THE PROGRAMMING LANGUAGE JIGSAW: MIXINS, MODULARITY AND MULTIPLE INHERITANCE

by

Gilad Bracha

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

March 1992

Copyright © Gilad Bracha 1992

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Gilad Bracha

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
	John Van Rosendale
	Joseph L. 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 Gilad Bracha 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 C. Henderson
Chair/Dean

Approved for the Graduate Council

B. Gale Dick
Dean of The Graduate School

ABSTRACT

This dissertation provides a framework for modularity in programming languages. In this framework, known as *Jigsaw*, inheritance is understood to be an essential linguistic mechanism for module manipulation.

In *Jigsaw*, the roles of classes in existing languages are "unbundled," by providing a suite of operators independently controlling such effects as combination, modification, encapsulation, name resolution, and sharing, all on the single notion of *module.*

All module operators are forms of inheritance. Thus, inheritance is not in conflict with modularity in this system, but is indeed its foundation.

This allows a previously unobtainable spectrum of features to be combined in a cohesive manner, including multiple inheritance, mixins, encapsulation and strong typing.

Jigsaw has a rigorous semantics, based upon a denotational model of inheritance.

Jigsaw provides a notion of modularity independent of a particular computational paradigm. *Jigsaw* can therefore be applied to a wide variety of languages, especially special-purpose languages where the effort of designing specific mechanisms for modularity is difficult to justify, but which could still benefit from such mechanisms.

The framework is used to derive an extension of *Modula-3* that supports the new operations. An efficient implementation strategy is developed for this extension. The performance of this scheme is on a par with the methods employed by the highest performance object-oriented language processors currently available.

In Memory of my Father, Peretz Bracha, 1933-1989.

CONTENTS

ABSTRACT iv

LIST OF FIGURES x

ACKNOWLEDGMENTS xii

CHAPTERS

1. INTRODUCTION **1**

1.1 Understanding the Problem 2
1.1.1 Inheritance 2
1.1.2 Abstract Classes and Frameworks 4
1.1.3 Module Manipulation 5
1.2 Mixins 5
1.3 Jigsaw 7
1.4 Semantics 8
1.5 Modula-π 9
1.6 Implementation 9
1.7 Conclusions 10

2. THE PROBLEM **12**

2.1 Modules and Modularity 13
2.1.1 What is a Module? 13
2.1.2 Desiderata for Modules 14
2.1.3 The Trend Toward Modularity 15
2.2 Modularity Problems in Existing Languages 16
2.2.1 Flat, Global Name Spaces 16
2.2.2 Lack of Inheritance 17
2.3 Difficulties with Inheritance 20
2.3.1 Classes and Types 21
2.3.1.1 Multiple Implementations of an Abstraction . . . 21
2.3.1.2 Subtyping and Inheritance 21
2.3.1.3 Other Considerations 23
2.3.2 The Diamond Problem 24
2.3.3 Accessing Indirect Ancestors 26
2.3.4 Visibility control 27
2.3.5 Limits on Module Construction 28
2.4 Problems by Language 30

3. MIXINS **33**

3.1 Mixins in Existing Languages 33
3.1.1 CLOS 34
3.1.2 SELF 35
3.1.3 Beta 35
3.2 Mixins as Abstractions 38
3.2.1 Mixins and Type Abstraction 40
3.2.2 A Dedicated Construct for Mixins 42
3.2.3 Mixin-based inheritance 43
3.2.4 Mixin Composition 45
3.3 Elevating Classes to Mixins 46
3.3.1 Extending Existing Languages 47
3.4 Limitations 47

4. JIGSAW **49**

4.1 Roles of a Class 50
4.2 The Jigsaw Operator Suite 52
4.2.1 Module Definition 52
4.2.2 Instantiation 52
4.2.3 Combining Modules 54
4.2.4 Modification 54
4.2.5 Name Conflict Resolution 55
4.2.6 Sharing 56
4.2.7 Restricting Modifications 57
4.2.8 Attribute Visibility 57
4.2.9 Access to Overridden Definitions 58
4.3 Nesting Modules 59
4.4 An Interactive Jigsaw Interpreter 60
4.5 Adding Modules to Existing Languages 62
4.5.1 Jigsaw as a Framework 64

5. SEMANTICS **71**

5.1 Background 73
5.1.1 Generators 73
5.1.2 Records 74
5.1.3 Inheritance 75
5.2 Modeling Mixins 77
5.2.1 A Mixin Composition Operator 77
5.2.2 Mixin Composition as Function Composition 78
5.3 Modeling Jigsaw 79
5.4 Formal Definition of Jigsaw 85
5.4.1 Syntax 85
5.4.2 Type Rules 86
5.4.2.1 Judgements 87

5.4.2.2 Key Rules . . . 88
5.4.3 Translation to λ calculus . . . 95
5.5 An Imperative Jigsaw . . . 96
5.5.1 Denotational Semantics of Imperative *Jigsaw* . . . 97
5.5.2 Syntactic Domains . . . 97
5.5.3 Semantic Domains . . . 97
5.5.4 Semantic Functions . . . 97

6. MODULA-π . . . 100

6.1 Choice of Language . . . 101
6.2 A Review of Modula-3 . . . 101
6.2.1 Modula-3 Inheritance . . . 101
6.2.2 Other Salient Features . . . 103
6.2.3 Typing . . . 104
6.3 Modula-π: An Extension of Modula-3 . . . 105
6.3.1 Object Types and their Operators . . . 105
6.3.2 Type Abstraction . . . 106
6.3.3 Subtyping . . . 107
6.3.4 Compatibility . . . 108
6.4 Assessing Modula-π . . . 109

7. IMPLEMENTATION . . . 111

7.1 Implementation of Modula-π . . . 112
7.1.1 Implementing Single Inheritance . . . 113
7.1.2 Implementing Multiple Inheritance . . . 114
7.1.3 Basic Implementation of Operator-based Inheritance . . . 116
7.1.3.1 Problems with standard techniques . . . 116
7.1.3.2 Implementing Primitive Object Types . . . 118
7.1.3.3 Implementing Object Type Composition . . . 120
7.1.4 Pure Virtuals . . . 122
7.1.5 Other Operations . . . 124
7.1.6 Jigsaw Operations Not in Modula-π . . . 124
7.1.7 Additional Details . . . 125

8. FINALE . . . 127

8.1 Related Work . . . 127
8.1.1 Jade . . . 127
8.1.2 CommonObjects . . . 128
8.1.3 Mixins . . . 129
8.1.4 Generator Operations . . . 129
8.1.5 Mitchell . . . 130
8.2 Future Work . . . 131
8.2.1 Name-based Typing . . . 131
8.2.2 Abstract Data Types . . . 132

8.2.3 Formal Specification of Inheritable Modules 132
8.2.4 Prototypes 133
8.2.5 Nested Modules Revisited 135
8.2.6 Process Calculi 136
8.3 Conclusion 136

REFERENCES 138

LIST OF FIGURES

1.1 A simple example of inheritance 2

1.2 Inheritance hierarchy 3

2.1 Use of **import** declarations 17

2.2 Code for Point and Manhattan_Point. 18

2.3 Inheritance as module manipulation. 18

2.4 The "diamond" problem 25

2.5 "Opening" the diamond. 26

2.6 Accessing an overridden method. 27

2.7 Accessing indirect ancestors violates encapsulation. 27

2.8 Lack of mixins causes repetitive code. 30

3.1 *Beta* prefixing 36

3.2 Mixins in *Beta* 37

3.3 Generics as mixins 39

3.4 Mixin application 41

3.5 The common form of mixins. 42

3.6 The signature of a mixin. 43

3.7 A dedicated mixin construct. 43

3.8 A simple multiple inheritance hierarchy 44

3.9 Linearized hierarchies 45

4.1 A module and its interface 53

4.2 **Rename** distributes over **override** 56

4.3 Using a mixin 58
4.4 A framework implementing *Jigsaw* 65
4.4 - Continued 66
4.5 *Jigsaw* specifies implementation frameworks 67
4.6 Specification frameworks are like implementation frameworks 68
4.7 The big picture 69
5.1 An object and its generator 73
5.2 A manhattan point inherits from a point 76
5.3 A valid definition of renaming. 82
5.4 An alternative definition of renaming. 83
5.5 Judgements of *Jigsaw*. 88
6.1 *Modula-3* object types 102
6.2 *Modula-3* subtyping rules for object types. 104
6.3 Difficulty with **merge** and abstract data types. 106
6.4 *Modula-π* object type subtyping 108
7.1 The *Offset* function. 115
7.2 Associativity 116
7.3 Constructors of object types 118
7.4 A primitive object type 119
7.5 Layout of primitive and composite object yypes 119
7.6 Offset value in the mtbl. 121
7.7 Several composite object types 121
7.8 Examples of pure virtual methods. 122
7.9 Layout of classes with pure virtuals 123

ACKNOWLEDGMENTS

They now turned and took a last look at the Emerald City.
The Wonderful Wizard of Oz.

It may be argued that there is nothing more rewarding than writing the acknowledgements section of a doctoral dissertation, especially one long overdue. Indeed, it was with some difficulty that I resisted the temptation to write this section before writing the dissertation itself.

First and foremost, I thank my advisor, Gary Lindstrom. This thesis would not have come about without his patience and support. Gary believed in this work at times when almost no one else did, and in spite of all evidence to the contrary. I would also like to thank the other members of my committee, John Van Rosendale and Joe Zachary, for friendship, encouragement and advise, above and beyond the call of duty.

Special thanks go to my family and friends in Israel: To my mother, Shoshana, and my late father, Peretz, whose trust gave me the confidence necessary to undertake this project. To my sister, Rachel and my aunt Lily, for being who they are. To Aaron and Sonia Rogozinski, for their unconditional support. And to my friends Yaron Kashai, Eyal Shim'oni and Liav Weinbaum, for shared insanity.

I owe a debt to William Cook, both for helping establish a theory of inheritance that I could use in my investigations, and for his frank comments on my technical writing. I hope I have learned something from him.

On two occassions, Luca Cardelli found the time to talk with me about my work, in its early stages. His criticisms and insights were truly invaluable. I hope I have succeeded in addressing some of his concerns.

Several faculty members here at Utah deserve honorable mention. Beat Bruderlin, Ganesh Gopalakrishnan, Bob Johnson, Bob Kessler and Jamie Painter were all helpful in a variety of ways.

My fellow graduate students have not only given useful comments on my work, but much more importantly, have been friends. Thanks go therefore to Venkatesh Akella, Art Lee, Surya Mantha and Rok Sosic. In the same vein, thanks to Cheol Kim, on the trans-siberian railway or wherever he may be.

I also want to thank the staff, past and present, at the Computer Science department front office for all their help, especially Susan Jensen, Colleen Hoopes and Marsha Thomas. Marilyn Owen and Charlene Urbanek at the University's International Center did a terrific job guiding me through that special bureaucratic jungle reserved for foreign students.

One of the most pleasant aspects of this research has been the opportunity to meet and interact with many good people working in the area of object-oriented programming. These are too numerous to mention, but I thank them all for stimulating exchanges, verbal and electronic.

Finally, thanks to Wu Weihong, for keeping me company these past five years.

CHAPTER 1

INTRODUCTION

> Language design is reminiscent of Ptolemaic astronomy - for ever in need of further corrections.
> *Jean-Yves Girard*

This dissertation addresses two problems: the problem of *multiple inheritance* in object-oriented programming languages, and the problem of *modularity* in programming languages.

On the surface, it may appear that these are two separate problems, and should be dealt with separately. Indeed, this study began as an investigation of multiple inheritance. A chief conclusion of that study is that the two problems are deeply related, and that solving either problem implies solving the other. In a sense, the two problems are one and the same.

This chapter aims to provide the reader with a bird's eye view of the dissertation as a whole. Accordingly, the chapter is structured as a dissertation in miniature. Each of the following sections mirrors one of the succeeding chapters.

Following a brief overview of the problem in section 1.1, section 1.2 discusses a partial solution. Then, in section 1.3, a comprehensive solution is sketched. The theoretical foundation for this work is outlined in section 1.4. Section 1.5 shows how the solution manifests itself in the context of an existing programming language, and section 1.6 discusses implementation. The chapter concludes with a summary.

1.1 Understanding the Problem

The first order of business is to understand what the problems are. Chapter 2 is entirely devoted to that task. Here, only highlights are given, with the objective of reviewing fundamental concepts, and establishing that a problem does in fact exist.

1.1.1 Inheritance

The first and foremost concept in this dissertation is inheritance itself. Inheritance is a powerful linguistic mechanism, introduced by object-oriented languages [21, 24]. Inheritance allows an incremental style of programming. Given an existing piece of software, the programmer can create a new one, simply by specifying how the new piece differs from a preexisting one.

The term "piece of software" is rather imprecise. It is more accurate to say that inheritance allows the programmer to create definitions, by specifying how new definitions differ from previous ones.

In object-oriented languages, software definitions that may be inherited are usually called *classes*.

Figure 1.1 shows an example of *single inheritance*. In single inheritance, a single preexisting definition is used as a basis for a new definition.

In this example, two classes are defined. The syntax **class** *Id* **is** ... binds the identifier *Id* to a class, specified by whatever constructs follow the keyword **is**. The

```
class road_vehicle  is
                     number_of_wheels = 4;
                     number_of_axles = number_of_wheels/2;
                    end;
class eighteen_wheeler  is  inherit road_vehicle
                              number_of_wheels = 18;
                              gross_weight = 10000;
                            end;
```

Figure 1.1. A simple example of inheritance

first class is defined by specifying its *attributes*. Various object-oriented languages support different kinds of attributes, with varying characteristics and terminology. Chief among these are *methods*, which are function valued attributes which may be redefined via inheritance, as described below. Other kinds of attributes are not important at this point.

What is important is how the second class is defined by inheriting from the first, as indicated by the phrase **inherit** road_vehicle. As a result, class eighteen_wheeler has all attributes of road_vehicle, except those attributes it has *overridden* by supplying alternate definitions, such as number_of_wheels. The inheriting class may also add additional attributes, such as gross_weight.

This situation is often depicted using graphs, as in Figure 1.2. Graphs for single inheritance are always trees. In *multiple inheritance,* a new definition is created using several prior definitions. The graph induced is a DAG (directed acyclic graph).

Single inheritance has won substantial acceptance as a useful technique for structuring programs. Multiple inheritance is much more controversial. There is no agreement on the appropriate semantics for multiple inheritance. In most current languages, multiple inheritance violates encapsulation [65]. The rules governing

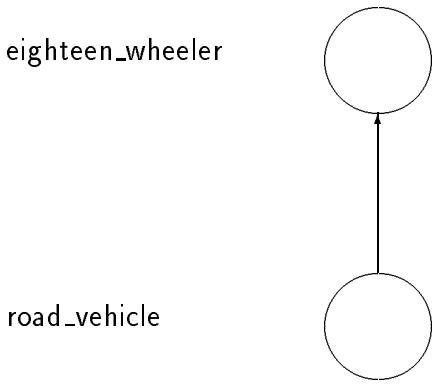

Figure 1.2. Inheritance hierarchy

multiple inheritance are complex. Proponents of multiple inheritance argue that it is essential, while critics contend that the perceived need for multiple inheritance is a symptom of poor program design, and that single inheritance is sufficient.

Finding a formulation of multiple inheritance that is rigorous, semantically clean and preserves modularity remains a difficult yet important problem.

1.1.2 Abstract Classes and Frameworks

One of the most useful ideas in object-oriented programming has been that of an *abstract class.* An abstract class is an incomplete class definition, in which one or more of the methods used by the class are not defined.[1] The expectation is that these missing method definitions will be provided in subsequently defined subclasses. In some languages, abstract classes have no special linguistic support. Programmers define "dummy" routines that typically produce an error if executed. More recent languages [23, 51] explicitly recognize abstract classes. In these languages, methods that are undefined in the abstract class are identified by special syntax. Here such methods are referred to as *pure virtuals,* using the terminology of *C++*.

Abstract classes are essential to the definition of frameworks [33]. A *framework* is a collection of classes designed to support a particular application in a modifiable and extensible manner. A framework is used as a basis for an application. Typically, some of the framework's abstract classes are modified and extended by inheritance to tailor them to specific needs. Examples of frameworks are [42, 45, 70, 73].

Abstract classes support a powerful form of parameterization, unique to the object-oriented paradigm. While standard parameterization allows structures to refer to parameters, abstract classes close the loop by also allowing parameters

[1] In some languages, a *declaration* of the undefined method may be provided, giving only type information.

to refer to the parameterized structure. Abstract classes are fundamental to the research presented in this dissertation.

1.1.3 Module Manipulation

Many non-object-oriented languages support some formal notion of module. However, there are usually only very limited facilities for *manipulating* modules. Mechanisms are provided for module interconnection, but these are usually ad hoc. An important exception is *ML* [52], which provides a well developed module manipulation language [44]. Integration of inheritance into *ML* in a modular fashion is the subject of ongoing research [53].

This thesis argues that inheritance is really an essential module manipulation mechanism. In languages without inheritance, its effects are obtained by extra-linguistic means (e.g., text editing). Thus, inheritance represents a natural step in the progression of linguistic support for modularity. Just as modules and interfaces subsume some functions supported by linkers, inheritance subsumes some functions of text editors. Incorporating inheritance in module manipulation languages is therefore a necessity.

In summary, the problem this dissertation addresses can be stated in two ways: find a formulation of multiple inheritance that is expressive, rigorous, semantically clean and preserves modularity; *or,* develop a comprehensive module manipulation language incorporating inheritance.

1.2 Mixins

Having described the problem, it is time to consider solutions. This section presents one solution, albeit partial. Valuable in itself, the partial solution is also a step leading to a more complete solution in section 1.3.

The key to a solution is to stop thinking about inheritance in operational terms. Consider the definition of eighteen_wheeler in example 1.1 once again. The definition binds the identifier eighteen_wheeler to the structure given after **is**.

To understand the meaning of a complex construct, it is useful to subdivide it into parts, and attempt to understand each part separately. The meaning of the structure as a whole should then be constructed from the meanings of its components. In this case, there are three parts: The keyword **inherit**; an identifier, road_vehicle; and the rest of the declaration. It is clear that road_vehicle represents the superclass, but what does the keyword **inherit** actually stand for? And what is the entity defined by "the rest of the declaration," something there does not even appear to be a name for? It turns out that **inherit** denotes an operator, that combines the other two parts of the structure. The third part of the declaration is something that will be called a *mixin*. Note that mixins cannot be named in existing programming languages. They are always an anonymous component of a larger class structure.

The easiest way to understand a mixin is to view it as a function from classes to classes. In that case, **inherit** stands for a form of function application. With this understanding, an improved formulation of inheritance is possible. The fact that mixins cannot be given names is now an obvious anomaly, and itself violates modularity. Once this anomaly is corrected, a form of multiple inheritance that preserves encapsulation is natural. However, this formulation, known as *mixin-based inheritance,* does not solve all the problems a module manipulation language must face.

Mixin-based inheritance is derived from an understanding of what mixins are, and will be described in detail in Chapter 3. Another insight has been that mixins and classes are composed by means of operators. This leads to a better solution, as outlined below.

1.3 Jigsaw

Jigsaw is a framework for modular programming languages. The word "framework" is used here in the much same sense given in subsection 1.1.2. The precise nature of the *Jigsaw* framework is the topic of Chapter 4.

The word "modular" here is used, quite deliberately, in two distinct ways. First, the languages designed using *Jigsaw* are (modular programming) languages; they support modular programming since programs in these languages may be divided into separate modules. Second, *Jigsaw* is highly modular in its own conception, permitting various module combinators to be included, omitted, or newly constructed in various realizations. This modular structure is inherited by all languages derived from the *Jigsaw* framework. Consequently, the languages produced are modular (programming languages).

In *Jigsaw*, the roles of classes in existing languages are "unbundled," by providing a suite of operators independently controlling such effects as combination, modification, encapsulation, name resolution, and sharing, all on the single notion of *module*.

Inheritance is understood to be an essential linguistic mechanism for module manipulation. All module operators are forms of inheritance. Unlike most formulations of inheritance, here inheritance is not in conflict with modularity. On the contrary, inheritance is the basic mechanism for module interconnection.

This allows a previously unobtainable spectrum of features to be combined in a cohesive manner, including multiple inheritance, mixins, encapsulation and strong typing. Traditional multiple inheritance is interpreted as an unsuccessful attempt to enhance the modularity of object-oriented programs. In the new framework, the distinction between single and multiple inheritance disappears, but the desired functionality remains available.

Jigsaw provides a notion of modularity independent of a particular computational paradigm. *Jigsaw* can therefore be applied to a wide variety of languages,

especially special-purpose and "little-languages" [5, Column 9], where the effort of designing specific mechanisms for modularity is difficult to justify, but which could still benefit from such mechanisms.

Jigsaw can be thought of as an abstraction, to be reified by application to a computational sublanguage, L_c. *Jigsaw* abstracts over L_c, but the abstraction is not merely parameterization. The interaction between *Jigsaw* and the language of computation is potentially bidirectional. This structure is exactly analogous to that typical of abstract classes and frameworks in object-oriented languages.

1.4 Semantics

Jigsaw has a rigorous semantics, based upon a denotational model of inheritance [18, 59]. Indeed, *Jigsaw* would not have been conceived without the insights derived from the study of the denotational semantics of object-oriented languages. The *Jigsaw* framework maps very directly to the underlying semantics. Modules have simple denotations, which are just functions from records to records. All module manipulation operators are defined by means of operations upon the denotations of modules. The denotational semantics are simpler than those of existing object-oriented languages, even though expressive power has been enhanced. Of course, the reason for this is that the linguistic constructs were inspired by the denotational semantics.

The reader should understand that this study does not introduce new theory. Instead, recent theory is applied to produce a new language design. Much work in denotational semantics is concerned with explaining existing linguistic constructs. That is one phase in a two phase process. The second step is using the understanding gleaned in the first phase to design better languages. This dissertation is concerned with this second phase.

