE Computer Society, pp. 282–290. Also available as Technical Report UUCS-91-017.

[10] Bruce, K. B. A paradigmatic object-oriented programming language: Design static typing and semantics. Tech. Rep. CS-92-01, Williams College, January 31, 1992.

[11] Burstall, R., and Lampson, B. A kernel language for abstract data types and modules. In *Proceedings, International Symposium on the Semantics of*

Data Types, G. Kahn, D. MacQueen, and G. Plotkin, Eds., vol. 173 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1984, pp. 1–50.

[12] Callahan, J. R., and Purtilo, J. M. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering 17*, 6 (June 1991), 626–635.

[13] Campbell, R. H., Islam, N., Johnson, R., Kougiouris, P., and Madany, P. *Choices*, frameworks and refinement. In *Object Orientation in Operating Systems* (Palo Alto, CA, October 1991), IEEE Computer Society, pp. 9–15.

[14] Canning, P., Cook, W., Hill, W., and Olthoff, W. Interfaces for strongly-typed object-oriented programming. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1989), N. Meyrowitz, Ed., pp. 457–467.

[15] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report. Tech. Rep. 31, Digital Equipment Corporation Systems Research Center, Aug. 1988.

[16] Cardelli, L., and Mitchell, J. C. Operations on records. Tech. Rep. 48, Digital Equipment Corporation Systems Research Center, Aug. 1989.

[17] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17*, 4 (December 1985), 471–522.

[18] Chamberlain, S. libbfd. Free Software Foundation, Inc. Contributed by Cygnus Support, March 1992.

[19] Chambers, C., Ungar, D., Chang, B.-W., and Holzle, U. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *LISP and Symbolic Computation: An International Journal 4*, 3 (1991).

[20] Chiba, S., and Masuda, T. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming* (New York, 1993), Springer-Verlag. LNCS 707.

[21] Clark, C. OO compilation — what are the objects? In *Addendum to the proceedings of OOPSLA* (Portland, Oregon, October 1994), M. C. Wilkes, Ed., OOPS Messenger, ACM SIGPLAN, pp. 67 – 71.

[22] Clinger, W., and Rees, J. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers 4*, 3 (1991).

[23] Cook, W., Hill, W., and Canning, P. Inheritance is not subtyping. In *Theoretical Aspects of Object-Oriented Programming*, C. Gunter and J. Mitchell, Eds. MIT Press, 1994, pp. 497 – 517.

[24] Cook, W., and Palsberg, J. A denotational semantics of inheritance and its correctness. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications* (1989), pp. 433–444.

[25] Curtis, P., and Rauen, J. A module system for Scheme. In *Conference Record of the ACM Lisp and Functional Programming* (1990), ACM.

[26] Deutsch, L. P. Design reuse and frameworks in the Smalltalk-80 programming system. In *Software Reusability*, T. J. Biggerstaff and A. J. Perlis, Eds., vol. 2. ACM Press, 1989, pp. 55–71.

[27] Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. Proposal for a monotonic multiple inheritance linearization. In *Proceedings of OOPSLA* (October 1994), pp. 164 – 175.

[28] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[29] Gallesio, E. STk reference manual. Version 2.1, 1993/94.

[30] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[31] Gingell, R. A. Shared libraries. *Unix Review 7*, 8 (August 1989), 56–66.

[32] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[33] Gordon, M. J. C. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.

[34] Hamilton, G., and Radia, S. Using interface inheritance to address problems in system software evolution. In *Proc. of Workshop on Interface Definition Languages* (January 1994), J. Wing, Ed., pp. 119 – 128. Available as August 1994 issue of ACM SIGPLAN Notices.

[35] Harper, R., Mitchell, J. C., and Moggi, E. Higher-order modules and the phase distinction. In *Conference record of the 17th ACM Symposium on Principles of Programming Languages (POPL)* (San Francisco, CA USA, 1990), pp. 341–354.

[36] Harper, R., and Pierce, B. A record calculus based on symmetric concatenation. In *Proc. of the ACM Symp. on Principles of Programming Languages* (Jan. 1991), pp. 131–142.

[37] Harrison, W., and Ossher, H. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA Conference* (September 1993), ACM Press, pp. 411 – 428.

[38] Ho, W., and Olsson, R. An approach to genuine dynamic linking. *Software— Practice and Experience 21*, 4 (April 1991), 375–390.

[39] Intermetrics, Inc. *Ada 9X Reference Manual.* Cambridge, Massachusetts, 1994. Ada 9X Mapping/Revision Team Effort.

[40] Jagannathan, S. Dynamic modules in higher-order languages. In *Proceedings of the International Conference on Computer Languages* (1994), IEEE.

[41] Jagannathan, S. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 456–492.

[42] Johnson, R. E., and Russo, V. F. Reusing object-oriented designs. Tech. Rep. UIUCDCS 91-1696, University of Illinois at Urbana-Champagne, May 1991.

[43] Joy, W., Graham, S., Haley, C., McKusick, M. K., and Kessler, P. Berkeley Pascal user's manual. In *UNIX Programmer's Manual*, vol. 1 of *4.3 BSD.* USENIX Association, CSRG, University of California, Berkeley, CA, April 1986.