1.5 Modula-π

Modula-π is an extension of *Modula-3* [55] that supports some of the key operations of the *Jigsaw* framework. The purpose of this extension is to demonstrate the applicability of *Jigsaw* to realistic programming languages.

Extending an existing language has several benefits. An upwardly compatible extension means that existing code is not invalidated. The existing implementation can be used as a basis for the extension. As a result, realistic performance can be achieved. For the language designer, using an existing language as a base is a mixed blessing. Designing an upwardly compatible extension is a difficult challenge. Many irrelevant details must be considered, and the purity of the model may be compromised. On the other hand, the rich functionality of the base language is already defined. The end result can be a tool that is realistic and practically useful, both in the range of its features and in its performance. It would be difficult, if not impossible, to achieve this goal within the scope of a dissertation, if a new language were to be defined and implemented.

A programming language's value is greatly enhanced if it can be implemented efficiently. This is discussed in the next section.

1.6 Implementation

Jigsaw is a linguistic framework, applicable in many different contexts. Each such context may place different requirements on an implementation. The main focus of implementation in this work has been on the *Modula-π* language.

The uniform structure of *Jigsaw* allows for a simple yet efficient implementation scheme. Simple modules (or object types in the case of *Modula-π*) are translated into dispatch tables which refer to methods. Object type instantiation, as usual, leads to the creation of an object, containing data and housekeeping information. Objects of course refer to the dispatch tables.

Given this representation, module operators are implemented as operations on dispatch tables, concatenating or modifying them as appropriate. The representations of corresponding instances are also manipulated in a similar manner.

The exact scheme used is presented in Chapter 7. It is an extension of existing techniques for implementation of object oriented languages [23]. The performance of this scheme is on a par with the methods employed by the highest performance object-oriented language processors currently available.

There are also published schemes for implementing more flexible language constructs, and their application to a language like that described in Chapter 4 is briefly analyzed.

1.7 Conclusions

In conclusion, it is appropriate to summarize key results, disclose some of the present research's limitations, and examine possible lines for future inquiry.

The solutions investigated here still leave certain issues unresolved, but they go a long way toward a fully satisfactory answer, and they point out promising directions toward a complete solution.

Jigsaw's chief limitation is that it is restricted to structural typing. Structural typing is fundamental to a truly modular system, and has naturally been the main focus of this work. Nevertheless, name-based typing and abstract datatypes are important issues that should be addressed. The prospects for an extension of *Jigsaw* dealing with name based typing are good. Abstract data types present a more difficult challenge. *Jigsaw* is defined by means of a denotational semantics. An axiomatic characterization of *Jigsaw* would be desirable as well.

Despite the admitted weaknesses just mentioned, a general and useful framework has been established for modular, incremental software construction. A module manipulation language that meets the criteria of expressiveness, theoretical soundness, efficiency and language independence has been developed.

All of the above issues, as well as related work, are discussed in Chapter 8. Now it is time to move on, to a full treatment of all the issues raised in this chapter.

CHAPTER 2

THE PROBLEM

> Operational reasoning is a tremendous waste of mental effort.
> *Edsger Dijkstra.*

This chapter illustrates the problem that this dissertation addresses: the difficulties of existing programming languages with respect to modularity. The concept of *module* is defined, and criteria for a modular programming language are given. Then, the evolution of programming language design toward enhanced support for modularity is examined.

Limitations of existing programming languages are discussed in light of the analysis mentioned above. No existing language meets all the criteria for a modular programming language. Special scrutiny is reserved for languages with inheritance. These languages have a variety of problems with respect to modularity. The well known encapsulation problems first demonstrated by Snyder [64, 65] are reviewed.

In the process of evaluating current language designs with respect to modularity, some novel insights are gained. Inheritance is identified as a necessary module manipulation mechanism. In addition, another important limitation on modularity, the absence of mixins, is discussed.

The chapter is organized by topic, not by language. Specific languages that exhibit a particular problem are mentioned in the appropriate section. The final section summarizes the problems of a variety of important programming languages.

2.1 Modules and Modularity

2.1.1 What is a Module?

The terms module, interface and system are defined informally below. The definitions are mutually recursive.

A *system* is a group of interconnected modules, that together fulfill some useful functionality.

An *interface* specifies a set of services, including conditions which must be met so that the services can be provided. Ideally, such an interface constitutes a logical specification of a module, stating necessary and sufficient conditions for its use, and giving a description of the system state after the module has provided a particular service.

In practice, such specifications cannot be verified automatically, and so, programming language interfaces are restricted to syntactic information that can be statically checked. A *module* is anything that supports an interface.

Using modules has two main implications.

1. Many different modules can be used at a given point, to provide a particular functionality. Anywhere a certain interface is required, any module that supports that interface can be used. Alternative modules can implement an interface, and can be interchanged freely, without influencing other modules in a system. It is therefore easier to locate and correct a performance problem or a reliability problem. Systems can be designed so that every module has sole responsibility for a particular function. Multiple modules can implement an interface simultaneously. In the context of programming languages, this means multiple implementations coexisting in a single program.

2. A module can be used at many different points. A module supporting an interface can be used wherever the interface is required. This means that

the same module can be (re-)used in many different contexts. Correcting a deficiency in the design of a module, corrects the deficiency everywhere the module is used.

Both these phenomena contribute to localization, making it easier to maintain a system. Because a module is independent of its context, it can be developed and understood independently, making it easier to design and maintain.

2.1.2 Desiderata for Modules

1. *Encapsulation.* Modules must be able to encapsulate information within themselves, so that no other module may access it. This guarantees that a module can be used only in accordance with its interface.

2. Composability. Any module can be combined with any other module with a compatible interface. The behavior of an assembly of modules can be deduced from the behavior of its component modules and their common interfaces.

3. Hierarchy. Modules can be built out of smaller modules, which in turn can be built out of smaller modules, and so on. This is distinct from composition, where modules are combined at the same level.

4. *Modifiability.* This property actually stems from hierarchy and compositionality. A module can be extended by combining it with other modules, or submodules of it can be replaced by alternate submodules. The replacement can be a modification of the original submodule.

5. *Static safety.* As noted above, it is generally impossible for a compiler to verify statically that a module is used correctly with respect to its interface. However, those syntactic properties that can be statically verified, should be checked.

6. *Separate Compilation.* A module may be separately compiled. This is necessary for parallel development, and reflects the fact that modules can be independently developed.

7. *Generality.* This is a rather vague notion. One advantage of modularity is reusability. The more general a module is, the more contexts it can successfully be reused in. The constructs of the language determine what degree of generality can be achieved by modules in that language. Polymorphism is a very important property that influences the degree of generality attainable. Polymorphism is a natural consequence of untyped specifications, but obtaining polymorphism while maintaining static typing is a difficult problem.

8. *Manipulability.* Ideally, modules are first class values in the language. This allows combinations of modules to be described in the language, and hence to be modularized themselves.

2.1.3 The Trend Toward Modularity

There has been a historic movement in programming language design toward providing increasing support for modularity. The trend has been to move functionality that was supported by tools in the program development environment into the language. Explicit linguistic support for modularity has several advantages. Various language-specific semantic consistency constraints can be imposed. Features are also much more likely to behave in a standard and portable way if they are defined within the language than if they are implementation dependent.

The need for separate compilation was recognized early on, beginning with *FORTRAN*. Separate compilation has been supported through external linkage. The minimum requirement is that the language processor recognize external references, and produce sufficient information for an extra-linguistic tool (the linker) to effect the module interconnection.

This state of affairs has several well known drawbacks. No static typechecking is performed across module boundaries. There is also poor support for encapsulation; typically, all globals in a module are available to other modules.

Later languages such as *CLU* [43], *Ada* [1], *Modula-2* [74], *Modula-3* and others provide a structured way of specifying modules and their interconnection. Formal notions of module and interface are part of the language. The language semantics guarantee that modules are used in accordance with their interface. This means that modules may explicitly encapsulate information, and that intermodule typechecking is supported.

Unfortunately, many functions related to modularity are still not supported in these languages. In practice, tools from the surrounding environment are used to perform these functions. This will be demonstrated in the following sections, where modularity problems in existing languages are examined.

2.2 Modularity Problems in Existing Languages

2.2.1 Flat, Global Name Spaces

In most languages, there is a flat, global name space for modules. This makes it difficult to resolve name conflicts. If a conflict arises, one of the conflicting modules must be renamed. Since modules are defined at the "top level," there is no scope in which a renaming operator can operate within the language. The only way to achieve the desired renaming is then physical editing of the module text. Editing is undesirable for several reasons. First, it is a manual process and hence laborious and error prone. Second, editing leads to the creation of multiple versions of a module, complicating program maintenance. Third, it requires recompilation of the edited modules. The disadvantages of renaming by editing are aggravated by references to a module in other modules. Such references often arise when specifying intermodule connectivity. Examples would be the use of class declarations in *C++* header files, or the use of **import** declarations in the *Modula* language family.

This is illustrated in Figure 2.1, where renaming module1 requires changing not only module1 itself, but additional modules, such as module3. Each such additional module now has the same problems of multiple versions and recompilation. Furthermore, if source code is not available for both conflicting modules, renaming by editing is not possible. If large scale reuse [20] becomes a reality, object modules provided by different vendors have to be combined, and name conflicts will become harder to avoid.

In practice, verbose naming conventions are used to minimize name conflicts, and the scale of reuse is presently small enough for the problem to be kept under control. Of course, if the problem is not addressed, the scale of reuse may never grow.

A global name space is a violation of the hierarchy criterion (3) mentioned above. There must not be a "top level" of the module hierarchy. As noted by Bertrand Meyer, real systems have no top [51][pg. 47].

2.2.2 Lack of Inheritance

This subsection makes the case that inheritance is really a module manipulation mechanism. This is demonstrated in Figures 2.2 and 2.3.

In Figure 2.2, two classes are defined. The first class, Point, describes points in the plane. A Point has coordinates x and y, and two methods. The dist method computes the distance to another Point passed as a parameter. The method closer determines if the Point is closer to the origin than its parameter, aPoint.

```
module module3;
import module1, module2;
.
.
.
module1.procedure2(module2.function3(2) + 5);
```

Figure 2.1. Use of **import** declarations

```
class Point  is
              x = 0; y = 0;
              dist =  function(aPoint)
                        {
                          sqrt(sqr((x - aPoint.x)) + sqr((y - aPoint.y)))
                        }
              closer =  function(aPoint)
                          {
                            dist(Point(0,0)) < aPoint.dist(Point(0,0))
                          }
            end;
class ManhattanPoint  is  inherit Point
                        dist =  function(aPoint)
                                  {
                                    (x - aPoint.x) + (y - aPoint.y)
                                  }
                        end;
```

Figure 2.2. Code for Point and Manhattan_Point.

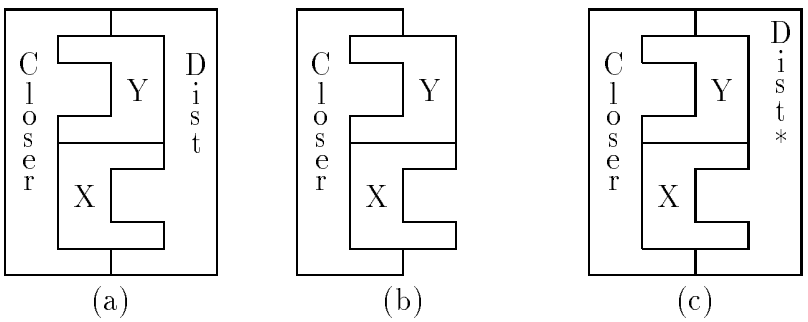

Figure 2.3. Inheritance as module manipulation.

The second class, Manhattan_point, is derived from Point by inheritance. The only difference between a Point and a Manhattan_point is the notion of distance they employ. Note that when the closer method is invoked on a Manhattan_point, the new dist method will be used, even though no explicit change has been made in the closer method. This illustrates an essential characteristic of inheritance; modifications are reflected in all self-reference within a structure.

The process of inheritance is represented graphically in Figure 2.3, using the metaphor of a jigsaw puzzle. Figure 2.3(a) shows a pictorial representation of class Point. The Point class is a module, composed of submodules which are its attributes. The derivation of Manhattan_point from Point is schematized in 2.3(b) and (c). The original definition of distance, Dist, is removed in (b). Then, a new definition, Dist*, is inserted instead. The references to the distance function in other parts of the class now refer to the new definition. This is exactly what one expects to happen when replacing one physical part by another within an assembly of parts.

In a modular system, it is always possible to remove a module from a larger assembly of modules, and then insert another, compatible module into the assembly. In the jigsaw puzzle metaphor, inheritance amounts to picking up a piece of the puzzle, and replacing it with another piece. The new piece must fit in the slot occupied by the original. This reflects the need for interface compatibility, so that existing references not be invalidated.

Inheritance is a language construct for expressing the sort of module manipulation discussed above. A language that does not support such a construct is clearly deficient in its support for module manipulation, violating the modifiability criterion (4 above).

In practice, modular programming languages provide no notation to express inheritance. Usually, there is no notation for manipulating modules at all. Even languages that do support module manipulation (e.g., *ML, Jade* [58]) are hampered

by lack of inheritance. Modification is achieved using an extra-linguistic tool, a text editor. Again, all disadvantages noted earlier apply. Access to source code is required. Recompilation is necessary. Multiple copies of modules are introduced. No semantic constraints are enforced. Errors are easily introduced, and the entire process entails more work than necessary. Another difficulty is that the granularity of module constructs is often inappropriate. Often, the changes needed are replacements of individual functions within a module, as the last example has shown.

In summary, inheritance is a linguistic mechanism that supports actions that occur naturally and frequently in modular systems. Its introduction into programming languages is an extension of a natural progression of increasing support for modularity in programming languages.

2.3 Difficulties with Inheritance

In contrast to the interpretation of inheritance as a modularity mechanism given above, the actual inheritance mechanisms available in current languages are in fact in conflict with modularity. This section discusses the modularity problems that arise in languages that incorporate inheritance. Snyder's classic paper [65] showed how inheritance commonly undermines modularity. Snyder's observations are recalled here, since they are central to this work. A modularity problem not discussed in [65] is that certain program constructs cannot be effectively modularized. This is addressed in section 2.3.5.

The next three subsections illustrate different manifestations of essentially the same problem: exposure of a class' use of inheritance to its clients. This violates criterion 1 - encapsulation. Inheritance is used to construct modules; it is an implementation mechanism. If it is visible to clients, then these clients may come to rely on the inheritance structure used. If that structure is changed, clients may cease to function correctly.

Subsection 2.3.4 discusses how inheritance may make visibility control unnecessarily complex, and constrain a client's design. Finally, subsection 2.3.5 shows how the absence of mixins makes most object-oriented languages incomplete with respect to modularity.

2.3.1 Classes and Types

In many object-oriented languages, types are identified with classes and subtyping with inheritance.

The distinction between class and type is absolutely crucial. A class is a unit of implementation (ideally, a *modular* unit). A type is a (partial) description of behavior - a statically verifiable interface. The distinction is essentially that between interface and implementation, and is well understood with respect to abstract data types. A class always has a type associated with it, but not vice versa. A type can be implemented by many different classes, as shown below.

2.3.1.1 Multiple Implementations of an Abstraction

Identification of classes and types would seem to preclude supporting multiple implementations of an interface within a single program. In practice, when multiple implementations of an abstraction are required, the notion of abstract class is often pressed into service as a substitute for interfaces. In this case, the abstract class provides no definitions at all, only declarations. This is inescapable when a language fails to distinguish between types and classes. A major disadvantage of this technique is that it requires advance planning. In languages where types and subtyping are separated from classes and inheritance, this subterfuge is unnecessary.

2.3.1.2 Subtyping and Inheritance

If classes and types are identified, so, per force, are the subtyping and inheritance hierarchies.

If a class A is defined by inheriting from classes B and C, then A is also understood to be a subtype of B and of C. From the viewpoint of modularity, this is undesirable. Should the designer of class A later wish to reimplement A using, say, D, E and F, the change would be visible to clients of A, because they may rely on the subtyping relation previously defined. In effect, the ancestors of A are part of its interface. By transitivity, the entire inheritance graph upstream of A is part of A's interface. Any change to this graph may affect the validity of class usage downstream of the change. This is an unbounded region, since new classes may be derived at any time. In practice, new classes are likely to defined at remote sites, that should not even be aware of the existence of the base class being changed. Consider an application based upon a framework supplied by a vendor. If the vendor chooses to reimplement a class, the application may fail.

In reality, inheritance hierarchies are hard to design correctly the first time, and need to be changed repeatedly. Changes in the hierarchy are difficult to make in languages with classes as types, because of the problem outlined above.

In *C++*, inheritance may be decoupled from subtyping, by declaring access to a base class to be `private`. However, this is of limited use, since the language provides no form of subtyping except that based on inheritance. If a class inherits without becoming a subtype, instances of the class cannot be used polymorphically.

C++ classes are distinguishable from types, but not in a very clear cut way. Membership in a type implies membership in a class,[1] and so subtyping implies subclassing. However, the converse is not always true, since an object of a subclass of some class A might not be a member of a subtype of the type of A. The fact that subtyping implies subclassing is valuable in an implementation, since it guarantees a large measure of structural compatibility among the objects operated upon by polymorphic code.

[1] Except for primitive types such as `int`, `float`, etc.

It is also possible to imagine a situation in which subclassing implied subtyping but not vice versa. This policy, suggested in [30], does not violate encapsulation, since information about the inheritance graph is not exposed through the type system. Multiple implementations of an interface are also possible. Inheritance is, however, restricted to create subtypes only. This limits the ways in which modules can be manipulated. The literature contains many examples of cases in which such restrictions are too harsh [8, 9, 19]. Current languages which unify types and classes, either restrict expressiveness in this way (e.g, *C++*), or have unsound type systems (e.g., *Eiffel*). In the case of unsound type systems, the problems may be rectified by use of dynamic typing, as in *Beta* [48].

2.3.1.3 Other Considerations

The separation of classes and types makes it easier to define orthogonal constructs for renaming [4, pp. 168] and visibility control. There are other reasons for separating classes and types. These have less to do with modularity. The interested reader is referred to [8, 9, 19, 48].

It is worth noting that there are arguments for merging the concepts of type and class. Programming languages have a long tradition of using type information for implementation purposes. Identifying the type of an object with its implementation is a natural consequence of that tradition, and makes it easier to devise an efficient language implementation.

Another longstanding tradition is that of name-based typing. Name based typing is motivated by modeling considerations; modules that share a common syntactic interface may represent semantically incompatible entities. Name-based, as opposed to purely structural, typing, can help prevent confusion between such modules. In the context of name-based typing, identifying classes with types seems natural.

Nevertheless, the disadvantages of merging types and classes seem to outweigh the advantages, especially as far as modularity is concerned.

2.3.2 The Diamond Problem

> One of the delicate problems raised by the presence of multiple inheritance is what happens when a class is an ancestor of another in more than one way. If you allow multiple inheritance into the language, then sooner or later someone is going to write a class D with two parents B and C, each of which has a class A as a parent - or some other situation in which D inherits twice (or more) from A. This situation is called repeated inheritance and must be dealt with properly.
>
> *Bertrand Meyer.*

In multiple inheritance, a class may inherit from an ancestor along multiple paths in the inheritance graph. The simplest such situation is shown in Figure 2.4.

The situation shown raises thorny questions. Does a FillCircle object contain one Ellipse subobject, or perhaps two (one for each path from Ellipse to FillCircle)? Name collisions must result from this state of affairs. Are these regarded as errors or not? If not, how are the conflicts resolved? Different languages treat these problems in different ways. It is instructive to review the approach taken by most major object-oriented languages.

Many languages follow a policy that is intuitive, and seemingly innocuous. The name clashes are harmless; the conflicting names all refer to the same method. The compiler can distinguish between cases such as that shown in Figure 2.4, and "real" name clashes, where the conflicting names arise from different definitions. This solution relieves the programmer from the tedious task of resolving many of the conflicts that arise in practice. This is the policy followed by *Eiffel* [51], *Owl*

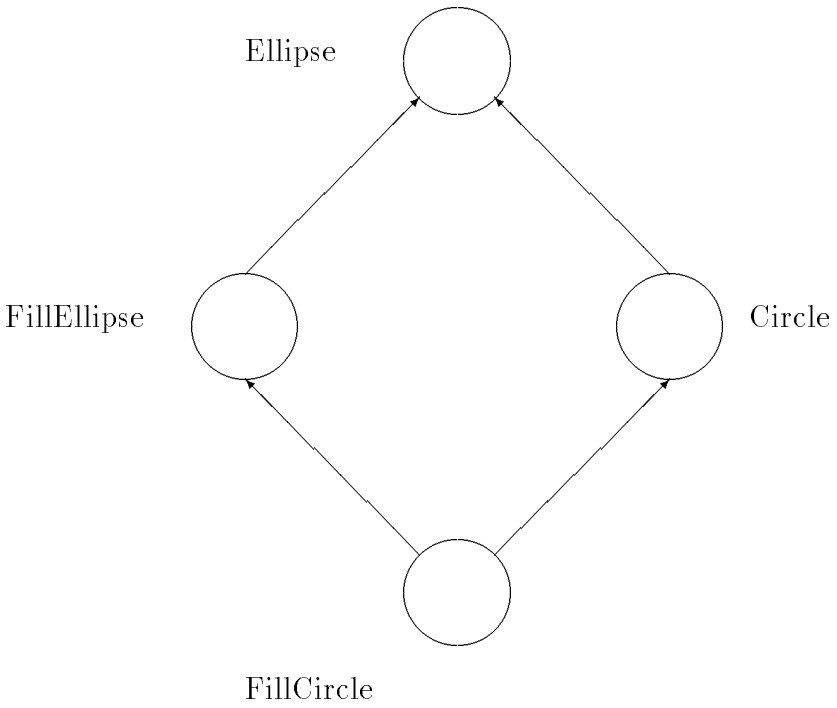

Figure 2.4. The "diamond" problem

[62], *CLOS* [35] and *SELF* [68, 69]. The reader may wish to ponder the obvious common sense of this approach before continuing.

The only modular solution is to treat the name collisions as errors, just as if the conflicting names had been defined in different classes. Similarly, each path in the graph must contribute a subobject. To do otherwise requires global knowledge of the inheritance graph. A class must not care about the provenance of the implementation of a particular method it is inheriting. If this is not so, a change in a remote ancestor can cause a class to break, as shown in Figure 2.5. Assume the hierarchy is reorganized , so that all filling is derived from a common root class FillGraphic. Fill classes must change, but not users of Fill classes. Since Ellipse and FillGraphic are likely to have method names in common (e.g., draw), name clashes

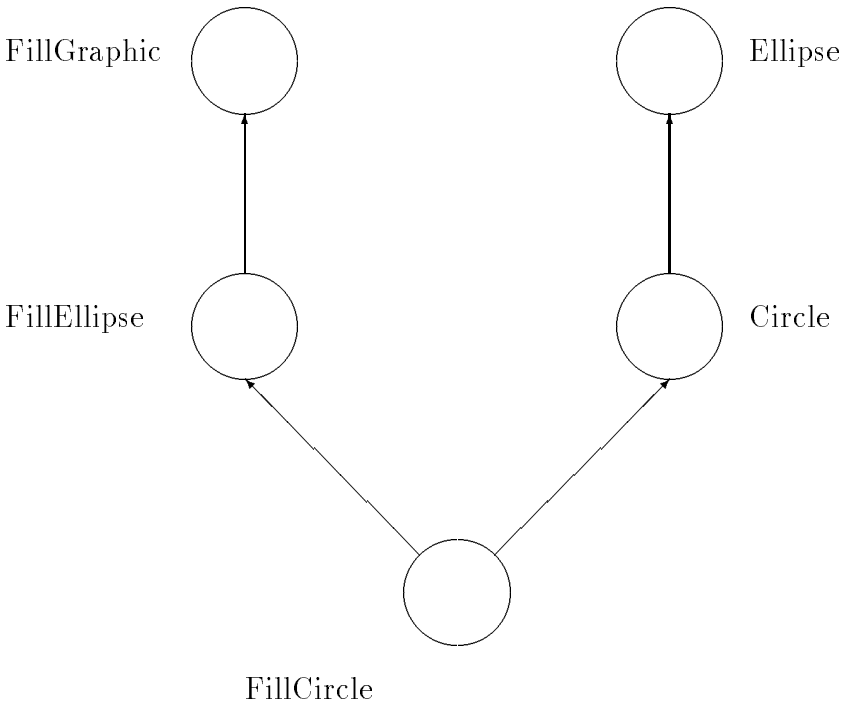

Figure 2.5. "Opening" the diamond.

will occur. Essentially, the problem is similar to that introduced by merging classes and types: knowledge of the entire inheritance graph "leaks" into the interface.

2.3.3 Accessing Indirect Ancestors

It is often necessary to access code that has been overridden. In some languages, the mechanism provided is to prefix the overridden method's name by the name of the class from which it was inherited. This is illustrated in Figure 2.6. Care must be taken that such access is allowed only within the inheriting class, and that only immediate ancestors may be referenced this way. Languages like *Owl* [62] that allow arbitrary ancestors to be accessed this way, expose the use of inheritance to clients. Consider Figure 2.7. FillCircle has a gratuitous dependency on the

```
class FillEllipse  is  inherit Ellipse
                       draw =  function()
                               {
                                 Ellipse::draw();
                                 Fill();
                                 ...
                               }
                   end;
```

Figure 2.6. Accessing an overridden method.

```
class FillCircle  is   inherit FillEllipse
                         radius =  function()
                                   {
                                     return Ellipse::minor_axis();
                                     // Should have used FillEllipse::minor_axis !
                                   }
                     ...
                   end;
```

Figure 2.7. Accessing indirect ancestors violates encapsulation.

implementation of FillEllipse. The programmer has assumed that the method for computing the minor axis of the ellipse was inherited from Ellipse. If the inheritance hierarchy is changed, FillCircle will either not compile, or worse, malfunction. Again, the problem is that changing the inheritance hierarchy has the effect of breaking downstream classes, as outlined in subsection 2.3.1.

2.3.4 Visibility control

In an object-oriented language, a class has two kinds of clients, users and heirs. Users utilize classes in the same way as client modules in more traditional languages use server modules, by invocation. Heirs inherit from a class, modifying it as necessary.

Heirs and users differ in the interface they require to the original class. Typically, heirs require access to a "wider" interface than users, in order to implement modifications and extensions efficiently. If only one interface is provided, it may be either

too "narrow," denying heirs the access needed for efficient implementation, or it may be too "wide," granting users unnecessary and potentially dangerous privileges. Designers of object-oriented languages have found it necessary to introduce two kinds of interface, corresponding to the two kinds of clients.

In *C++*, these interfaces are known as **public** (for users) and **protected** (for heirs). In *Owl*, heirs have access to a **subtype visible** interface. Similar ideas appear under the names of "internal" and "external" interfaces, in [53].

While these constructs do not strictly violate modularity, they seem overly complex, and introduce a subtle anomaly, pointed out in [25]. Once certain features of a class have been placed in the **protected** interface, those features can be accessed only via inheritance. A nested instance of the class does not provide access to that feature, since it is not in the **public** interface. This constrains the designers of client software. The choice between inheritance and nesting is no longer available to them. Modularity should guarantee the ability to associate multiple interfaces with a module, not just two. Furthermore, the linguistic mechanisms for using an interface should be orthogonal to what interfaces are available.

2.3.5 Limits on Module Construction

Previous sections have shown how languages make it difficult to combine modules, or impossible to define modules. This section illustrates restrictions on the way a module can be constructed.

Object-oriented languages originally supported single inheritance. The question whether single inheritance is sufficient is still the topic of some controversy [15]. Proponents of single inheritance argue that multiple inheritance is complex and poorly understood, that it is frequently abused, and that cases in which it is used could be better handled by single inheritance. Conversely, supporters of multiple inheritance argue that it is both natural and required. Arguments on both sides

are often anecdotal, and the debate sometimes suffers from confusion as to the relationship between inheritance, subtyping and modularity.

The essential characteristic of single inheritance is that it is not possible to combine several classes into one. Rather, one class may be modified to produce another. This process may be iterated, producing a path of successively more refined classes. Since a class may be modified in any number of different ways, this leads to a tree structured inheritance hierarchy. Usually this hierarchy also serves as a classification hierarchy. One common argument against single inheritance is that a tree structured classification scheme is inadequate to model relationships in the real world. However, the viewpoint advocated here is that inheritance is a modularity mechanism, not a classification mechanism. While tree structured classification is indeed limited, it is not a fundamental characteristic of single inheritance. If inheritance is divorced from subtyping, a language can support single inheritance simultaneously with overlapping (graph structured) classification. The fact that subtyping does indeed induce a lattice structure was demonstrated in the classic paper by Cardelli [10].[2]

There are, however, valid arguments against the restriction to single inheritance. Viewed as a modularity mechanism, single inheritance seems very constraining. It allows modification to a module, but does not allow for combination of modules. Multiple inheritance can thus be viewed as an attempt to make object-oriented languages more modular. Ironically, existing languages have tended to undermine modularity when introducing multiple inheritance.

A clear limitation on modularity in all existing object-oriented languages with static types is the existence of an entire class of software definitions that cannot be modularized at all. These definitions are known as *mixins*. Consider Figure

[2]The paper's title, "A Semantics of Multiple Inheritance," is a misnomer. It actually defines subtyping, not inheritance. The distinction is not made clear in the paper.

2.8. This example is very similar to that given in Figure 2.6. The only difference is that Ellipse has been replaced by Rectangle. In most object-oriented languages, and certainly in all those that employ static typing, there is no way to factor out the commonality evident in the example into a separate abstraction, let alone a separately compilable module. An important contribution of this chapter is the identification of mixins as candidates for programming language support. Their absence is a violation of modularity in a language supporting inheritance.

2.4 Problems by Language

This chapter has surveyed the serious modularity problems that exist in today's programming languages. To facilitate understanding, the presentation has been organized by problem, not by language. In order to convince the reader that there is no programming language that does not suffer from some of the problems discussed above, the relevant properties of most important programming languages are summarized in Table 2.1.

The languages listed include both the major languages in use today, and languages that are important for their innovative constructs, even if they are not widely used.

Each row in a table corresponds to one of the problems cited earlier in this chapter, and each column corresponds to a particular programming language. An entry marked "X" signifies that the language in question suffers from the corresponding

```
class FillRectangle  is  inherit Rectangle
                         draw =  function()
                                 {
                                    Rectangle::draw();
                                    Fill();
                                 }
                          ...
                    end;
```

Figure 2.8. Lack of mixins causes repetitive code.

Table 2.1. Problems by language

Language	Ada	Beta	C++	CLOS	CLU	CommonObjects
Problem						
Global Name Space	X		X		X	
Class = Type		X	X			
Diamond problem				X		
Remote ancestor access						
No Inheritance	X				X	
Single Inheritance		X				
No Static Typing				X		X
No Mixins		X	X		X	

Table 2.1. - Continued

Language	Eiffel	Haskell	Jade	ML	Modula-2
Problem					
Global Name Space	X	X			X
Class = Type	X				
Diamond problem	X				
Remote ancestor access					
No Inheritance		X	X	X	X
Single Inheritance					
No Static Typing					
No Mixins	X				

Table 2.1. - Continued

Language	Modula-3	Oberon	Owl	Self	POOL	Smalltalk
Problem						
Global Name Space	X	X	X		X	X
Class = Type			X			
Diamond problem			X	X		
Remote ancestor access			X			
No Inheritance		X				
Single Inheritance	X					X
No Static Typing				X		X
No Mixins	X		X		X	X

problem. A blank entry means either that the problem is handled correctly, or that the issue does not arise - for example, the diamond problem of subsection 2.3.2 does not arise in languages with only single inheritance, and the absence of mixins is not a deficiency in languages that do not support inheritance at all.

One of the things Table 2.1 makes clear is that there is no language supporting inheritance that combines static typing with the ability to express mixins. The following chapter studies mixins in detail, and shows how they may be used to address some of the problems this chapter has raised.

CHAPTER 3

MIXINS

> It was not obvious how to combine the *C++* strong static type checking with a scheme flexible enough to support directly the "mixin" style of programming used in some *LISP* dialects. *The C++ Annotated Reference Manual.*

As mentioned in Chapter 2, mixins can be used as the basis of a powerful form of inheritance, *mixin-based inheritance* [7]. Now is the time to investigate mixins more thoroughly. This chapter examines different ways in which mixins can be incorporated as full-fledged constructs in programming languages, and demonstrates the usefulness of such an endeavor. A more theoretical treatment of mixins is left for section 5.2.

The chapter begins with a review of the informal use of mixins in current programming languages. Next, the nature of mixins as abstractions is discussed. Appropriate linguistic formulations of mixins and mixin-based inheritance are then presented. In conclusion, the limitations of mixin-based inheritance are reviewed. This in turn, sets the stage for a more comprehensive approach to inheritance and its problems, in the next chapter.

3.1 Mixins in Existing Languages

The previous chapter introduced the notion of mixin, a construct that seems to be missing in existing object-oriented programming languages. In fact, mixins, as an informal construct, are present in several dynamically typed object-oriented programming languages. This is analogous to the use of while loops in *FORTRAN* programs - the construct is in use, but the language does not support it.

3.1.1 CLOS

The use of the word mixin as a technical term originates with the *LISP* community. It was first used by the developers of the *Flavors* [54] language. In *CLOS*, mixins are available as a result of two factors: dynamic typing and the notion of *linearization*.

In *CLOS*, all classes that contribute to an object's behavior are ordered linearly in a *class precedence list*. The ordering is determined by a linearization algorithm. Various algorithms may be used [22], but they all produce a linear ordering that preserves the partial ordering inherent in the original graph. Each contributing class occurs only once in the resulting precedence list. Linearization serves to disambiguate name clashes in multiple inheritance, but has serious negative consequences. Encapsulation is violated, as discussed in section 2.3.2. In addition, a class may not be adjacent to its immediate ancestors in the class precedence list produced. This may affect program behavior, and is heavily dependent on the specific linearization algorithm used, and on the global structure of the inheritance graph.

Classes in *CLOS* may refer to their ancestors using a special function, call-next-method. This allows access to overridden methods, as mentioned in section 2.3.3. When executing a method, an invocation of call-next-method will invoke the method of the same name, as defined on the next class on the class precedence list.

Given the absence of static checking, it is possible to place an invocation of call-next-method in a class that does not have any ancestors. Of course, the invocation will fail if the class is used by itself. The designer of a mixin relies on the linearization algorithm to place the mixin before other classes in the class precedence list to achieve the effect of binding a mixin to a parent. Thus, mixins are expressible in *CLOS* as a by-product of the procedural model of inheritance used by the language. Mixins are not expressed as explicit abstractions, nor do

they have any formal language support. *CLOS* is representative of the approach taken by a variety of *LISP* dialects with respect to inheritance. The main exception is *CommonObjects* [63, 65], which is discussed in Chapter 8 in the context of related work.

3.1.2 SELF

SELF, like *CLOS*, is dynamically typed. Unlike *CLOS*, it is not based upon a linear form of inheritance, but rather upon *delegation* [2, 6, 41, 66]. Delegation is a form of inheritance that occurs between objects (often referred to as *prototypes*) at execution time, rather than between classes at the time of compilation.

Like *CLOS*, *SELF* has a built-in mechanism for accessing overridden methods (known as **resend**). Since no static typechecking is performed, it is possible to define objects which use **resend** without binding them to parents. Using delegation these objects can be bound to a parent object later in the program execution. It is the programmer's responsibility to ensure that such a mixin object is bound to a parent before being used.

Since objects are first-class values that may be abstracted over, it is also easy to write a method that takes an object as an argument and uses it as a parent for another object. This method insures that binding to a parent takes place. However, in *SELF* this assurance is of limited value, because there is still no guarantee that the parent will support the interface expected of it.

SELF's approach is much more satisfactory than the one taken by *CLOS*, since mixins are available as a natural consequence of delegation, rather than as an artifact of undesirable linearization.

3.1.3 Beta

Beta uses a form of single inheritance called *prefixing*. When a class (known as the *prefix*) is modified through prefixing, the language guarantees that the prefix's

original code will be executed. The prefix determines if, and at what point in the code, the modification's (*extension* in *Beta* parlance) code will be invoked. This is indicated by the keyword **inner**.

Beta uses the concept of *pattern* uniformly for classes, types, functions and procedures. *Beta*'s syntax can be disconcerting for novices, so here a more conventional notation is used.[1]

Figure 3.1 is an example demonstrating how prefixing works in *Beta*. Two classes, Graduate and Person, are defined. The definition of Graduate is said to be *prefixed* by Person. Person is the *superpattern* of Graduate, which, correspondingly, is a *subpattern* of Person. Display is declared to be **virtual**, which means that it may be extended in a subpattern. This does not mean that it may be arbitrarily redefined, as in most object-oriented languages.

The behavior of the display method of a Person is to display the name field and then perform the **inner** statement. For a plain Person instance, which has no inner behavior, the **inner** statement is a null operation (i.e., skip or no-op). When a subpattern of Person is defined, the inner statement will execute the corresponding display method in the subpattern.

[1]This syntax is used by the implementors of Beta for tutorial purposes [39].

```
Person: class
(# name : string;
   display: virtual proc
     (#  do name.display; inner #);
#);

Graduate: class Person
(# degree: string;
   display: extended proc
     (#  do degree.display; inner #);
#);
```

Figure 3.1. *Beta* prefixing

The subpattern Graduate extends the behavior of the Person display method by supplying inner behavior. For a Graduate instance G, the initial effect of G.display is the same as for a Person: the original method from Person is executed. After the name is displayed, the inner procedure supplied by Graduate is executed to display the graduate's degree. The use of **inner** within Graduate is again interpreted as a no-op. It only has an effect if the display method is extended by a subpattern of Graduate. Notice how in Beta prefixing, the prefix controls the behavior of the result.

Figure 3.2 shows how a mixin can be defined in *Beta*. The objective is to capture the "graduate behavior" embedded in the subpattern (# degree: ...; display: ... #) in an abstraction, so it need not be repeated time and again. The mixin is called GraduateMixin, and is defined in a rather involved way. GraduateMixin comprises two nested classes, Super and Result. GraduateMixin should be thought of as a function from classes to classes. Super represents the function's formal parameter, its input.

```
Displayable: class (# display: virtual proc (# inner #) #);

Person: class Displayable (# name: @String;
                display : extended proc (# do name.display; inner #) #);

GraduateMixin: (# Super : virtual class Displayable; (* Formal Parameter *)
          Result: class Super (# degree: @String ;
                                 display : extended proc
                                        (# do
                                           degree.display
                                        #)
                               #) (* Desired Combination *)
        #)

GraduatePattern: class GraduateMixin (# Super : extended class Person #);
(* Pass "Person" as actual parameter *)

Graduate: class GraduatePattern.Result (* Extract Final Result *)
```

Figure 3.2. Mixins in *Beta*

Result represents the function's output. Super must be a subclass of Displayable. Displayable is an abstract class whose purpose is to serve as an interface specification, or "type," for Super. This kind of use of abstract classes is always necessary when types and classes are not clearly distinguished, and was discussed in section 1.1.2.

What the GraduateMixin "function" computes is a new class, Result, which extends the input parameter Super with "graduate" information. The "invocation" of GraduateMixin proceeds in two stages. First, an extension GraduatePattern is defined. The extension refines the class Super to be class Person. This is the analog of passing Person in as an actual parameter. Person is a subclass of Displayable, so it is a valid argument. The second stage is to explicitly retrieve the "output." This is done by selecting Result from GraduatePattern.

The solution takes advantage of *Beta*'s unusual ability to nest classes in an arbitrary fashion, and redefine nested classes via inheritance. The approach taken here is closely related to *Beta*'s use of nested patterns to represent genericity or procedures as parameters [49].

Support for mixins in *Beta* is not deliberate, however; until an early version of this work was circulated, no one, including *Beta*'s designers, had investigated use of mixins in *Beta* [47]. This explains why it is rather awkward to define a mixin in *Beta*.

The idea that mixins can be treated as functions from classes to classes is valuable. In the next section this idea will be made readily apparent, free of *Beta*'s somewhat idiosyncratic syntactic and conceptual baggage.

3.2 Mixins as Abstractions

Tennent's principle of abstraction [67, page 114] states that "any semantically meaningful syntactic class...can in principle be used as the body of a form of abstract." The introduction of mixins into object-oriented languages is a direct application of this principle. Since a mixin is not inextricably bound to any

particular parent, we can regard a mixin as being parameterized by a parent, which it is modifying. So mixins can be treated as functions from classes to classes.[2] This is shown in Figure 3.3. The BorderWindow function accepts an argument W, which must be a Window, and returns a result that is a modified version of the input class W. The modification adds a border, to be displayed around the window. This requires a new display routine, which first displays the window's body, and then surrounds it with a border.

The notation W : Window requires some discussion. This syntax is clearly analogous to standard programming language notations like i : Integer, which signify that i is a variable denoting a value belonging to a collection of values known as Integer. Similarly, W denotes a class that belongs to the collection of classes known as Window. Such collections of classes will be referred to as *interfaces.* The intuition is that W must be a class that supports the interface specified by Window.

Existing object-oriented languages do not have a formal notion of interface. However, many have a notion of type (which may be distinct from the notion of class, but usually is not.). In that case, an alternative notation, W <: Window, can be used. This can be interpreted as stating that W is a class whose instances have type Window (or some subtype of Window). The most common situation is

[2] In fact, that is one of several semantic views of mixins, and not exactly the one originally developed in [7]. See section 5.2 for more details.

```
BorderWindow[W : Window] = inherit W
    borderWidth = 5; borderColor = red;
    display = function(dontCare: Unit)
                  {
                    W.display();
                    displayBorder();
                  }
    displayBorder = function(dontCare: Unit) { ... }
```

Figure 3.3. Generics as mixins

that classes and types are identified (as discussed in Chapter 2). The reading of W <: Window then reduces to "W is a subclass of Window."

A corollary of the view of mixins as functions from classes to classes is that if classes and inheritance are first class operations, mixins fall out automatically. That is essentially what happens in *SELF*, where objects and delegation are first class, as noted in section 3.1.2.

Making inheritance a runtime operation may not be desirable. One consideration is that a high performance implementation becomes much more difficult. Another complication is that static typechecking of such constructs is problematic to say the least. This last problem will be discussed shortly. However, it is also possible to define abstractions over classes without taking the radical step of making inheritance a first class (runtime) operation. Such abstractions are the topic of the next subsection.

3.2.1 Mixins and Type Abstraction

Many programming languages support abstractions over "second-class" entities, such as types, classes or modules [43, 1, 62, 51, 55, 23, 26]. These constructs are often referred to as *generics*. In some languages, generics are merely macros, separately expanded and recompiled for every application of the abstraction. This is the case in *Ada*, *Modula-3* and *C++*. Such constructs are easily incorporated into almost any language. However, they preclude separate compilation of the abstraction they represent, and are nothing more than syntactic devices, with no semantic content.

More significant are constructs such as *Owl* type modules and *ML* functors, which are compiled only once. In object-oriented languages, generics have been used to define container classes, such as stacks or linked lists, that are conveniently parameterized by the type (or class) of objects they contain. It appears that one

could easily use such a construct to express example 3.3. Still, existing languages preclude such usage.

The reason is that guaranteeing the type-correctness of such an abstraction is in general extraordinarily difficult. Figure 3.4 illustrates the problem. A mixin M is defined that adds a boolean-valued method x to its argument. M is then applied to class P. P meets the requirements in the abstraction's header: P <: A. However, C is a malformed class because the x attribute of P conflicts with the x attribute added on by the abstraction. Note that if the type of the actual parameter (P in this case) was known *exactly* then this problem could not arise. However, a typical mixin is not useful unless it can be applied to classes with various interfaces. In other words, useful mixins are *polymorphic;* they are meaningfully applied to arguments of different (though related via subtyping) types. The difficulty is that while useful mixins are polymorphic, it appears that without exact type information, one cannot guarantee the type safety of inheritance.