[44] Katiyar, D., Luckham, D., and Mitchell, J. A type system for prototyping languages. In *Proc. of the ACM Symp. on Principles of Programming Languages* (Portland, OR, January 1994), ACM, pp. 138–150.

[45] Keene, S. E. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, Reading, MA, 1989.

[46] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[47] Kiczales, G., des Rivières, J., and Bobrow, D. G. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, 1991.

[48] Kiczales, G., and Lamping, J. Issues in the design and specification of class libraries. In *OOPSLA Proceedings* (1992), ACM.

[49] Kiczales, G., Lamping, J., and Mendhekar, A. What a metaobject protocol based compiler can do for Lisp. Unpublished report. A modified version presented at the OOPSLA '94 workshop on O-O Compilation [21], 1994.

[50] Kiczales, G., and Rodriguez, L. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (1990), ACM, pp. 99–105.

[51] Kristensen, B. B., Madsen, O. L., Moller-Pedersen, B., and Nygaard, K. The BETA programming language. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 7 – 48.

[52] Lamping, J. Typing the specialization interface. In *Proceedings of OOPSLA '93, ACM SIGPLAN Notices* (October 1993), pp. 201–214.

[53] Lamport, L. *LaTeX, a Document Processing System.* Addison Wesley Publishing Company, Reading, MA, 1986.

[54] Luckham, D., and Vera, J. Event-based concepts and language for system architecture. Available from Stanford Program Analysis and Verification Group, March 1993.

[55] Madsen, O. L. Block structure and object-oriented languages. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 113 – 128.

[56] Menapace, J., Kingdon, J., and MacKenzie, D. The "stabs" debug format. Free Software Foundation, Inc. Contributed by Cygnus Support, 1993.

[57] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1987.

[58] Meyer, B. Eiffel, the environment, August 1989.

[59] Nelson, E. G. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[60] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, December 1991. Revision 1.1.

[61] Orr, D., Bonn, J., Lepreau, J., and Mecklenburg, R. Fast and flexible shared libraries. In *Proc. USENIX Summer Conference* (Cincinnati, June 1993), pp. 237–251.

[62] Orr, D. B. Application of meta-protocols to improve OS services. In *HOTOS-V: Fifth Workshop on Hot Topics in Operating Systems* (May 1995).

[63] Orr, D. B., and Mecklenburg, R. W. OMOS — An object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems* (Paris, September 1992), IEEE Computer Society, pp. 200–209. Also available as technical report UUCS-92-033.

[64] Orr, D. B., Mecklenburg, R. W., Hoogenboom, P. J., and Lepreau, J. Dynamic program monitoring and transformation using the OMOS object server. In *The Interaction of Compilation Technology and Computer Architecture.* Kluwer Academic Publishers, February 1994.

[65] Ossher, H., and Harrison, W. Combination of inheritance hierarchies. In *OOPSLA Proceedings* (October 1992), pp. 25–40.

[66] Purtilo, J. M. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems 16*, 1 (January 1994), 151–174.

[67] Rees, J. Another module system for Scheme. Included in the Scheme 48

distribution, 1993.

[68] Rossberg, W., Smith, E., and Matinkhah, A. Structured text system. US Patent Number 5,341,469, August 1994.

[69] Sabatella, M. Issues in shared libraries design. In *Proc. of the Summer 1990 USENIX Conference* (Anaheim, CA, June 1990), pp. 11–24.

[70] Sankar, S., and Hayes, R. ADL — an interface definition language for specifying and testing software. In *Proc. of Workshop on Interface Definition Languages* (January 1994), J. Wing, Ed., pp. 13 – 21. Available as August 1994 issue of ACM SIGPLAN Notices.

[71] Seeley, D. Shared libraries as objects. In *Proc. USENIX Summer Conference* (Anaheim, CA, June 1990).

[72] Sheldon, M. A. Static dependent types for first class modules. In *ACM Conference on Lisp and Functional Programming* (June 1990).

[73] Steele Jr., G. L. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1984.

[74] Stroustrup, B. Type-safe linkage for C++. In *USENIX C++ Conference* (1988).

[75] Thatte, S. R. Automated synthesis of interface adapters for reusable classes. In *Symposium on Principles of Programming Languages* (January, 1994).

[76] Tofte, M. *Four Lectures on Standard ML.* LFCS Report Series, Department of Computer Science, University of Edinburgh, 1989.

[77] Tung, S.-H. S. Interactive modular programming in Scheme. In *Proceedings of the ACM Lisp and Functional Programming Conference* (1992), ACM, pp. pages 86 – 95.

[78] Vlissides, J. M., and Linton, M. A. Unidraw: a framework for building domain-specific graphical editors. In *Proceedings of the ACM User Interface Software and Technologies '89 Conference* (November 1989), pp. 81–94.

[79] Weinand, A., Gamma, E., and Marty, R. ET++: an object-oriented application framework in C++. In *Proceedings of OOPSLA '88* (November 1988), ACM, pp. 46–57.

[80] Winnicka, B., Bodin, F., and Gannon, D. C++ objects for representing and manipulating program trees in the Sage++ system. Presented at the *O-O Compilation* Workshop at OOPSLA, October 1994. See [21].