Various typing schemes have been developed in an attempt to address this problem [13, 27].[3] None seems to present a solution that is simple and understandable enough to be useable by programmers, efficiently implementable, and covers the important cases. The problem is an exceedingly difficult one, and remains the subject of intense research. Related typing problems will arise repeatedly in this dissertation.

Rather than attempt (or wait for) a general solution, an alternative is to restrict

[3]Typically, what is actually studied is polymorphic record concatenation, but the problems are essentially the same.

```
P = A  inherit x: Bool → Bool = ...; ...  end;
M[B <: A] = B  inherit x: Real to Real = ...; ...  end;
C = M[P];
```

Figure 3.4. Mixin application

the problem. While the ability to inherit within a polymorphic abstraction is sufficient to define typed mixins, it is not necessary. Defining dedicated constructs for expressing mixins is a pragmatic alternative, discussed next.

3.2.2 A Dedicated Construct for Mixins

Mixins, expressed as abstractions, have a common form, as indicated in Figure 3.5. A mixin's signature contains sufficient information to determine the type correctness of an application (assuming exact type information about the actual parameter is available). The key is to recognize the characteristic form of a mixin's signature, especially its range.

First, some notation must be introduced. The leftward arrow $\leftarrow$ is the override operation on *interfaces.* If R,S are interfaces, R *override* S is the interface that results when R and S are concatenated, with the proviso that, if any attribute names are defined in both R and S, then

- the attribute value from S is used in the result.
- the type of the S attribute value must be a subtype of the type of the R attribute value.

The notation forall T <: S. type-expr means that within the type expression type-expr, T is a bound variable that denotes a type that is guaranteed to be a subtype of S. The types in question should be interpreted as the interfaces associated with classes.

Figure 3.6 shows that the actual result type of the mixin depends on the type of its actual argument. The mixin aMixin can be thought of as a polymorphic function

aMixin[T <: S] = T **inherit** *some modifications*

Figure 3.5. The common form of mixins.

between classes. For all classes with interface T, where T is a subinterface of S, aMixin takes a class and produces a new one, whose interface is given by (T ← ...).

At the point of application, the result type will be malformed if the argument is inappropriate. One can then detect statically any maltyped mixin applications. As long as inheritance manifests itself in the abstraction's signature, type safety can be guaranteed.

It is therefore imperative to ensure that every use of inheritance inside an abstraction is indeed reflected in its signature, and thus propagated to the top level, where exact type information is available. The easiest solution is to define a dedicated construct, as in Figure 3.7. A mixin abstraction of this form could be invoked with an actual parameter just like an ordinary generic. The meaning of the invocation is defined as M[R] = R **inherit** *body*. The crucial restriction is that the formal parameter R may not be inherited from, directly or indirectly in *body*. The invocation is legal as long as R <: S and the result type of the mixin invocation (which depends on R) is well-formed. This handles many interesting cases, and guarantees type safety.

3.2.3 Mixin-based inheritance

The use of mixins naturally introduces a new form of multiple inheritance, *mixin-based inheritance*. Mixin-based inheritance subsumes other forms of linear multiple inheritance, typical of *LISP* based object-oriented languages. If formulated with care, mixin based inheritance is a truly modular form of inheritance.

aMixin: forall T <: S. T → (T ← ...)

Figure 3.6. The signature of a mixin.

M = **mixin**[T <: S] *body* **end**

Figure 3.7. A dedicated mixin construct.

Figure 3.8 shows a simple multiple inheritance hierarchy. This hierarchy can be linearized in two different ways, as illustrated in Figure 3.9. Both linearizations can be defined using mixins. The first corresponds to C[B[A[Base]]], where Base is a simple base class with no attributes. The second linearization is likewise representable by C[A[B[Base]]]. In general, any linear encoding of a multiple inheritance hierarchy can be represented by a series of nested mixin invocations, as long as all classes (except Base) are represented as mixins.

Note that modularity need not be violated here. No implicit linearization is performed. The linear order is determined explicitly by the programmer. If the other precepts of [65] are followed, this form of inheritance does not violate encapsulation.

The formulation described up until now provides essentially the same level of functionality provided in [28]. Further refinements are developed below.

The next logical step is to define combinations of classes as mixins, so that when the new combination is used, the same flexibility is available. Instead of writing C[B[A[Base]]], define **mixin** CBA[X] C[B[A[X]]] **end**. This raises a problem. The argument X is being inherited from, within the body of the abstraction, contrary to

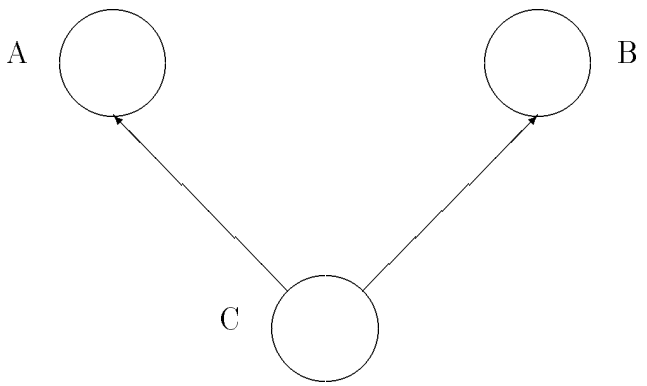

Figure 3.8. A simple multiple inheritance hierarchy

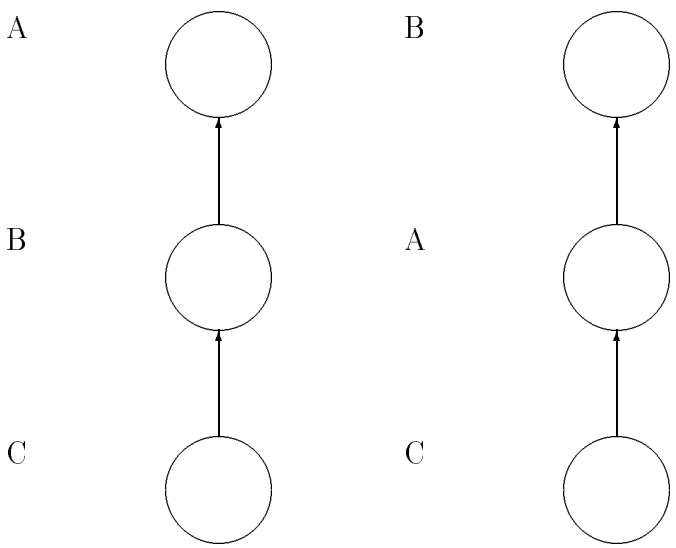

Figure 3.9. Linearized hierarchies

the restriction given above. Fortunately, the restriction can be eased in this case, since the use of inheritance is reflected in the signature of the mixin CBA.

Prior to instantiation, mixins must be bound to a parent, as in **new** CBA[Base]. The type system will be able to determine if Base is a valid parameter for the mixin being instantiated. If not, the mixin is not ready for instantiation, since it still makes nontrivial use of its parameter.

3.2.4 Mixin Composition

Mixins like CBA are more concisely formulated via *mixin composition*. In this context, mixin composition is exactly function composition. Define

$$(M_1 \circ M_2)[M_3] = M_1[M_2[M_3]].$$

Then $\mathsf{CBA} = \mathsf{C} \circ \mathsf{B} \circ \mathsf{A}$.

The style of programming that emerges from the examples above is one in which all user-defined classes are defined either as mixins or mixin compositions. This leads to the idea that only one construct, the mixin, is really needed by users, and that mixin composition is the normal mode of combining these constructs. This design is explored next.

3.3 Elevating Classes to Mixins

Instead of representing mixins with a new construct, an existing construct, the class, can be generalized. In some cases, this approach is more natural. For example, in *Beta*, the entire language is centered around a single abstraction, the pattern. Adding a new, special-purpose construct solely for mixins would not fit in such a framework at all.

The main attraction of this approach is uniformity. The language retains a single abstraction, the class, for module definition. All classes are considered to be mixins, and are always combined by means of the composition operator. Ordinary classes need not declare a formal parameter. Nonetheless, they are viewed as shorthand for degenerate mixins that do not make use of their parent parameter. Mixins thereby generalize *Smalltalk* classes, *Beta* patterns and *CLOS* style mixins. A mixin is *complete* if it does not refer to its parent parameter, and defines all fields that it refers to in itself. Otherwise, it is *partial*. Only complete mixins may be instantiated meaningfully. This can easily be enforced by the type system. This approach was first presented in [7].

The advantages of uniformity are:

- It makes the language simpler.
- As shown above, this simplifies the expression of useful classes.

- It allows inheritance to viewed as an *operator* over a *uniform* space of values (mixins). This represents a radical shift in thinking about inheritance. Instead of viewing inheritance operationally in terms of graphs and algorithms for traversing them, inheritance is thought of in a declarative way. These observations will be exploited in the next chapter, to develop a more comprehensive yet simpler solution.

3.3.1 Extending Existing Languages

Mixin-based inheritance can be incorporated in a natural way into programming languages that employ a linear inheritance scheme. These include single inheritance languages such as *Beta, Smalltalk* or *Modula-3* [7]. It also includes languages such as *CLOS*, which use linear multiple inheritance. *CLOS* is a particularly attractive candidate for experimentation, because it incorporates a *meta-object protocol (MOP)*[37] that was specifically designed to allow for easy language modification. In fact, a *CLOS* implementation of mixin-based inheritance was seriously considered as part of this work. It was rejected because a compiler that actually implemented the MOP was not available.

3.4 Limitations

While mixin-based inheritance offers significant improvement over other linear inheritance schemes (both single inheritance and linearized multiple inheritance), it still inherits the fundamental limitations of the linear approach. There is only one way to resolve name conflicts - placing the mixins with the conflicting attributes in a certain order. This leads to three main problems:

1. There is no allowance for selectively choosing attributes from various mixins.

2. No means is provided for resolving incidental name conflicts.

3. No warning is given about conflicts - they are resolved automatically.

There are various other refinements missing from the presentation so far. Notions of information hiding have not been discussed. The ability to distinguish between static and dynamic binding of methods is absent. One could elaborate the mixin construct to support these last two, with a corresponding loss of simplicity.

Rather than extending the concept of mixin, it will be advantageous to simplify it. I have chosen to focus on the idea that inheritance is an operation over a uniform space of values, as discussed in section 3.3. This idea lends itself to a clean extension that deals with all of the problems mentioned here. The next chapter explores that approach, which is at the heart of this dissertation.

CHAPTER 4

JIGSAW

> 1. jig.saw n : a machine saw with a narrow vertically reciprocating blade for cutting curved and irregular lines or ornamental patterns in openwork 2. jigsaw vt 1: to cut or form by or as if by a jigsaw 2: to arrange or place in an intricate or interlocking way
> *Unix Webster online dictionary.*

This chapter argues that inheritance, properly formulated, is a powerful modularity mechanism that can constitute the basis of a module manipulation language.

The formulation of inheritance presented herein is derived by observing that in languages supporting multiple inheritance (e.g., [23, 51, 62]), classes are burdened with too many roles. The class construct is "large" and monolithic. Here classes are simplified, and their functionality is partitioned among separate operators.

Classes are reduced to a simple notion of module - a mutually recursive scope. These modules form a uniform space of values upon which operators act. The operators accept modules as arguments, and produce modules as results. The notion of module with its associated operations can thus be viewed as an abstract datatype.

The set of operators presented supports encapsulation, multiple inheritance, mixins and strong typing in a single, cohesive language. These features have not been successfully combined before.

Apart from the obvious relevance to object-oriented programming languages, the *Jigsaw* framework can be used to introduce modularity into a variety of languages, regardless of whether they support first class objects.

The approach is itself modular. Language designers can use this approach, and add, remove or replace operators. This makes the benefits of extensibility and modifiability associated with object-oriented programming available at the language design level.

These points are demonstrated via the module manipulation language *Jigsaw*. For concreteness, assume that *Jigsaw* manipulates modules written in an applicative language with a type system based upon bounded universal quantification [14]. However, the discussion remains virtually unchanged if modules are written in another language. For instance, although a subtype relation is assumed, its particulars are not relied upon. Hence the approach applies to languages without subtyping as well. These have type equivalence as a degenerate subtyping relation.

The remainder of the chapter is structured as follows. Section 4.1 discusses the many roles played by classes in object oriented languages. Section 4.2 then demonstrates how each of these roles is supported by *Jigsaw*'s operators. *Jigsaw* allows arbitrary nesting of modules, and this is the subject of section 4.3. A *Jigsaw* interpreter is sketched in section 4.4. This is followed by section 4.5, which shows how *Jigsaw* can be applied to a variety of languages, and why *Jigsaw* can justifiably be considered a framework in the sense used in the object-oriented programming community, as mentioned in Chapter 1.

4.1 Roles of a Class

In a language supporting multiple inheritance, the class construct typically supports a large subset of the following functions:

1. Defining a module.

2. Constructing instances of a module definition.

3. Combining several classes together. This is characteristic of multiple inheritance.

4. Modifying a class. This function is characteristic of all inheritance systems, single or multiple.

5. Resolving name conflicts among class attributes. This can be done in various ways, by *renaming* or by *explicitly specifying* the desired attribute.

6. Defining sharing constraints among classes. When classes are combined, certain attributes or groups of attributes may exist in several of the classes being combined. The question is whether these attributes should be duplicated for each participant class, or shared. Too often, the language designer has decided on a particular answer. In fact, different applications have different needs in this respect, and programmers should be able to make the choice.

7. Restricting modifiability. Usually, all visible attributes of a module are subject to modification. It is sometimes desirable to restrict this flexibility, and state that a certain attribute may not be modified by inheritance. This is useful both from a design point of view, and also for optimization.

8. Determining attribute visibility. Different mechanisms may be available, to determine visibility to users, heirs or "friends."

9. Accessing overridden attributes. It is common that a method in a modified class makes use, during computation, of the method it has overridden, using special notation.

In addition, if the language is strongly typed, one often finds that a class fulfills additional roles:

10. Defining a type.

11. Defining a subtyping relation.

Jigsaw separates inheritance from subtyping to preserve encapsulation, as discussed in Chapter 2.

The following section presents *Jigsaw*'s operator suite. The roles detailed above are examined in turn, and, for each role, the relevant operator(s) described.

4.2 The Jigsaw Operator Suite

4.2.1 Module Definition

The primary definitional construct in *Jigsaw* is the *module.* A module is a self-referential scope, binding names to values. A binding of name to a value is a *definition.* Unlike *ML* [44], modules do not bind names to types. Type abbreviations may be used, as syntactic sugar.[1] Typing in *Jigsaw* is purely structural.

Modules may include not only definitions, but *declarations.* A declaration gives the type of an attribute, but no value for it. Declarations are used to define "abstract classes." Modules may be nested. Every module has an associated *interface*, which gives the types (or interfaces, for nested modules) of all visible attributes of a module. The subtyping relation on interfaces is defined as interface equivalence. Two interfaces are equivalent if they have exactly the same attribute names, and the attributes have equivalent types or interfaces.

Modules have no free variables, and module operators do not require access to the source code of their operands. This allows for separate compilation, including inheriting from separately compiled modules.

4.2.2 Instantiation

A module M is instantiated by the expression **instantiate** M. The result of this expression is known as an *object* or *instance.* The module in Figure 4.1 is similar

[1] In *ML* terms, only **type** declarations, not **datatype** declarations, are supported.

```
module
{  x = 0; y = 0;
   dist =  function(aPoint:{ x:Int, y:Int })
           {
               sqrt(sqr((x - aPoint.x)) + sqr((y - aPoint.y)))
           }
} : {  define x:Int, y:Int, dist:{ x:Int, y:Int } → Real }
```

Figure 4.1. A module and its interface

to the class shown in Figure 2.2, and can be instantiated into a point object with coordinates at the origin.

In an applicative language, all instantiations of a module are identical. Then why distinguish between a module and its instance? The main reason is typing. It is extremely desirable to use instances polymorphically. On the other hand, module operations require exact knowledge of the type of their operands. Distinguishing modules from instances allows separate type rules to be given for each.

An alternative would be to introduce a new judgement into the type system, indicating that a value is exactly of some type, in addition to the ordinary judgement that a value has some type. This solution is more verbose. Also, the solution chosen here is more natural, since modules do denote a different kind of value than objects. This will be discussed in Chapter 5.

Another reason for keeping modules and instances distinct is that the decision to make module instances first class values (as in "Class-based" languages [72]) need not imply that modules themselves are first class values. If modules are identified with instances, the two decisions cannot be separated. The use of *Jigsaw* should not constrain language designers in this way. Subsection 4.5 discusses a language design where neither modules nor instances are values; Chapter 6 refers to a language where instances are values, but modules are not; in *Jigsaw*, both modules and their instances are first class values (the fourth option, making modules values while instances are not, is self-contradictory).

Of course, imperative languages based on *Jigsaw* are of great practical interest. In this case, the distinction between modules and objects is essential. Some imperative object-oriented languages provide constructors or destructors for initializing or eliminating objects. *Jigsaw* does not support such constructs. Instead, modules are expected to incorporate an initialization method that can be invoked immediately after instantiation. This solution is also advocated in *Modula-3*.

4.2.3 Combining Modules

Two modules may be combined using the **merge** operation. The result is a new module, in which all names declared in either of the inputs are declared. Name conflicts are not permitted, and result in a static error. Note that the merge operator does not provide any mechanism for resolving such conflicts. Other operators are used for this purpose. This is one example of how definitions are simplified in this approach.

Merge is commutative and associative. The **merge** operator is discussed further in the context of sharing (subsection 4.2.6).

4.2.4 Modification

One module may be modified by another. This is an asymmetric operation, in which one module overrides the other. This is supported by the **override** operation: M1 **override** M2. The **override** operator takes two modules and combines them. If an attribute is defined by both modules, then the type of the attribute in M2 must be a subtype of its type in M1. In that case, the value from M2 will appear in the result.

Override is associative and idempotent, but not commutative.

4.2.5 Name Conflict Resolution

Name conflicts can be resolved in several ways. One can explicitly choose one of the conflicting attributes in preference to all others. This eliminates the conflict, but requires that all modules share a common version of the attribute. This may not always be desired. Furthermore, the types of the conflicting attributes may be incompatible, in which case such sharing is impossible. Sharing is discussed in the following subsection.

An alternative is to eliminate the conflict by renaming. This is always possible, and all attributes remain available. The one drawback is that in a structure-based type system, attribute names are meaningful for subtyping, and renaming may adversely affect polymorphism.

The renaming operator changes the name of a single attribute:

M **rename** a **to** b

The effect is equivalent to a textual replacement of all occurrences of the attribute name a in M, by the name b. Attribute a must be declared by M, and b neither declared nor defined.

The type rule for rename must ensure that the attribute is renamed in the type of the result.

It is worth pointing out that a dedicated renaming operator is more than just a convenience. A naive interpretation of renaming would lead one to the idea that to rename a to b, it suffices to add a b method that invokes a, and then hide a. This is a valid way to define renaming for records. However, when inheritance is involved, this solution is *not* equivalent to textual substitution. When a modified version of b is introduced, the expectation is that all internal references will invoke the new b method. Since many of these references actually refer to the old name, a, they will not invoke the revised method. This behavior is different from what

would have occurred had renaming not taken place. The desired property is that **rename** distributes over **override**, and is illustrated in Figure 4.2.

4.2.6 Sharing

When modules are merged in *Jigsaw*, multiple definitions of an attribute give rise to errors. In contrast, multiple declarations of an attribute are shared, and are perfectly legal.

Of course, this is only valid as long as the declaration agrees with the definition. The definition must have a type that is a subtype of the declaration. Similarly, two declarations may clash, as long as they have a subtype in common. Existing object-oriented languages that recognize the notion of "pure virtual" do not make this distinction, and treat identically all name clashes between classes being combined. In contrast, in *Jigsaw*, declarations can help specify sharing constraints among modules being combined, at the granularity of attributes.

Sharing is facilitated by the **restrict** operator. The effect of a restrict operation is to eliminate the definition of an attribute, but retain its declaration. Unlike records, it is not generally possible to completely remove an attribute from a module, because the module may contain internal references to the attribute. **Restrict** creates an abstract class, by making an attribute "pure virtual." Therefore, abstract classes may be created "after the fact." The attribute being restricted must be defined by the argument module. The **restrict** operation is associative.

When several modules are combined via **merge**, sharing of conflicting attributes may be specified by restricting all but one. This supports conflict resolution via explicit specification, a feature that was missing in mixin-based inheritance.

(m1 **rename** a **to** b) **override** (m2 **rename** a **to** b) = (m1 **override** m2) **rename** a **to** b

Figure 4.2. **Rename** distributes over **override**

Project is a dual of **restrict.** Rather than specifying which attribute to remove, **project** specifies which attributes to retain. A module, M, and a list of attributes, A, are the inputs to the **project** operation. **Project** requires that all names in A be defined by M.

4.2.7 Restricting Modifications

The **freeze** operator accepts an attribute name, a, and a module as parameters, and produces a new module in which all references to a are statically bound.

Some languages support this using the notion of nonvirtual attributes (static binding). However, this does not allow for changing the status of a virtual attribute to nonvirtual (e.g., as in *Beta* [38]). In addition, it complicates the model, since not all methods are defined in the same way - there are two kinds, declared differently. In the *Jigsaw* model, it is preferable to have only virtual attributes declared, and perform the change by means of an operator on modules. The attribute being frozen must be defined.

Freeze has a dual operation, **freeze_all_except** M A, that freezes all features of a module M, except those specified in the list A. The attributes listed in A must be defined by M.

4.2.8 Attribute Visibility

Visibility control is implemented by means of the operations **hide** and **show.** M **hide** a eliminates a from the interface of M. The attribute a must be defined by M.

Conversely, M **show** A hides everything except the specified attributes. All attributes listed in A must be defined by M.

4.2.9 Access to Overridden Definitions

Access to overridden definitions is supported through the use of the **copy-as** operator. M **copy** a **as** b creates a copy of the a method, under the name b. The a method can now be overridden, while the old implementation remains available under the name b. M must not declare an attribute b, but must define a.

Consider Figure 4.3, which also demonstrates how *Jigsaw* emulates mixins. Recall that the intent here is that the BorderMixin module modifies the Window module by adding a border, to be displayed around the window. The new display routine first displays the window's body, and then surrounds it with a border. BorderMixin declares an unimplemented routine displayBody, which is invoked within the display routine. Before overriding Window with BorderMixin, Window's display routine is copied as displayBody.

Note that renaming display to displayBody in Window would be inappropriate. When display was modified by BorderMixin, references to display within Window would not be modified. Defining a displayBody routine that called display and

```
BorderMixin = module
   { borderWidth = 5; borderColor = red;
     display = function(dontCare: Unit)
                    {
                      displayBorder();
                      displayBody();
                    }
     displayBorder = function(dontCare: Unit) { ... }
     displayBody : Unit → Unit;
   }
Window = module
   { x = 0; y = 0;
     display = function(dontCare: Unit) { ... }
   }
BorderWindow = Window copy display as displayBody override BorderMixin;
```

Figure 4.3. Using a mixin

adding that to Window would yield an infinite recursion once the modification by BorderMixin was performed.

Another point is that BorderMixin is not technically a mixin, in the sense defined in Chapter 3. BorderMixin is not a function on classes or modules, but an ordinary module (albeit an abstract class). However, it fulfills the same purpose as a mixin, since it is a modification that stands on its own, and can reference functionality it overrides. This is done without recourse to more elaborate structure. Instead, the functionality is delivered using additional operators.

4.3 Nesting Modules

The ability to nest modules within one another was mentioned in Chapter 2 as an important requirement for modularity (criterion 3). Nesting addresses the global name space problem. The former global space is a module. It can be extended, modified, renamed, etc. Renaming means name conflicts are never an issue. Modules developed at remote sites are in their separate "global" modules. These can be merged, and conflicts resolved by sharing, hiding and renaming.

Consequently, class libraries are simply modules, with ordinary classes as nested modules. In particular, note that frameworks (as defined in Chapter 1) are class libraries designed to be extended with additional classes, many of which extend the classes defined within the framework. So the process of completing a framework can be viewed as extension of a module containing nested modules.

In principle, nested modules have many additional applications, including modifying entire class hierarchies via inheritance and use as "factories" that produce instances of nested modules while serving as shared data repositories for all these instances. Unfortunately, the limited nature of subtyping on modules restricts these solutions. Chapter 8 includes an overview of the exciting possibilities mentioned here, and what steps might be taken to support them. The only language that

currently supports unrestricted class nesting is *Beta*. Again, Chapter 8 discusses class nesting in *Beta*.

Given nested modules, running a program is simply instantiating the "top-level" module and invoking some user-written initialization method. To illustrate the use of *Jigsaw* for the purposes discussed above, a conceptual sketch of a *Jigsaw* interpreter follows.

4.4 An Interactive Jigsaw Interpreter

In this section, an outline of an interactive interpreter based on *Jigsaw* is described. The interpreter will be used to demonstrate how one would actually utilize *Jigsaw* to obtain the advantages described above.

Jigsaw defines a language for manipulating modules. *Jigsaw*'s notion of module is a mutually recursive scope, so *Jigsaw* module operators are also operators on scopes. This makes *Jigsaw* well suited to handling problems that arise in interactive language systems.

Consider the usual *ML* top level interpreter. It takes the view that every expression submitted to it is implicitly prefixed by a **let**, and followed by an **in**, creating the top level scope (known as the top level environment in *ML*).

This is appealing as it makes the interpreter behave like a language processor, according to the lexical scope rules of the language. Unfortunately, it also means that it is often impossible to interactively correct a bug. Suppose f is a function whose definition is buggy. If one realizes this and submit a new definition for f, it shadows the old one. This is consistent with the notion of lexical scoping in nested **let** expressions - the innermost definition shadows all others. However, any other function using f remains bound to the previous definition, and will not be corrected. This defeats the much of the purpose of having an interactive language processor. The only workable way to develop programs is through editing files, outside the interpreter.

One should point out that the approach taken by *ML* and others has the advantage that code can be compiled as it is submitted. Later changes will not require existing code to be adapted, by, say, linking in updated definitions of functions. Another advantage is that there is no typing constraint on the new definitions, since they are in a new scope.

LISP interpreters usually have a more intuitive behavior. A new definition will affect the execution of earlier code using it. Of course, these interpreters rely on the late binding of names to values, with a corresponding cost in performance. Typing is not a problem in these languages either, since they are typically dynamically typed.

A new definition is an extension of the existing environment, or scope. A revised definition means that the existing environment is being overridden. To shadow an existing definition, one may hide it and extend it with a new one. The concepts of extension (via **merge**) and overriding are exactly those supported by *Jigsaw*. Unlike a *LISP* interpreter, *Jigsaw* performs static typechecking, and can compile modules as they are submitted like an *ML* interpreter.

The operation of a *Jigsaw* interpreter can now be described. A *Jigsaw* language processor expects to be presented with a module expression. Such an expression is either a single module, or a series of modules composed by operators.

The *Jigsaw* interpreter would first read in the initial environment (I/O routines, system calls, standard utilities, etc.). In practice, this environment may be built in, but conceptually this makes no difference. This standard environment is a module expression. The interpreter would then expect a connecting operator, such as **merge, override, rename** and so forth, followed by a new module expression. The user thus specifies how to modify the top level environment. Each succeeding input expression continues to modify the environment in this way.

If the type of a function is being revised in an inconsistent way, the interpreter will insist that the old version be hidden. All functions referencing the old definition need to be redefined with new type information in any case, so the interpreter may eventually eliminate the old definition, when it is no longer referenced.

The interpreter must support commands for saving and retrieving modules to and from the file system. Call these commands **save** and **retrieve**. **Retrieve** is a function that takes a string specifying a filename, and returns the module expression contained in that file. A program created separately using a text editor can then be added to the existing environment in several ways. **Retrieve** filename is an expression, that can be placed wherever a module expression is expected. In this way, one can either import all the definitions in a file, using a **merge** operator:

merge retrieve filename

or, one can import them at a nested level, as in

merge module m = **retrieve** fn **end**

This latter form might be useful for importing an entire class library, which might then require renaming, etc.

A **save** command would simply have the effect of writing out the value of the current environment into a file whose name is specified. This file can then be retrieved in a later interpreter session. Similar utilities that save and retrieve compiled modules are also required.

4.5 Adding Modules to Existing Languages

Many languages do not have adequate modularity constructs. These include widely used programming languages (e.g., *C* [36], *Pascal* [32]), as well as countless special-purpose and "little-languages" [5, Column 9], where the effort of designing specific mechanisms for modularity is difficult to justify, but which could still benefit from such mechanisms.

The simple notions of module and interface defined above are largely language independent. This is because neither the value set used in definitions nor the form of the types used in declarations are specified by *Jigsaw*. One requirement is that the language being "modularized" support recursion, since modules are mutually recursive scopes. When working with a language that does not support recursion, users may accidentally create mutually recursive definitions which are in fact illegal, and not get any compile time error or warning. [2]

Suppose one wishes to define and manipulate modules consisting of statements in some programming language, L_c. The definitions in modules will bind names to denotable values of L_c. For example, if $L_c = C$, the denotable values will include C functions and variables. Declarations and module interfaces will bind names to L_c types (in fact, since modules may be nested, definitions may also bind names to modules, and declarations may bind names to interfaces). Again using C as our example, the typing rules for module operators will rely on C type equivalence as the subtyping relation $\leq$ mentioned above.

The resulting language is not object-oriented, since it does not support first class objects. Nevertheless, it employs inheritance. Inheritance supports module interconnection by combining self reference among modules, and, of course, allows existing code to be extended and modified.

A wide range of languages can be extended as described here. Many of these languages are dynamically typed. In this case, the subtyping relation is simply *true*. This restricts the degree of static interface checking possible. However, any language that is extended with *Jigsaw* style modules gains substantial benefits from encapsulation, separate compilation (for compiled languages), modifiability and the ability to define partially specified modules analogous to abstract classes.

[2] A specialized version of *Jigsaw* could be created to deal with this problem in some way, e.g., by banning cyclic references.

4.5.1 Jigsaw as a Framework

Throughout this dissertation, *Jigsaw* is often referred to as a framework. As discussed in section 1.1.2, the term *framework* has a particular meaning in the context of object-oriented programming. The purpose of this section is to explain the exact sense in which *Jigsaw* can be considered a framework.

Jigsaw defines a number of abstractions that are useful in the context of module manipulation. These abstractions include those of *module, interface* and *instance*. Of course, each of these abstractions has associated with it syntax and semantics. Both the syntax and semantics of *Jigsaw* are defined relative to other, incompletely specified abstractions such as *value, type* and even *label*, which represents the lexical form of attribute names. This second set of abstractions belongs to the computational sublanguage.

One way of reifying *Jigsaw* is to associate a class with each of the key abstractions it defines. Classes representing L_c are pure virtual classes. The result of this reification is a collection of (abstract) classes, that together form a basis for implementing a modular programming language processor. This is a framework, in the sense used in the object-oriented programming community. A particular modular programming language can be implemented, by supplying definitions for the pure virtual classes. These definitions are an implementation of the computational sublanguage. Pseudo-code for such a framework is shown in Figure 4.4. Such a framework could be coded up in *Beta*, a language that supports nested classes and name-based typing. The pseudo-code shows many of the functions that would be needed in such a framework, but does not purport to be complete. For example, many functions would also need to access a module wide symbol table, but that is not described here.

Jigsaw relates to this framework as a language definition relates to a compiler (see Figure 4.5). It is therefore a *framework specification*. This last phrase has

```
class Jigsaw_Interface

class module;

function make_module(value_bindings:  List[(label,value_or_module)]):module;
function interface_of():interface;

function merge(b: module):module;
function override(b: module):module;
function restrict(l:label):module;
function project(l:  List[label]):module;
function rename(l1,l2:label):module;
function freeze(l:label):module;
function freeze_all_except(l:  List[label]):module;
function hide(l:label):module;
function show(l:  List[label]):module;
function copy_as(l1,l2:label):module;
function instantiate():instance;

function parse_module(s: stream):module;

end; /* module

class interface;


function make_interface(bindings:  List[(label,type_or_interface)]):interface;

function interface_eq(i2: interface): boolean;
function subinterface(i2: interface): boolean;

function parse_interface(s: stream):interface;

end; /* interface
(* Continued in next figure *)
```

Figure 4.4. A framework implementing *Jigsaw*

```
class label; /* pure virtual

function label_eq(l2: label): boolean; /* pure virtual

function parse_label(s: stream):label; /* pure virtual

end; /* label

class value; /* pure virtual

function parse_value(s: stream):value; /* pure virtual

function type_of():type; /* pure virtual

end; /* value

class type; /* pure virtual

function type_eq(t2: type): boolean; /* pure virtual
function subtype(t2: type): boolean; /* pure virtual

function parse_type(s: stream):type; /* pure virtual

end; /* type

class instance;

function select(l: label):value;

end; /* instance

type value_or_module = value | module;
type type_or_interface = type | interface;

end;
```

Figure 4.4. - Continued

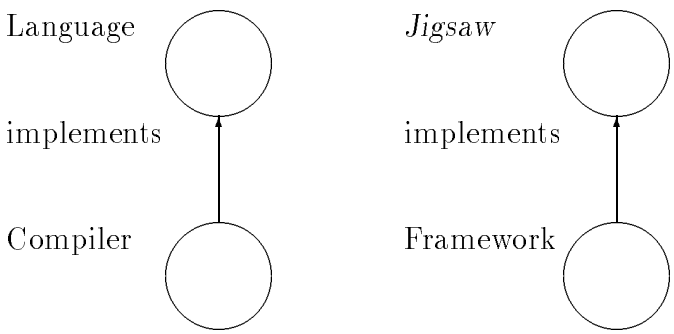

Figure 4.5. *Jigsaw* specifies implementation frameworks

a double meaning. *Jigsaw* specifies how frameworks for implementing modular languages should behave. *Jigsaw* is also a framework for specifications. Just as implementation frameworks are completed by implementations, specification frameworks are completed with specifications (Figure 4.6). Completing *Jigsaw* with specifications for L_c yields a specification for a particular modular programming language. Figure 4.7 shows all the relationships between *Jigsaw*, frameworks for implementing modular programming languages, modular programming language specifications and modular programming language processors.

It is important to realize that *Jigsaw* is indeed a framework. If *Jigsaw* was only parameterized by L_c, then it could be thought of as a function from languages to languages. After all, given a language L_c, *Jigsaw* produces a modular version of

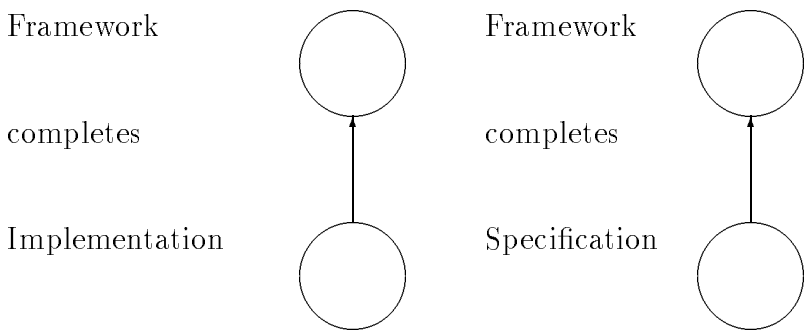

Figure 4.6. Specification frameworks are like implementation frameworks

that language. However, the relationship between a parameterized abstraction and its parameter is unidirectional. The abstraction may refer to the parameter, but not vice versa. As noted in Chapter 1, abstract classes present a bidirectional form of abstraction. This is an important characteristic of frameworks.

Obviously, *Jigsaw* depends on L_c. To see that the relationship between *Jigsaw* and L_c is bidirectional, consider a language with first class objects. In such a language, the *Jigsaw* statement **instantiate** M is available within L_c. L_c depends on *Jigsaw*'s notion of object. It is interesting to note that it is exactly when *Jigsaw* is used in an object-oriented way, as a true framework rather than just as a parameterized abstraction, that the resulting language is object-oriented.

One of the welcome properties of inheritance is that it can be applied repeatedly

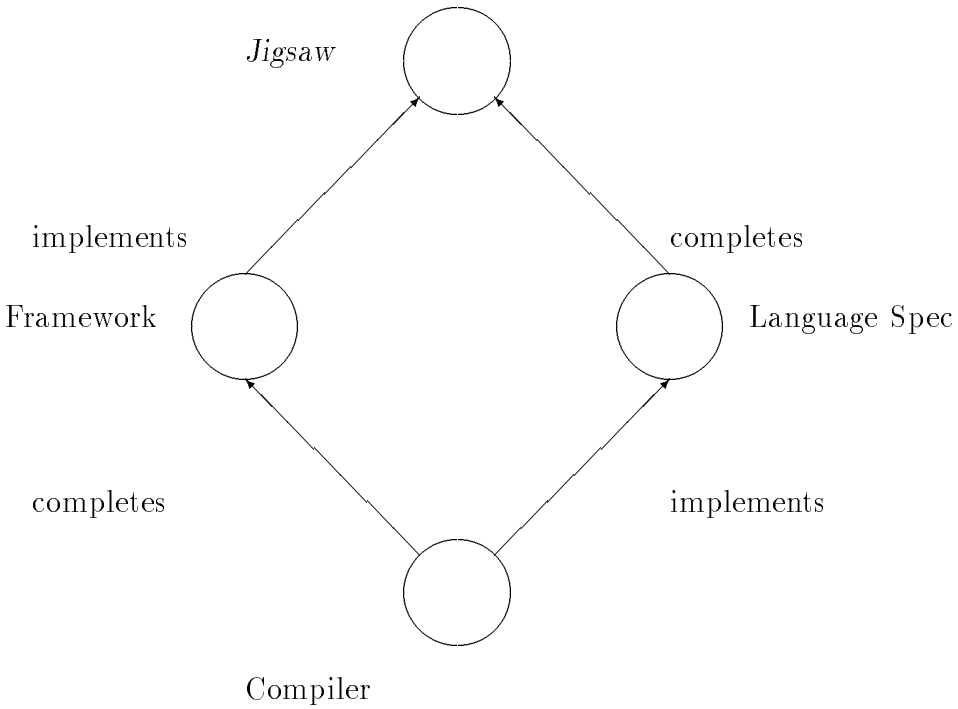

Figure 4.7. The big picture

to both complete and incomplete structures. An abstract class can be made concrete, or it may be merely extended or modified but still remain abstract. Even a concrete class can easily be modified. Similarly, a framework can be fleshed out into an application, but it can also be extended and modified without becoming a complete application. When a framework is completed, the set of classes that comprise it can still be modified further.

Jigsaw retains these properties as well. Variations on *Jigsaw* can be defined, that are not complete language specifications. When a full language specification is derived from *Jigsaw*, it is still structured so that it can be changed with relative ease.

Conceptually, different interfaces are associated with different uses of the *Jigsaw*

framework. If the framework is being extended to create a new framework with, say, additional operations on modules, the structure of modules is considered public, and can be taken advantage of when new operators are defined. On the other hand, to the language designer using a particular variant of *Jigsaw*, the module abstraction is an abstract datatype, which provides certain known operations such as **merge, override, rename**, etc.

One implementation of *Jigsaw* would be a framework like that described in Figure 4.4. Such a framework could be the basis for a family of interoperable language processors.

As noted earlier, *Jigsaw* permits type definitions in modules as a syntactic shorthand, but does not support a semantic notion of types as module components. Support for named typing is important when modularizing languages like *Pascal*, whose type system is name-based. If *Jigsaw* modules could include types in this manner, *Jigsaw* could be represented in itself. It does not seem difficult to extend *Jigsaw* in this manner. While the details of such an extension are left for future work, an outline is given in Chapter 8.

Even more valuable would be support for abstract datatypes. This is more challenging problem, again discussed in Chapter 8.

CHAPTER 5

SEMANTICS

> The formalism remains an unaccommodating object of study, without true structure, a piece of *soft camembert. Jean-Yves Girard*

This chapter presents the theoretical basis of the dissertation. The work presented here is an application of denotational models of inheritance, as pioneered by Reddy [59] and especially by Cook [18]. It also builds on the extensive body of work on typing object-oriented languages, starting with [10]. Readers unfamiliar with the details of the aforementioned work need not worry; the background section provides all the requisite information for understanding this chapter.

Those readers whose primary interest is pragmatic can skip this chapter altogether, without loss of continuity. Conversely, anyone whose main concern is theoretical may find this chapter an interesting application of theory, but should be warned again that this dissertation does not introduce new theory, but rather utilizes theory to gain insight into language design. Finally, this chapter should be of considerable interest to those computer scientists eager to translate theoretical progress into improved language design.

One of the main points of this thesis is an elucidation of the meaning and importance of the notion of *mixin*. Three different approaches to modeling mixins denotationally are presented. These approaches represent a historical succession, each progressively more uniform and simpler than its predecessor. This succession culminates in the semantic definition of *Jigsaw*.

Jigsaw is a framework for modular programming language design that abstracts over the computational sublanguage used, L_c.

Naturally, the formal definition of *Jigsaw* must also be abstract with respect to L_c. There are several places in the definition where this abstraction is evident. These include the context-free syntax, the type rules (context-sensitive syntax) and the semantics.

In the context-free syntax, certain nonterminals are not defined. These include *expr* and *type*, which represent the language L_c and its type-expression sublanguage, respectively. These must be provided by L_c.

In the type rules, some rules are left unspecified. *Jigsaw* gives rules for well-typed module values, but not for well-typing values of L_c. The situation is similar with respect to rules for subtyping, type equivalence and type formation.

In the semantics, there are semantic clauses used, that are not given by *Jigsaw*. *Jigsaw*'s semantics are defined by a translation to an untyped λ calculus augmented with basic types, records, record operators, and *let* and *where* constructs.

The interesting part of the translation is that which defines modules and the operations upon them. The translation of other parts of the language is simply that defined by L_c.

Jigsaw is a typed language that guarantees the type safe use of module operators. However, the operators are defined in an *untyped* λ calculus. A typed calculus is not used, because no known typed calculus can express all the module operators defined here in their full generality.

Section 5.1 gives a brief review of prior work on the semantics of object oriented languages. Knowledge of this work is essential for understanding the semantics of *Jigsaw*. Section 5.2 describes how to model the notion of mixin. Section 5.3 presents the semantic basis for *Jigsaw* in informal terms. A complete formal definition of *Jigsaw* follows in Section 5.4. Section 5.5 presents denotational semantics for an imperative version of *Jigsaw*.

5.1 Background

5.1.1 Generators

In object oriented programming, objects include data, and code that operates upon that data. Objects are thus inherently self-referential. The standard technique for modeling self-reference is fixpoint theory [50]. Using fixpoint theory, an object may be modeled using a record-generating function (called a *generator* following Cook [18]). Figure 5.1 shows a simple object and its associated generator function. This function takes a record as a parameter, and returns a record as a result. The result record is similar to the object being modeled. The object's methods, such as dist, are represented by function valued fields in the result. The object's data are represented by fields with ordinary values (e.g., x and y). All self-reference in the object is replaced by reference to the generator's formal parameter, s. The desired object is the least fixed point of the generator function $Y(Pgen)$.

An abstract class may be modeled as an *inconsistent generator.* An inconsistent generator has the form $\lambda s : \sigma.e$, where $e : \sigma'$ and σ is a subtype of σ' [18]. This captures the fact that self-reference within the class (σ) assumes more methods than the class provides (σ'). One cannot take the fixpoint of such a generator, since its domain is a proper subtype of its range. This models the fact that abstract classes must not be instantiated.

```
P =  object
   {  x = 0; y = 0;
      dist =  function(aPoint)
                        {
                          sqrt(sqr((x - aPoint.x)) + sqr((y - aPoint.y)))
                        }
   }
```

$$Pgen = \lambda s.\{\ x = 0, y = 0,\ dist = \lambda aPoint.\sqrt{((s.x - aPoint.x)^2 + (s.y - aPoint.y)^2)}\}$$

Figure 5.1. An object and its generator

5.1.2 Records

The record operations used in this chapter are defined in this subsection. Similar operations have been used in the study of typed record calculi [13, 27, 71, 60]. However, this dissertation is not concerned with the typing problems raised by these operators. Here, record operations are only used in the definitions of module operators. These, in turn, are used only when the types of their operands (i.e., modules) are exactly known, so that type safety is easily guaranteed.

The meaning of each record operator is explained informally in this section. It is not difficult to encode these operators in λ calculus; first, records are taken to be a syntactic shorthand for functions from a domain of labels to the domain of values. Using this representation for records, all operators used here are also easily expressed. Records are not self-referential. Other mechanisms such as generators are needed to induce self-reference. See [10] for more details.

Each record operator has a corresponding operator for generators (see section 5.3). Related operators are distinguished by subscripts (e.g., θ_r is a record operator, and θ_g is a generator operator).[1] Records are denoted by r, r_1, r_2, names of record attributes by a, b, and lists of attribute names by A.

The operators used here are:

- *Merge*, $\|_r$. $r_1 \|_r r_2$ yields the concatenation of r_1 and r_2. The records must not have any attribute names in common.

- *Restrict*, $\backslash_r$. $r\backslash_r a$ removes the attribute named a from r. If a is not defined in r, $r\backslash_r a = r$.

- *Project*, π_r. $r\pi_r A$ projects the record r on the names A. The use of the word *project* is by analogy with relational algebra. The result of this operation

[1] For a record operator θ_r, θ_g is usually what is referred to in [18] as the distributed version of θ.

consists exclusively of the fields named in A. The names in A must be defined in r.

- *Select*, $._r$. $r._r a$ returns the value of the attribute named a in r. The name a must be defined in r.

- *Override*, $\leftarrow_r$. $r_1 \leftarrow_r r_2$ produces a result that has all the attributes of r_1 and r_2. If r_1 and r_2 have names in common, the result takes its value for the common names from r_2.

- *Rename*, $[_ \leftarrow _]_r$. $r[a \leftarrow b]_r$ renames the attribute named a to b. a must be defined in r, and b must not.

The syntax used here for records is a list of label bindings enclosed in braces:

$record$ ::=	$\{def_list\}$ |
	$record_1 \parallel_r record_2$ |
	$record \backslash_r label$ |
	$record\ \pi_r\ label_list$ |
	$record\ ._r\ label$ |
	$record_1 \leftarrow_r record_2$ |
	$record[label_1 \leftarrow label_2]$
$label_list$ ::=	$label$ |
	$label\ label_list$
def_list ::=	$nonempty_def_list$ |
	$empty$
$nonempty_def_list$ ::=	def |
	$def, nonempty_def_list$
def ::=	$label = expr$

5.1.3 Inheritance

This subsection discusses the denotational semantics of inheritance [34, 59, 18]. Inheritance provides a way of modifying self-referential structures [18]. When a value is modified via inheritance, all self-reference within the result refers to the modified value. Inheritance involves manipulating the self-reference within objects. Technically, this is achieved by manipulating generators, *before* taking their fixpoint

[59], [18]. Figure 5.2 illustrates this process. The object MP inherits from P, but specializes the dist method. MP is modeled by a generator that invokes the generator for P. This invocation yields a record, that is combined using the override operation with another record which represents the specialized or new methods. In the modifying record, self-reference is modeled in the usual way, by reference to the generator's parameter. P's generator is passed this parameter as well, thereby binding self-reference in all methods to the modified object.

Modeling practical object-oriented languages has required slightly more involved constructs. The main reason is to allow access to prior definitions, as discussed in subsection 2.3.3. In that case, modifications are modeled as *wrappers*, of the form:

$$\lambda self.\lambda super.\{\ldots\}$$

The parameter *super* is a record representing the attributes of the ancestor being inherited from. The wrapper will produce a record representing the new attributes being added during inheritance. Inheritance is then an operation on a generator and a wrapper, yielding a new generator:

$$C \leftarrow_{g,w} W = \lambda s.(Cs) \leftarrow_r (Ws(Cs))$$

The expression (Cs) represents the subobject corresponding to the parent class. This is passed to the wrapper (along with s, representing self-reference). The record

```
MP = P  override
   {  dist =  function(aPoint)
                    {
                      (x - aPoint.x) + (y - aPoint.y)
                    }
   }
```

$$MPgen = \lambda s.Pgen(s) \leftarrow_r \{dist = \lambda aPoint.(s.x - aPoint.x) + (s.y - aPoint.y)\}$$

Figure 5.2. A manhattan point inherits from a point

corresponding to the new attributes is returned by the wrapper application, and added to original by the $\leftarrow_r$ operation.

As with all the formulations presented in this chapter, care must be taken when extending this formulation to imperative languages, so that the instance variables of the superclass are only allocated once. In the applicative case, repeated applications of the generator C are equivalent, but in imperative extensions, a **let** must be used to express the sharing of instance variables.

The following section shows that the semantic notion of wrapper corresponds to the linguistic notion of mixin.

5.2 Modeling Mixins

This section shows how to model a language with mixins, as discussed in Chapter 3. Two approaches are presented. Subsection 5.2.1 discusses a direct generalization of [18]. Subsection 5.2.2 gives an alternative approach in which mixin composition is viewed as a special kind of function composition.

5.2.1 A Mixin Composition Operator

Mixins can be modeled as special functions called *wrappers*. Mixin composition can be modeled as a binary operation on wrappers, $\leftarrow_w$, which returns a wrapper as its result:

$$M_1 \leftarrow_w M_2 = \lambda s.\lambda j.M_1(s)(j) \leftarrow_r M_2(s)(j \leftarrow_r M_1(s)(j))$$

This operator is associative:

$$(M_1 \leftarrow_w M_2) \leftarrow_w M_3 =$$
$$\lambda s.\lambda j.(M_1 \leftarrow_w M_2)(s)(j) \leftarrow_r M_3(s)(j \leftarrow_r (M_1 \leftarrow_w M_2)(s)(j)) =$$
$$\lambda s.\lambda j.[M_1(s)(j) \leftarrow_r M_2(s)(j \leftarrow_r M_1(s)(j))] \leftarrow_r$$
$$M_3(s)(j \leftarrow_r [M_1(s)(j) \leftarrow_r M_2(s)(j \leftarrow_r M_1(s)(j))])$$

$$M_1 \leftarrow_w (M_2 \leftarrow_w M_3) =$$
$$\lambda s.\lambda j.M_1(s)(j) \leftarrow_r (M_2 \leftarrow_w M_3)(s)(j \leftarrow_r M_1(s)(j)) =$$
$$\lambda s.\lambda j.M_1(s)(j) \leftarrow_r [M_2(s)(j \leftarrow_r M_1(s)(j)) \leftarrow_r$$
$$M_3(s)((j \leftarrow_r M_1(s)(j)) \leftarrow_r M_2(s)(j \leftarrow_r M_1(s)(j)))]$$

Assuming that $\leftarrow_r$ is associative, then

$$(M_1 \leftarrow_w M_2) \leftarrow_w M_3 =$$
$$M_1 \leftarrow_w (M_2 \leftarrow_w M_3) =$$
$$\lambda s.\lambda j.M_1(s)(j) \leftarrow_r M_2(s)(j \leftarrow_r M_1(s)(j)) \leftarrow_r$$
$$M_3(s)(j \leftarrow_r M_1(s)(j) \leftarrow_r M_2(s)(j \leftarrow_r M_1(s)(j)))$$

This operator was first defined in [7].[2] That account deliberately omitted the discussion of self-reference, in order to simplify the presentation.

5.2.2 Mixin Composition as Function Composition

Here is another formulation of mixins, which seems more intuitive. A mixin is modeled as an abstract subclass. The superclass is explicitly abstracted over, as in

$$\lambda p.\lambda s.p \leftarrow_g \lambda s.e$$

where p is a generator, *not* a record, representing the superclass; e is a record valued expression, and $\leftarrow_g$ is the override operation on generators, defined as:

$$g_1 \leftarrow_g g_2 = \lambda s.g_1(s) \leftarrow_r g_2(s)$$

Such wrappers can be composed via ordinary function composition. If C and P are such wrappers, one has

[2] Thanks to William Cook for help in defining this operator, and in proving associativity.

$$C \circ P = \lambda g.C(P(g))$$

This formulation has the advantage of simplicity. Its relationship to the language constructs of Chapter 3 is obvious.

In this version, mixin composition is precisely function composition, so it is clearly associative. Instantiation involves passing an empty generator $BASE$ as a parameter, and then taking the fixpoint of the result. The keyword **super** is modeled as $p(s)$.

Note that this formulation is not equivalent to the previous one. In section 5.2.1, mixins were modeled as wrappers of type

$Record \rightarrow Record \rightarrow Record = Record \rightarrow Generator$

Here, wrappers have type

$Generator \rightarrow Generator$

The fact is that even this formulation is overly complex. As shown in earlier chapters, the important functionality of mixins can be expressed based on a uniform notion of module, augmented with judiciously chosen operations. The rest of this chapter is devoted to giving a precise semantic characterization of the *Jigsaw* framework.

5.3 Modeling Jigsaw

In the semantics of *Jigsaw*, all modules are modeled as generators. Module combination operators are then modeled as functions that manipulate generators, and return new generators as results. The operator definitions make use of the record operations introduced in section 5.1.2. All module operators employ the technique demonstrated above to manipulate self-reference. Modules with declarations are modeled as inconsistent generators. Module operators can take inconsistent generators as operands and may return them as results.

The rest of this section defines the set of operations upon generators used in this chapter. These are then used in the translation to λ-calculus define in subsection 5.4.3.

The merge operator, $\|_g$, is defined below. It takes two generators as parameters and produces a new generator as a result. Note that self-reference in the two generators is shared in the resulting generator, since they are both applied to same argument, s.

$$\|_g = \lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s)$$

$\|_g$ is commutative and associative. The proofs of both properties are immediate, based upon the commutativity and associativity of $\|_r$.

$\|_g$ commutes:

$m_1 \|_g m_2 =$
$(\lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s))m_1 m_2 =$
$\lambda s.m_1(s) \|_r m_2(s) = \lambda s.m_2(s) \|_r m_1(s) =$
$(\lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s))m_2 m_1 =$
$m_2 \|_g m_1$

$\|_g$ associates:

$(m_1 \|_g m_2) \|_g m_3 =$
$((\lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s))m_1 m_2) \|_g m_3 =$
$(\lambda s.m_1(s) \|_r m_2(s)) \|_g m_3 =$
$(\lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s))(\lambda s.m_1(s) \|_r m_2(s))m_3 =$
$\lambda s.(\lambda s.m_1(s) \|_r m_2(s))(s) \|_r m_3(s) =$
$\lambda s.(m_1(s) \|_r m_2(s)) \|_r m_3(s) =$
$\lambda s.m_1(s) \|_r (m_2(s) \|_r m_3(s)) =$
$\lambda s.m_1(s) \|_r (\lambda s.m_2(s) \|_r m_3(s))(s) =$
$((\lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s))m_1(\lambda s.m_2(s) \|_r m3(s)) =$
$m_1 \|_g (\lambda s.m_2(s) \|_r m3(s)) =$
$m_1 \|_g ((\lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s))m_2 m_3) =$
$m_1 \|_g (m_2 \|_g m_3)$

Override is defined as:

$$\leftarrow_g = \lambda g_1.\lambda g_2.\lambda s.g_1(s) \leftarrow_r g_2(s)$$

$\leftarrow_g$ is associative and idempotent, but not commutative. To show that $\leftarrow$ associates, substitute $\leftarrow$ for $\|$ in the associativity proof above. The idempotency of $\leftarrow_g$ also follows directly from the corresponding property of $\leftarrow_r$:

$$m \leftarrow_g m = (\lambda g_1.\lambda g_2.\lambda s.g_1(s) \leftarrow_r g_2(s))m\ m = \lambda s.m(s) \leftarrow_r m(s) = \lambda s.m(s) = m$$

$\leftarrow_g$ may also be derived from the combination of merge and restrict (defined below).

There are several alternatives for defining the renaming operator. Rather than present a single formulation, it seems valuable to discuss the various possibilities. From this, one may better understand the trade-offs and subtleties involved in formulating these operations.

Unlike other generator operations, $[a \leftarrow b]_g$ is not exactly a distributed version of $[a \leftarrow b]_r$. The result generator's argument, s, cannot be passed unchanged to the input generator g. The reason is that g expects an argument with type $\sigma \leq \{a : \alpha\}$ for some α, whereas $s : \tau \leq \{b : \alpha\}$; the value associated with a in g, is named b in s.

A first formulation of $[a \leftarrow b]_g$ might therefore be

$$\lambda g.\lambda s.g(s[b \leftarrow a]_r)[a \leftarrow b]_r$$

This assures that g gets a parameter of the desired type, with the "right" value for a. Unfortunately, this is not correct. The record renaming operator assumes that the new name is not already defined in the argument record; otherwise, a name would be doubly defined, and thus ambiguous. This is a type error. However, the generators defined here are to be used polymorphically. The argument s may very well have a defined, since it originates in a module derived later. In fact, this the

usual case for renaming. The reason for renaming an attribute is typically to avoid conflict with another attribute with the same name, prior to a merge. So if one attribute named a is being renamed, it is very likely that another attribute named a will be part of the module. Certainly this possibility cannot be precluded. If this is the case, the definition above will fail; it will pass on to g a malformed argument with multiple attributes with the same name, a.

Figure 5.3 shows the next attempt, which uses $\leftarrow_r$ instead of $[a \leftarrow b]_r$. Though this seems slightly less natural, it is necessary to avoid the problem mentioned in the preceding paragraph.

This version also raises problems. The difficulty here is related to the fact that it is possible to rename not only defined attributes, but attributes that have only been declared (pure virtuals). If a denotes a pure virtual, than the result returned by g does not include an attribute named a. The question then is whether renaming an undefined attribute is legal. This depends on the underlying record calculus. It seems quite reasonable to allow such renaming, and expect it to have no effect. However, in existing record calculi, renaming is often derived from restriction and merge. In this case, renaming a nonexisting attribute is invalid, since it implies selecting the attribute. If one wishes to inherit all the valuable results proved for such a calculus, it may be worthwhile to change the definition of $[a \leftarrow b]_g$, as shown in Figure 5.4.

It is tempting to define **rename** by composing the generator versions of restrict and merge. This is not possible, due to the presence of self-reference. The type rule for rename must ensure that the attribute is renamed in the type of the result.

$$[a \leftarrow b]_g = \lambda g.\lambda s.g(s \leftarrow_r \{a = s._r b\})[a \leftarrow b]_r$$

Figure 5.3. A valid definition of renaming.

$$[a \leftarrow b]_g = \lambda g.\lambda s.g(s \leftarrow_r \{a = s._r b\})[a \leftarrow b]_r$$

If g defines a, else

$$[a \leftarrow b]_g = \lambda g.\lambda s.g(s \leftarrow_r \{a = s._r b\})$$

Figure 5.4. An alternative definition of renaming.

A very important property of the renaming operator is that it distributes over override (see section 4.2.5), namely:

$$(m_1 \leftarrow_g m_2)[a \leftarrow b]_g = (m_1[a \leftarrow b]_g) \leftarrow_g (m_2[a \leftarrow b]_g)$$

Using definition 5.4, this is not strictly true; distributivity holds whenever all operations are type correct, but that need not always be the case. Using definition 5.3, their is no such problem. The proof, as usual, follows from the corresponding property for record operations. The distributive property holds always if rename is defined as a primitive in the record calculus, such that renaming a nonexistent attribute is allowed. If rename is a derived operation, distributivity holds whenever the expression is type correct. The upshot of all this is that users can rely on the distributivity property, since it holds in all type-correct programs.

The restrict operation is defined below, and is associative. Again, proof of this property follows trivially from the same property for the corresponding record operator.

$$\backslash_g a = \lambda g.\lambda s.g(s)\backslash_r a$$

The semantic definition for $project_g$ is

$$\pi_g\ A = \lambda g.\lambda s.g(s)\pi_r A$$

and the definition for $freeze$ is

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

This definition deserves some discussion. The result is a generator, the fixpoint of a generator generating function, $q = \lambda f.\ldots$. The generator $Y(q)$ agrees with g, with the exception of its self-reference to attribute a. Regardless of the value of s, all references to $s.a$ within the methods of $Y(q)$ are bound to $f(s).{}_r a = Y(q)(s).{}_r a$. When the fixpoint is taken again, all references to $s.{}_r a$ will be equal to $Y(Y(q)).{}_r a = Y(g).{}_r a$.

Similarly,

$$freeze_all_except\ A = \lambda g.Y(\lambda f.\lambda s.g(s \leftarrow_r f(s) \leftarrow_r (s\pi_r A)))$$

Overriding s with $f(s)$, rather than just $\{a = f(s).{}_r a\}$, means that all defined attributes are being frozen. We then override again, with $s\pi_r A$, guaranteeing that the attributes in A will indeed get their values from s, and therefore still be subject to redefinition.

Here are the definitions of $show$ and $hide$:

$$hide\ a = \lambda g.\lambda s.(freeze\ a)(g)(s)\backslash_r a$$

$$show\ A = \lambda g.\lambda s.(freeze_all_except\ A)(g)(s)\pi_r A$$

The duality between $show$ and $hide$ is apparent in the use of π_r instead of $\backslash_r$, and in the use of $freeze_all_except$ instead of $freeze$.

The definition of *copy as* is straightforward

$$copy\ a\ as\ b = \lambda g.\lambda s.let\ super = g(s)\ in\ super \parallel_r \{b = super._r a\}$$

This concludes the definitions of generator operations, which constitute the core of *Jigsaw*'s semantics. The definitions given above will be used in the full semantics, given in section 5.4.

5.4 Formal Definition of Jigsaw

5.4.1 Syntax

$mexpr$ ::=	**module** $binding_list$ **end** |
	$mexpr_1 \parallel mexpr_2$ |
	$mexpr_1 \leftarrow mexpr_2$ |
	$mexpr[label_1 \leftarrow label_2]$ |
	$mexpr \backslash label$ |
	$mexpr\ \pi\ label_list$ |
	$mexpr$ **freeze** $label$ |
	$mexpr$ **freeze_all_except** $label_list$ |
	$mexpr$ **hide** $label$ |
	$mexpr$ **show** $label_list$ |
	$mexpr$ **copy** $label_1$ **as** $label_2$
$iexpr$ ::=	**instantiate** $mexpr$
$label_list$ ::=	$label$ |
	$label\ label_list$
$binding_list$ ::=	$nonempty_binding_list$ |
	$empty$
$nonempty_binding_list$ ::=	$binding$ |
	$binding, nonempty_binding_list$
$binding$ ::=	$decl$ |
	def
$decl$ ::=	$label : type$ |
	$label : mtype$
def ::=	$label = expr$ |
	$label = mexpr$
$mtype$ ::=	{**define** $decl_list$ **declare** $decl_list$}
$itype$ ::=	{$vdecl_list$}
$decl_list$ ::=	$nonempty_decl_list$ |
	$empty$

$nonempty_decl_list ::=$	$decl \mid$
	$decl, nonempty_decl_list$
$vdecl_list ::=$	$nonempty_vdecl_list \mid$
	$empty$
$nonempty_vdecl_list ::=$	$vdecl \mid$
	$vdecl, nonempty_vdecl_list$
$vdecl ::=$	$label : type$

As noted early in this chapter, the nonterminals *expr* and *type* are not defined as part of *Jigsaw*. They must be provided by the language of computation, L_c. *Jigsaw* itself provides two kinds of expressions: *mexpr*'s, which denote modules, and *iexpr*'s which denote module instances. In some cases, L_c may define *expr* so that some derivations of *expr* lead to *iexpr*. This shows that L_c is not merely a parameter to *Jigsaw*, but that there is a bidirectional interaction, characteristic of inheritance. *Expr* refers back to *iexpr*, precisely when the resulting language is object-oriented; the language supports module instances (objects) as values. Going further, *expr* may derive *mexpr*.[3] In this case, module definitions themselves are first class values. Associated with *mexpr* and *iexpr* are *mtype* and *itype*, representing interfaces and instance types. The terminal symbol *label* is also not determined by *Jigsaw*, but by L_c, according to its lexical conventions.

5.4.2 Type Rules

In this subsection, the type rules of *Jigsaw* are given. The rules are given in a natural deduction notation. Each rule consists of antecedents, or assumptions, and a conclusion that is true provided all the antecedents are true. The antecedents and conclusion are separated by a horizontal line. The conclusion and some of the antecedents are in the form of assertions. An *assertion* has the form $\Gamma \vdash a$, and means that in *context* Γ, assertion a is provable.

[3]It does not appear useful for *expr* to derive *mexpr* but not *iexpr*. Such a language can manipulate modules, but never instantiate them.

5.4.2.1 Judgements

Judgements are generic assertions that one wants to prove within a type (or other) calculus. The judgements of *Jigsaw* are described in this section.

First, it must be clear what entities are being reasoned about. These are modules, including literal modules as well as compound module expressions. The goal of the system is to verify that a module denoted by a module expression has a valid interface. Modules have expressions (and other modules) as components. Expressions denote values of L_c. Expressions are associated with types, and the type rules of L_c determine the types of expressions. In addition to types and interfaces (the types of modules), it is useful to define the concept of a *signature*.

$$signature ::= type \mid interface$$

Since modules may contain both expressions and other modules, interfaces must contain both types and other interfaces. It is convenient to reason about them collectively, as signatures.

In addition, there are instance expressions, which have instance types. Whether instance expressions (types) are expressions (types) of L_c depends on the particular choice of L_c.

The judgements are summarized in Figure 5.5. The purpose of the rules given is to give an unambiguous description of the semantics. Those rules deemed relevant have been included, while others, necessary for formal soundness and completeness, have been omitted. The most important rules in *Jigsaw* are those for the judgements $M : I$ and $O :: \omega$. These rules are given in the following section.

A complete formalization of the *Jigsaw* type system includes rules for all the judgements mentioned in Figure 5.5. Note that several of the judgements listed are judgements of L_c. These include $e : \tau, \tau\ type$ and $\sigma \leq \tau$. These may be considered to be pure virtual judgements.

$e : \tau$	Expression e has type τ
$M : I$	Module expression M has interface I
$V : \Sigma$	Value V has signature Σ
$O :: \omega$	Object O has object signature ω
$\omega\ osig$	Object signature ω is well formed
$\Gamma\ context$	Context Γ is well formed
$\tau\ type$	Type τ is well formed
$I\ interface$	Interface I is well formed
$\Sigma\ signature$	Signature Σ is well formed
$\sigma \equiv_T \tau$	Type σ is equivalent to type τ
$I \equiv_I K$	Interface I is equivalent to interface K
$\Sigma \equiv_S T$	Signature Σ is equivalent to signature T
$\sigma \leq_T \tau$	Type σ is a subtype of τ
$I \leq_I K$	Interface I is a subinterface of K
$\Sigma \leq_S T$	Signature Σ is a subsignature of T

Figure 5.5. Judgements of *Jigsaw*.

A considerable number of rules of a purely technical nature. For example, there must be rules to allow permutation of the order of attributes in interfaces and object signatures. Other rules relate types and interfaces to signatures, as well as subtypes and subinterfaces to subsignatures. Essentially these rules state that a type is a signature, an interface is a signature and the signature equivalence and subsignature relationships follow from the corresponding relationships on types and interfaces. Likewise, if a value has a type or interface, that type or interface is its signature. All these rules, as well as those governing well-formedness, have been omitted here.

5.4.2.2 Key Rules

The following notational conventions are used throughout this subsection. Module attributes are denoted by the letters a, b, c, d, e, f. Their signatures are $\alpha, \beta, \gamma, \delta, \epsilon$ and ϕ, respectively. Attribute values are denoted by the letter v. Attribute names and values are indexed with the letters $i, j, k, l, m, n, p, q, r, s$. Contexts are denoted by Γ. The notation $\tau \downarrow \sigma$ denotes the least common subsignature of τ and σ. For

interfaces, this is defined only when $\sigma = \tau$. For types, this depends on L_c's type system. Each rule is preceded by its name, written in *italics*.

The rule *Module* specifies how to deduce the signature of a module. One difficulty that may arise is that it may not always be possible to infer the type α_i of a value definition v_i, because of recursion or due to idiosyncracies of the type system of L_c. Assume that an explicit declaration $a_i : \alpha_i = v_i$ is given in such cases.

Module

$$\frac{\begin{array}{l}\Gamma, a_1 : \alpha_1, \ldots, a_m : \alpha_m, d_1 : \delta_1, \ldots, d_k : \delta_k \vdash \delta_1 \; signature, \\ \ldots, \\ \Gamma, a_1 : \alpha_1, \ldots, a_m : \alpha_m, d_1 : \delta_1, \ldots, d_k : \delta_k \vdash \delta_k \; signature, \\ \Gamma, a_1 : \alpha_1, \ldots, a_m : \alpha_m, d_1 : \delta_1, \ldots, d_k : \delta_k \vdash v_1 : \alpha_1, \\ \ldots, \\ \Gamma, a_1 : \alpha_1, \ldots, a_m : \alpha_m, d_1 : \delta_1, \ldots, d_k : \delta_k \vdash v_m : \alpha_m, \\ \forall i \in 1 \ldots m . \forall j \in 1 \ldots m . i \neq j \Rightarrow a_i \neq a_j, \\ \forall i \in 1 \ldots k . \forall j \in 1 \ldots k . i \neq j \Rightarrow d_i \neq d_j, \\ \forall i \in 1 \ldots m . \forall j \in 1 \ldots k . a_i \neq d_j\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{module} \\ \qquad a_1 = v_1, \ldots, a_m = v_m, \\ \qquad d_1 : \delta_1, \ldots, d_k : \delta_k \\ \quad \textbf{end} : \{\textbf{define} \\ \qquad\quad a_1 : \alpha_1, \ldots, a_m : \alpha_m \\ \qquad \textbf{declare} \\ \qquad\quad d_1 : \delta_1, \ldots, d_k : \delta_k\}\end{array}}$$

The next two rules, *Merge* and *Override*, are the most involved, and employ the following additional notational conventions. The attributes $a_{1i}, i \in 1 \ldots m$, are those defined only in m_1. Likewise, the attributes $a_{2i}, i \in 1 \ldots n$, are those defined only in m_2. Similarly, the attributes $d_{1i}, i \in 1 \ldots k$, are those declared only in m_1, and $d_{2i}, i \in 1 \ldots l$, are those declared only in m_2. Attributes $b_i, i \in 1 \ldots p$, are defined in m_1 and also declared in m_2, while $c_i, i \in 1 \ldots q$, are defined in m_2 and also declared in m_1. Finally, attributes $e_i, i \in 1 \ldots r$, are declared in both m_1 and m_2.

Merge

$$\begin{array}{l}
\Gamma \vdash m_1 : \{\textbf{define} \\
\qquad a_{11} : \alpha_{11}, \ldots, a_{1m} : \alpha_{1m}, \\
\qquad b_1 : \beta_{11}, \ldots, b_p : \beta_{1p} \\
\quad \textbf{declare} \\
\qquad c_1 : \gamma_{11}, \ldots, c_q : \gamma_{1q}, \\
\qquad d_{11} : \delta_{11}, \ldots, d_{1k} : \delta_{1k}, \\
\qquad e_1 : \epsilon_{11}, \ldots, e_r : \epsilon_{1r}\} \\
\\
\Gamma \vdash m_2 : \{\textbf{define} \\
\qquad a_{21} : \alpha_{21}, \ldots, a_{2n} : \alpha_{2n}, \\
\qquad c_1 : \gamma_{21}, \ldots, c_q : \gamma_{2q} \\
\quad \textbf{declare} \\
\qquad b_1 : \beta_{21}, \ldots, b_p : \beta_{2p}, \\
\qquad d_{21} : \delta_{21}, \ldots, d_{2l} : \delta_{2l}, \\
\qquad e_1 : \epsilon_{21}, \ldots, e_r : \epsilon_{2r}\}, \\
\\
\forall i \in 1 \ldots p . \Gamma \vdash \beta_{1i} \leq_S \beta_{2i}, \\
\forall i \in 1 \ldots q . \Gamma \vdash \gamma_{2i} \leq_S \gamma_{1i}, \\
\forall i \in 1 \ldots r . \Gamma \vdash \epsilon_{1i} \downarrow \epsilon_{2i}\ signature, \\
\forall i \in 1 \ldots m . \forall j \in 1 \ldots l . a_{1i} \neq d_{2j}, \\
\forall i \in 1 \ldots n . \forall j \in 1 \ldots k . a_{2i} \neq d_{1j}, \\
\forall i \in 1 \ldots k . \forall j \in 1 \ldots l . d_{1i} \neq d_{2j}, \\
\forall i \in 1 \ldots m . \forall j \in 1 \ldots n . a_{1i} \neq a_{2j} \\
\hline
\Gamma \vdash m_1 \parallel m_2 : \{\textbf{define} \\
\qquad a_{11} : \alpha_{11}, \ldots, a_{1m} : \alpha_{1m}, a_{21} : \alpha_{21}, \ldots, a_{2n} : \alpha_{2n}, \\
\qquad b_1 : \beta_{11}, \ldots, b_p : \beta_{1p}, \\
\qquad c_1 : \gamma_{21}, \ldots, c_q : \gamma_{2q} \\
\quad \textbf{declare} \\
\qquad d_{11} : \delta_{11}, \ldots, d_{1k} : \delta_{1k}, d_{21} : \delta_{21}, \ldots, d_{2l} : \delta_{2l}, \\
\qquad e_1 : \epsilon_{11} \downarrow \epsilon_{21}, \ldots, e_r : \epsilon_{1r} \downarrow \epsilon_{2r}\}
\end{array}$$

The *Override* rule uses on additional notational convention: attributes $f_i, i \in 1 \ldots s$, are those defined in both m_1 and m_2.

Override

$$
\begin{array}{l}
\Gamma \vdash m_1 : \{\textbf{define} \\
\qquad\qquad a_{11} : \alpha_{11}, \ldots, a_{1m} : \alpha_{1m}, \\
\qquad\qquad b_1 : \beta_{11}, \ldots, b_p : \beta_{1p}, \\
\qquad\qquad f_1 : \phi_{11}, \ldots, f_s : \phi_{1s} \\
\qquad\quad \textbf{declare} \\
\qquad\qquad c_1 : \gamma_{11}, \ldots, c_q : \gamma_{1q}, \\
\qquad\qquad d_{11} : \delta_{11}, \ldots, d_{1k} : \delta_{1k}, \\
\qquad\qquad e_1 : \epsilon_{11}, \ldots, e_r : \epsilon_{1r}\}, \\
\Gamma \vdash m_2 : \{\textbf{define} \\
\qquad\qquad a_{21} : \alpha_{21}, \ldots, a_{2n} : \alpha_{2n}, \\
\qquad\qquad c_1 : \gamma_{21}, \ldots, c_q : \gamma_{2q}, \\
\qquad\qquad f_1 : \phi_{21}, \ldots, f_s : \phi_{2s} \\
\qquad\quad \textbf{declare} \\
\qquad\qquad b_1 : \beta_{21}, \ldots, b_p : \beta_{2p}, \\
\qquad\qquad d_{21} : \delta_{21}, \ldots, d_{2l} : \delta_{2l}, \\
\qquad\qquad e_1 : \epsilon_{21}, \ldots, e_r : \epsilon_{2r}\}, \\
\\
\forall i \in 1 \ldots p . \Gamma \vdash \beta_{1i} \leq_S \beta_{2i}, \\
\forall i \in 1 \ldots q . \Gamma \vdash \gamma_{2i} \leq_S \gamma_{1i}, \\
\forall i \in 1 \ldots r . \Gamma \vdash \epsilon_{1i} \downarrow \epsilon_{2i} \; signature, \\
\forall i \in 1 \ldots s . \Gamma \vdash \phi_{2i} \leq \phi_{1i}, \\
\forall i \in 1 \ldots m . \forall j \in 1 \ldots l . a_{1i} \neq d_{2j}, \\
\forall i \in 1 \ldots n . \forall j \in 1 \ldots k . a_{2i} \neq d_{1j}, \\
\forall i \in 1 \ldots k . \forall j \in 1 \ldots l . d_{1i} \neq d_{2j}, \\
\forall i \in 1 \ldots m . \forall j \in 1 \ldots n . a_{1i} \neq a_{2j} \\
\hline
\Gamma \vdash m_1 \leftarrow m_2 : \{\textbf{define} \\
\qquad\qquad a_{11} : \alpha_{11}, \ldots, a_{1m} : \alpha_{1m}, a_{21} : \alpha_{21}, \ldots, a_{2n} : \alpha_{2n}, \\
\qquad\qquad b_1 : \beta_{11}, \ldots, b_p : \beta_{1p}, \\
\qquad\qquad c_1 : \gamma_{21}, \ldots, c_q : \gamma_{2q}, \\
\qquad\qquad f_1 : \phi_{21}, \ldots, f_s : \phi_{2s} \\
\qquad\quad \textbf{declare} \\
\qquad\qquad d_{11} : \delta_{11}, \ldots, d_{1k} : \delta_{1k}, d_{21} : \delta_{21}, \ldots, d_{2l} : \delta_{2l}, \\
\qquad\qquad e_1 : \epsilon_{11} \downarrow \epsilon_{21}, \ldots, e_r : \epsilon_{1r} \downarrow \epsilon_{2r}\}
\end{array}
$$

Rename Def

$$\frac{\begin{array}{l}\Gamma \vdash m : \{\textbf{define} \\ \qquad a_1 : \alpha_1, \ldots, a_m : \alpha_m \\ \quad \textbf{declare} \\ \qquad d_1 : \delta_1, \ldots, d_k : \delta_k\}, \\ \forall i \in 1 \ldots k.b \neq d_i, \\ \forall i \in 1 \ldots m.b \neq a_i\end{array}}{\begin{array}{l}\Gamma \vdash m[a_m \leftarrow b] : \{\textbf{define} \\ \qquad a_1 : \alpha_1, \ldots, b : \alpha_m \\ \quad \textbf{declare} \\ \qquad d_1 : \delta_1, \ldots, d_k : \delta_k\}\end{array}}$$

Rename Decl

$$\frac{\begin{array}{l}\Gamma \vdash m : \{\textbf{define} \\ \qquad a_1 : \alpha_1, \ldots, a_m : \alpha_m \\ \quad \textbf{declare} \\ \qquad d_1 : \delta_1, \ldots, d_k : \delta_k\}, \\ \forall i \in 1 \ldots k.e \neq d_i, \\ \forall i \in 1 \ldots m.e \neq a_i\end{array}}{\begin{array}{l}\Gamma \vdash m[d_k \leftarrow e] : \{\textbf{define} \\ \qquad a_1 : \alpha_1, \ldots, a_m : \alpha_m \\ \quad \textbf{declare} \\ \qquad d_1 : \delta_1, \ldots, e : \delta_k\}\end{array}}$$

Restrict

$$\frac{\Gamma \vdash m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}{\Gamma \vdash m \backslash a_m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_{m-1} : \alpha_{m-1}\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k, a_m : \alpha_m\}}$$

Project

$$\frac{\Gamma \vdash m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m, b_1 : \beta_1, \ldots, b_n : \beta_n\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}{\Gamma \vdash m\pi(b_1, \ldots, b_n) : \{\textbf{define}\ b_1 : \beta_1, \ldots, b_n : \beta_n\ \textbf{declare}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m, d_1 : \delta_1, \ldots, d_k : \delta_k\}}$$

Freeze

$$\frac{\Gamma \vdash m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}{\Gamma \vdash m\ \textbf{freeze}\ a_m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}$$

Freeze_all_except

$$\frac{\Gamma \vdash m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m, b_1 : \beta_1, \ldots, b_n : \beta_n\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}{\Gamma \vdash m\ \textbf{freeze_all_except}\ (b_1, \ldots, b_n) : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m, b_1 : \beta_1, \ldots, b_n : \beta_n\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}$$

Hide

$$\frac{\Gamma \vdash m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}{\Gamma \vdash m\ \textbf{hide}\ a_m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_{m-1} : \alpha_{m-1}\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}$$

Show

$$\frac{\Gamma \vdash m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m, b_1 : \beta_1, \ldots, b_n : \beta_n\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}{\Gamma \vdash m\ \textbf{show}\ (b_1, \ldots, b_n) : \{\textbf{define}\ b_1 : \beta_1, \ldots, b_n : \beta_n\ \textbf{declare}\ d_1 : \delta_1, \ldots, d_k : \delta_k\}}$$

Copy as

$$\frac{\begin{array}{l}\Gamma \vdash m : \{\textbf{define} \\ \qquad a_1 : \alpha_1, \ldots, a_m : \alpha_m \\ \quad \textbf{declare} \\ \qquad d_1 : \delta_1, \ldots, d_k : \delta_k\}, \\ \forall i \in 1 \ldots k . b \neq d_i, \\ \forall i \in 1 \ldots m . b \neq a_i\end{array}}{\begin{array}{l}\Gamma \vdash m\ \textbf{copy}\ a_m\ \textbf{as}\ b : \{\textbf{define} \\ \qquad\qquad a_1 : \alpha_1, \ldots, a_m : \alpha_m, b : \alpha_m \\ \qquad\quad \textbf{declare} \\ \qquad\qquad d_1 : \delta_1, \ldots, d_k : \delta_k\}\end{array}}$$

Instantiate

$$\frac{\begin{array}{l}\Gamma \vdash m : \{\textbf{define}\ a_1 : \alpha_1, \ldots, a_m : \alpha_m\} \\ \Gamma \vdash \alpha_1\ type, \ldots, \Gamma \vdash \alpha_m\ type\end{array}}{\Gamma \vdash \textbf{instantiate}\ m :: \{a_1 : \alpha_1, \ldots, a_m : \alpha_m\}}$$

5.4.3 Translation to λ calculus

The meaning of *Jigsaw* expressions can now be defined by a translation function $\mathcal{T}$ that translates *Jigsaw* syntax into λ-calculus.

$$\mathcal{T}[\![\textbf{module}\ binding_list\ \textbf{end}]\!] = \lambda s.\mathcal{T}[\![binding_list]\!] \tag{5.1}$$

$$\mathcal{T}[\![m_1 \parallel m_2]\!] = \|_g\ \mathcal{T}[\![m_1]\!]\mathcal{T}[\![m_2]\!] \tag{5.2}$$

$$\mathcal{T}[\![m_1 \leftarrow m_2]\!] = \leftarrow_g\ \mathcal{T}[\![m_1]\!]\mathcal{T}[\![m_2]\!] \tag{5.3}$$

$$\mathcal{T}[\![m[a \leftarrow b]]\!] = [a \leftarrow b]_g \mathcal{T}[\![m]\!] \tag{5.4}$$

$$\mathcal{T}[\![m \backslash a]\!] = (\backslash_g a)\mathcal{T}[\![m]\!] \tag{5.5}$$

$$\mathcal{T}[\![m \pi A]\!] = (\pi_g A)\mathcal{T}[\![m]\!] \tag{5.6}$$

$$\mathcal{T}[\![m \textbf{ freeze } a]\!] = (freeze\ a)\mathcal{T}[\![m]\!] \tag{5.7}$$

$$\mathcal{T}[\![m \textbf{ freeze_all_except } A]\!] = (freeze_all_except\ A)\mathcal{T}[\![m]\!] \tag{5.8}$$

$$\mathcal{T}[\![m \textbf{ hide } a]\!] = (hide\ a)\mathcal{T}[\![m]\!] \tag{5.9}$$

$$\mathcal{T}[\![m \textbf{ show } A]\!] = (show\ A)\mathcal{T}[\![m]\!] \tag{5.10}$$

$$\mathcal{T}[\![m \textbf{ copy } a \textbf{ as } b]\!] = (copy\ a\ as\ b)\mathcal{T}[\![m]\!] \tag{5.11}$$

$$\mathcal{T}[\![\textbf{instantiate } m]\!] = Y(\mathcal{T}[\![m]\!]) \tag{5.12}$$

$$\mathcal{T}[\![decl, binding_list]\!] = \mathcal{T}[\![binding_list]\!] \tag{5.13}$$

$$\mathcal{T}[\![def, binding_list]\!] = \mathcal{T}[\![def]\!] \parallel_r \mathcal{T}[\![binding_list]\!] \tag{5.14}$$

$$\mathcal{T}[\![label = expr]\!] = \{label = \mathcal{T}[\![expr]\!]\} \tag{5.15}$$

$$\mathcal{T}[\![label = mexpr]\!] = \{label = \mathcal{T}[\![mexpr]\!]\} \tag{5.16}$$

$$\mathcal{T}[\![expr]\!] = \mathcal{T}_{L_c}[\![expr]\!] \tag{5.17}$$

The "pure virtual" function $\mathcal{T}_{L_c}$ defines the meaning of L_c expressions by translating them into λ-calculus as well.

5.5 An Imperative Jigsaw

This section shows how to modify *Jigsaw* to support imperative computational sublanguages. One cannot just "plug in" an imperative language as L_c. The module operators defined in section 5.3 are no longer sufficient. However, entirely analogous definitions that are cognizant of imperative constructs, can be substituted for the applicative module operator definitions.

In [29], Andreas Hense showed how the applicative semantics of inheritance given by Cook [18] could be extended to model imperative object oriented languages. The key problem is how to formulate the domains. In particular, the domains of generators must somehow model the fact that instantiation effects the store. The semantics given below use the solution proposed in [29], applied to the operator based formulation of *Jigsaw*.

The basic intuition behind these semantics is that when the fixpoint of a generator is taken, the result is a constructor function that may be invoked upon a particular store, to create an instance. This insight is due to Hense. Used in the context of *Jigsaw* rather than in that of a more conventional object-oriented language, the semantics become simpler, due to the uniformity of the *Jigsaw* approach.

5.5.1 Denotational Semantics of Imperative *Jigsaw*

5.5.2 Syntactic Domains

Id	I	Identifiers
Bl	B	Binding lists
Mdl	M	Module expressions
$Syntax_{L_c}$	L	Syntax of L_c denotable values
$Val = Mdl + Syntax_{L_c}$	V	Denotable values
Uop	U	Unary operators
Bop	D	Binary operators
Typ	T	Types and Interfaces

5.5.3 Semantic Domains

Dv_{L_c}		L_c Denotable values
Loc	l	Locations
Sv		Storable values
$Dv = Dv_{L_c} + Object + Loc + Generator$	v	Denotable values
$Env = Id \rightarrow Dv$	r	Environments
$Object = Env$		Objects
$Store = Loc \rightarrow Sv$	s	Stores
$Constructor = Store \rightarrow Object \times Store$	c	Object constructors
$Generator = Constructor \rightarrow Constructor$	m	Classes or Modules
$UnaryOp = Generator \rightarrow Generator$	u	Unary operators
$BinOp = Generator \rightarrow Generator \rightarrow Generator$	d	Binary operators (curried)

5.5.4 Semantic Functions

The definitions of semantic functions make use of the auxiliary function new : $Store \rightarrow Loc \times Store$. This function allocates a new location in the store.

$$\mathcal{M}_{L_c} : Syntax_{L_c} \rightarrow Env \rightarrow Dv_{L_c}$$

$$\mathcal{B} : Bl \rightarrow Env \rightarrow Store \rightarrow Env \times Store$$

$$\mathcal{B}[\![v : T]\!]\, r \quad s \;=\; (\{\}, s)$$

$$\mathcal{B}[\![v \;:=\; V]\!]\, r \quad s \;=\;$$
$$let\ (l, s') \;=\; new\ s\ (\ \mathcal{V}[\![V]\!]\ r) \quad in$$
$$(\{v \;=\; l\ \},\ s')$$

$$\mathcal{B}[\![v \;=\; V]\!]\, r \quad s \;=\; (\{\ v \;=\; \mathcal{V}[\![V]\!]\ r\ \},\ s)$$

$$\mathcal{B}[\![B_1, B_2]\!]\, r \quad s \;=\;$$
$$let \quad (r_1, s_1) \;=\; \mathcal{B}[\![B_1]\!]\ r \quad s \quad in$$
$$let \quad (r_2, s_2) \;=\; (\ \mathcal{B}[\![B_2]\!]\ r \quad s_1) \quad in$$
$$(r_1 \ \|_r\ r_2,\ s_2)$$

$$\mathcal{V} : Val \rightarrow Env \rightarrow Dv$$

$$\mathcal{V}[\![L]\!] \quad r \;=\; \mathcal{M}_{L_c}[\![L]\!] \quad r$$

$$\mathcal{V}[\![M]\!] \quad r \;=\; \mathcal{M}[\![M]\!] \quad r$$

$$\mathcal{M} : Mdl \rightarrow Env \rightarrow Generator$$

$$\mathcal{M}[\![\textbf{module} \quad B \quad \textbf{end}]\!]\, r \;=$$
$$\lambda\ c_{self}.\ \lambda\ s_{create}.$$
$$let \quad (r_{self}, _) \;=\; c_{self}\ s_{create} \quad in$$
$$\mathcal{B}[\![B]\!]\ (r \ \leftarrow_r\ r_{self})\ s_{create}$$

$$\mathcal{M}[\![M_1 \quad D \quad M_2\ ]\!]\, r \;=$$
$$let \quad m_1 \;=\; \mathcal{M}[\![M_1]\!]\ r \quad in$$
$$let \quad m_2 \;=\; \mathcal{M}[\![M_2]\!]\ r \quad in$$
$$\mathcal{D}[\![D]\!] \quad m_1 \quad m_2$$

$$\mathcal{M}[\![M \quad U]\!]\, r \;=$$
$$let \quad m \;=\; \mathcal{M}[\![M]\!]\ r \quad in$$
$$\mathcal{U}[\![U]\!]\ m$$

$$\mathcal{D} : Bop \rightarrow BinOp$$

$$\mathcal{D}[\![D]\!] \;=$$
$$\lambda\ m_1.\ \lambda\ m_2.\ \lambda\ c_{self}.\ \lambda\ s_{create}.$$
$$let \quad (r_1, s_1) \;=\; m_1 \quad c_{self} \quad s_{create} \quad in$$
$$let \quad (r_2, s_2) \;=\; m_2 \quad c_{self} \quad s_1 \quad in$$

$$(r_1\ D_r\ r_2,\ s_2)$$

$$\mathcal{U} : Uop \rightarrow UnaryOp$$

$$\begin{array}{l}\mathcal{U}[\![U]\!] = \\ \quad \lambda\ m.\ \lambda\ c_{self}.\ \lambda\ s_{create}. \\ \quad\quad let\quad (r, s) = m\quad c_{self}\quad s_{create}\quad in \\ \quad\quad\quad (r\quad U_r,\ s)\end{array}$$

CHAPTER 6

MODULA-π

Be real, Lambdaman. This isn't a POPL conference. *Harry Hackwell*

In view of the difficulty of introducing new languages into widespread use, it is extremely valuable to be able to incorporate new linguistic developments in an evolutionary manner. Adding operators like those defined in the previous two chapters to existing languages is therefore an attractive possibility.

This chapter presents an upwardly compatible extension of *Modula-3*, incorporating most of the operators described in Chapter 4. In this extension, the operators are applied not to the modules of *Modula-3* but to its classes (known as object types).[1]

Naturally, the full flexibility of *Jigsaw* is not supported. Still, the resulting language supports strong typing, multiple inheritance and mixins in an encapsulated manner.

This design represents a particular configuration of *Jigsaw*. In this configuration, the language of computation is a general purpose, object-oriented programming language, and *Jigsaw* modules serve as classes in the computation language. Furthermore, the language incorporates certain restrictions intended to accommodate efficient implementation, without sacrificing the general principles of modularity.

Section 6.1 explains why *Modula-3*, rather than another programming language, was chosen as a candidate for extension. Section 6.2 presents a review of the salient features of *Modula-3*, for those unfamiliar with the language. Section 6.3 then discusses the extension, Modula-π.

[1] An early, less ambitious version of this work appeared in [7].

6.1 Choice of Language

Modula-3 [11] was chosen as a basis for an extension incorporating *Jigsaw* style inheritance operators. The particular form of inheritance developed below will be referred to as *operator-based inheritance*. *Modula-3* is well suited for such an extension, because

1. It supports single inheritance. Single inheritance naturally generalizes either to mixin-based inheritance or to *Jigsaw* style inheritance. Languages that already provide a notion of multiple inheritance are harder to modify cleanly.

2. It is is strongly typed. Strong typing is necessary for safety, and is a desirable property in a modular language, as reflected in criterion 5 of Chapter 2. The type regime of *Modula-3* also allows for an efficient implementation. One of the goals of this extension was to show how *Jigsaw* style operations could be incorporated in a high-performance language.

3. It employs structural subtyping. *Jigsaw* already employs a form of structural subtyping, because structural subtyping is preferable to name based typing when separately developed modules must interact [55, section 8.1].

6.2 A Review of Modula-3

The purpose of this section is to present an overview of the key *Modula-3* features necessary to understand the language extension. Readers are referred to [55] for a complete language definition.

6.2.1 Modula-3 Inheritance

Modula-3 supports inheritance via object types. Object types are roughly analogous to classes in most object-oriented languages. An example of object types in *Modula-3* is given in Figure 6.1.

```
type Person =
        object name: string
        methods display() := displayPerson
        end;

type Graduate = Person
        object degree: string
        overrides display := displayGraduate
        end;

procedure displayPerson(self: Person) =
  begin
   self.name.display();
  end displayPerson;

procedure displayGraduate(self: Graduate) =
  begin
   Person.display(self);
   self.degree.display()
  end displayGraduate;
```

Figure 6.1. *Modula-3* object types

In the example, Person defines an instance variable name of type string[2] and a method display. The method is defined by providing a name, followed by a *signature*, or formal parameter list. In this case, the signature is empty. The method is then assigned a value, which is a separately defined procedure, displayPerson. If o is an object of type Person, o.display() is interpreted as displayPerson(o).

The definition of Graduate has two parts: A preexisting definition, Person, and a modification given by the **object** ...**overrides** ...**end** clause. Graduate is a *subtype* of Person, which is its *supertype*. Graduate inherits from Person, but includes a *method override* for display. The method override names the method being overridden, and then assigns a new value to it, namely displayGraduate. A signature is not given, since it will always be identical to the signature of the

[2] *Modula-3* uses **text** for character strings. However, it is assumed that string has been defined.

corresponding method in the supertype. The overridden methods of Person may be referred to by Graduate through the syntax Person.*methodname*. This is similar to **super** in Smalltalk, but more general.

6.2.2 Other Salient Features

Modula-3 programs are composed of separately compilable parts. Specifications (only syntactic) are given by *interfaces*, and implementations are defined by *modules*. It is important not to confuse these modules and interfaces with those of *Jigsaw*.

Module interconnection is by means of **import** and **export** clauses. A module may export several interfaces. Other modules may import such interfaces, establishing a connection between modules.

Data abstraction is supported by the notion of opaque types. The concepts of interface, import/export, and opaque types are well known, and need not be treated extensively here. The *Modula-3* language does introduce several newer ideas which must be understood before the language extension can be presented.

One of the relatively new constructs supported by *Modula-3* is that of a *partially opaque type*. Unlike completely opaque types, some information about the structure of a partially opaque type is publically available.

Modula-3 relies on structural typing. However, in order to support name-based typing as well, *brands* are used. A type may have a unique brand associated with it, which distinguishes the type from all other types that would otherwise be structurally equivalent.

Modula-3 distinguishes between *traced* and *untraced* references. Traced references are are automatically reclaimed by the garbage collector, while untraced references are not. This allows writing both low-level code that may be impaired by the presence of built in storage reclamation, and higher level code that benefits from garbage collection.

In some cases, the static typechecking of *Modula-3* is deemed too rigid, and so dynamic typechecks are also supported. Of special interest in this chapter is the **narrow** construct. Using **narrow**, it is possible to explicitly coerce a type into one of its supertypes or subtypes (the latter option induces a dynamic check). Narrowing also occurs implicitly during assignments and parameter passing.

Revelations are a mechanism that allows information about opaque types to be selectively distributed. Like the **friend** clauses of *C++*, revelations allow finer control over information hiding. Revelations can be *full* or *partial*, just like opaque types.

Revelations and partially opaque types are features new in *Modula-3* and their interaction with inheritance is subtle. This subtlety is what makes the extension of *Modula-3* with *Jigsaw* style operators challenging, as section 6.3.2 demonstrates.

6.2.3 Typing

As noted above, *Modula-3* employs structural typing. In this section, typing of objects is discussed. The subtyping relation is different from *Jigsaw*'s, and is more sensitive to the needs of an efficient implementation. This relation will be modified in the language extension, but its basic premises will be preserved. The rules defining *Modula-3*'s subtype relation on object types are shown in Figure 6.2.

Object types are a special kind of reference type. As described above, there are two kinds of references: traced and untraced. The type of all traced references is **refany**, while that of untraced references is **address.** Analogously, there are traced and untraced objects, which belong to types **root** and **untraced root** respectively.

root <: **refany**
untraced root <: **address**
null <: T **object** ... **end** <: T

Figure 6.2. *Modula-3* subtyping rules for object types.

All traced objects are also traced references, and likewise for untraced objects and references. The type **null** includes only the special reference **nil**, the null reference. **nil** is a member of every reference type. Finally, object types are always subtypes of their ancestors.

The subtyping relation recognizes the boundaries between **object** constructors. It makes the order of these constructors, and of the attributes within them, significant. This is different from the subtyping relation used in *Jigsaw*. Object types that support the same protocol may not be the same, restricting reusability. On the other hand, this subtyping relation makes it easy to ensure a commonality of structure among subtype representations, allowing a more efficient implementation.

6.3 Modula-π: An Extension of Modula-3

The goal of the extension was to provide as much of the power of the *Jigsaw* framework as possible, in the context of an *upwardly compatible* extension of an existing language. Such an extension must be syntactically, semantically and pragmatically upward compatible. Syntactic and semantic compatibility are widely recognized needs. In a situation like this, pragmatic compatibility is also crucial. *Modula-3* was engineered to work well in certain contexts. The extension should not violate the language's assumptions as far as performance (compilation and execution time and space) etc.

6.3.1 Object Types and their Operators

The extension is based on an analogy between object types and *Jigsaw* modules. Object type expressions will be constructed using *Jigsaw*-style operators. Object type expressions can be either primitive or compound.

The primitive constructor for object types is

object *fieldlist* **methods** *methodlist* **end**.

Such clauses are then connected by means of object-type operators. The operators are: **merge, override, restrict, project, rename, copy_as, show** and **shadow.** The operators are essentially the same as in *Jigsaw*, except for **show** and **shadow.**

6.3.2 Type Abstraction

In the context of *Modula-3*, the use of **merge** poses some difficulties. The *Jigsaw* type system was based upon the assumption that the exact signature of every module was known. Translating this assumption into terms of *Modula-π*, this means that all the fields and methods of an object type are completely known when any operator acts upon an object type. This assumption is not valid in *Modula*, due to the presence of abstract data types.

For example, given the declarations of Figure 6.3 one cannot determine whether there are any conflicts between T1 and T3. If T1 = T2 T6, and T3 = T4 T6, then the above declaration should be flagged as illegal. But if T2 and T4 do not conflict, we have no way of knowing of a conflict. In fact if T1 = T2, T3 = T4, there need not be a conflict. It could be argued that since the conflict is invisible, we can ignore it, since no ambiguity can arise. However, there may be scopes in which enough information is known for the ambiguity to surface. To see this, consider a scope which imports T5 and includes revelations for T1 and T3. It would appear that (partially) opaque object types cannot be used in object type expressions.

```
T1 <: T2
T3 <: T4

T5 = T1  merge T3
```

Figure 6.3. Difficulty with **merge** and abstract data types.

This another manifestation of the problem mentioned in Chapter 3. Inheritance in the presence of polymorphic type abstraction poses a serious challenge to static typechecking.

In *Jigsaw*, the general approach to solving problems is to introduce appropriate operators. In this case, the **show** operator is used. The main purpose of the **show** operation is to resolve problems that arise due to opacity and revelations.

A **show** B is similar to the *Jigsaw show* operation, except that the parts "hidden" by the operation are potentially observable via narrowing. The **show** operator is used to ensure that only known fields of the operands of **merge** are accessible. The type system requires that all accessible fields in both arguments to **merge** be known. This forces the user to explicitly hide any potentially conflicting fields. The example given above can be rewritten as

T5 = (T1 **show** T2) **merge** (T3 **show** T4)

Of course, if T2 and/or T4 are not completely known, this will not be sufficient, and the process may have to continue. In other scopes, the explicit use of **show** will prevent additional knowledge of potential conflicts from becoming a problem. **show** has no effect except for typing. A **show** B is well formed only when A <: B.

6.3.3 Subtyping

This section presents the typing rules for object types in *Modula-π*.

Type identity is defined as in *Modula-3*. Two types are identical iff their expanded definitions are identical. The subtyping relation on mixins, $T <: S$ (read T is a subtype of S, or S is a supertype of T) is defined in Figure 6.4. The **shadow** operation shown in the figure is discussed in the following subsection.
<: is reflexive and transitive.

```
object ... end <: root. All object types are subtypes of root.
T1 merge T2 <: T1
T1 merge T2 <: T2
T1 override T2 <: T1
T1 override T2 <: T2
T1 show T2 <: T1
T1 show T2 <: T2
T1 shadow T2 <: T1
T1 shadow T2 <: T2
T project a,b,c ... <: T
T restrict a <: T
T rename a as b <: T
T copy m as n <: T
object ... methods ... m(...) := p ... end copy m as n <:
object methods n(...) := p end
```

Figure 6.4. *Modula-π* object type subtyping

6.3.4 Compatibility

In order to obtain a language compatible with *Modula-3*, an additional operator is introduced, and some syntactic adjustments are made.

Recall that in *Modula-3*, inheritance is expressed by adjacent object-type constructors. The semantics are defined so that the modifying object-type constructor "shadows" the supertype. Fields and methods in conflict between the two are resolved in favor of the extension, but the shadowed fields and methods can be accessed by means of narrowing. Overriding of methods is by means of a special override clause.

To emulate this behavior, the **shadow** operator is introduced. A **shadow** operator is placed implicitly between every pair of adjacent object types in a type declaration.

A **shadow** B returns a type in which all fields and methods of B are accessible, as well as all fields and methods of A that do not recur in B. This implies that if A and B have fields and methods in common, their values are taken from B.

The "hidden" fields and methods are accessible via **narrow,** or via assignment and parameter passing.

The **overrides** clause of *Modula-3* is considered syntactic sugar for an **override** operator followed by a separate object-type constructor. This defines a translation from *Modula-3* syntax into the syntax of object-type expressions, and preserves syntactic and semantic compatibility.

One minor incompatibility relates to pure virtuals. An uninitialized method is considered a pure virtual, and is not initialized to **nil**. *Modula-3* does not support the notion of a pure virtual method. This change allows the definition of abstract classes, and their interconnection by means of **merge**.

It is, however, a checked runtime error to invoke such a pure virtual method. It is not an error to instantiate an abstract class. This is for compatibility with *Modula-3*. It is conceivable that some programs might instantiate abstract classes, but not invoke the **nil** methods of those classes. Such programs should continue to run under the new language. Another reason for not enforcing a policy against instantiating abstract classes is that syntax changes would be needed to detect this across modules in some cases.

6.4 Assessing Modula-π

Applying the framework to a realistic programming language teaches valuable lessons. First, support for the functionality of name-based typing is possible in the context of structural typing, using *Modula-3*'s concept of brands. Second, a way of supporting abstract data types has been developed. The technique used does not have the same formal foundation as the original *Jigsaw* framework, but it can be used in a practical setting.

Evaluating *Modula-*π against the modularity criteria of Chapter 2 shows that it is still not a completely modular programming language. Nesting of object types is not supported, and modularity operations have not been applied to the

language's modules. *Modula-π* also retains some of the deficiencies of *Modula-3*. For example, though structural typing is used, it is defined in such a way that multiple implementations of the same protocol yield distinct types. This situation is similar to that of object-oriented languages that employ name-based typing. It is tempting to forego compatibility on this point.

The measures described in section 6.3.4 make *Modula-π* syntactically and semantically compatible with *Modula-3*. The issue of pragmatic compatibility has not yet been addressed. The key requirement for pragmatic compatibility is an implementation strategy for the new language that is competitive with existing implementation techniques. The following chapter discusses such a strategy.

CHAPTER 7

IMPLEMENTATION

> Theorists need not bother: The European Common Market already has a glut of butter, milk, wine, and theorems. *Andy Tanenbaum*

A major factor in the success of any piece of software is its performance. The most elegant design may be virtually ignored unless it can be used effectively. Conversely, efficiency can compensate for almost any other weakness in a software system. The time has come to face the issue of implementation.

Most of this chapter is devoted to the presentation of a pragmatic and highly efficient implementation technique for the language *Modula-*π discussed in Chapter 6.

The implementation is efficient enough to fit into a practical programming language like *Modula-3*. *Modula-3* restricts subtyping by making it dependent on the order in which attributes are specified, and on the boundaries between constituent object types. These restrictions, coupled with the fact that *Jigsaw* modules never have any free variables, lead to an implementation based upon a straightforward extension of standard dispatch table techniques.

This dissertation presents no new techniques for implementing interface-based type systems such as *Jigsaw*'s. It has been noted in the literature [16, 31] that interface-based type systems contribute little to efficient implementation, in contrast to more traditional type systems.

Operator-based inheritance was derived from *Jigsaw* by modifying the notion of interface to reveal enough about the structure of modules for an efficient implementation. In *Jigsaw* itself, interfaces disclose no such information.

Traditionally, an implementation of a type system like *Jigsaw*'s might involve searching for an attribute at run time, and cacheing its value. Even the best such schemes are not competitive with the approaches discussed in this chapter. Recently, alternative schemes have been proposed [17]. While still not as efficient as the scheme proposed below, the gap is smaller than with cache based lookup techniques.

7.1 Implementation of Modula-π

This section describes the proposed implementation technique for operator-based inheritance in *Modula-*π.

Implementations of single inheritance languages such as *Modula-3* support the notion of virtual procedures by associating with each class a table whose entries are addresses of the class' methods. Each instance of a class contains a reference to the class method table. It is through this reference that the appropriate method to be invoked on an instance is located.

Under multiple inheritance, the above technique must be modified, since the superclasses of a class no longer share a common prefix. Offsets must be added to an object, so that the appropriate subobjects are passed to methods defined by superclasses. Since methods may be overridden, these offsets must be part of the class' method table. The offsets are different for every superclass, so a separate subtable is created for each superclass. The size of the tables is linear in the number of superclasses.

Operator-based inheritance incorporates a structural subtyping discipline. This requires that the implementation completely preserve algebraic properties of operators. Traditional multiple inheritance schemes do not do this. A further problem is that in operator-based inheritance, the number of supertypes of a type grows quadratically with the number of component types. Using the traditional approach would require quadratic table space, which is unacceptable. The solution is to

factor out the prefix offsets, which are statically known, and retain only the offsets due to method redefinition in the tables. As a result, only one table per component is created, and table space is linear in the number of components.

The following two subsections review dispatch table based implementation techniques for single and multiple inheritance, respectively. Subsection 7.1.3 discusses the basic implementation of object types and binary operators upon them. Subsection 7.1.4 discusses the treatment of pure virtuals. Subsection 7.1.5 discusses the implementation of unary operators, while subsection 7.1.6 discusses operators not included in *Modula-*π. Subsection 7.1.7 briefly discusses various other implementation issues, such as garbage collecting, dynamic type checking and the like.

7.1.1 Implementing Single Inheritance

In single inheritance, every class has a unique superclass. A class has the form $C_{new} = C_{old}\Delta$, where C_{old} is the superclass, and Δ represents the additions and changes given by the new class. An instance of C_{new} is represented by concatenating the representation of the fields added by Δ to the representation of an instance of C_{old}. It follows that every class shares a common prefix with all of its subclasses. It is therefore possible to compile code acting upon a statically known class, based on its known structure. This structure will be repeated in all subclasses, making the code reusable by the subclasses.

Virtual methods introduce a complication, since the exact method to be invoked is no longer statically known when code is compiled. The solution is to have every instance point at a method table (henceforth referred to as an *mtbl*). Each entry in the table contains the address of a method. Calls to a method become indirect calls, via a fixed entry in the mtbl. There is a table for each class. The table for C_{new} is created by copying the table for C_{old}, and appending entries for any new methods. If C_{new} overrides previously defined methods, the appropriate entries in

the table are changed accordingly. Again, the structure of a class' table is a shared prefix of the tables of all its subclasses. The size of a class' table is linear in the size of components and deltas:

$tablesize(C_0\ \Delta_1 \ldots \Delta_n) = tablesize(C_0) + \Sigma_{i=1}^{n} tablesize(\Delta_i)$.

7.1.2 Implementing Multiple Inheritance

Multiple inheritance can be implemented by a direct generalization of the technique described in the prior section. This technique has been used to implement multiple inheritance in *C++* [23], and was pioneered by Krogdahl [40]. Other approaches are possible, but I focus on this one, since it forms the basis for my implementation of operator-based inheritance.

Instances of a class that does not inherit from any other class (a *base* class), can be represented by a record of their instance variables. To support virtual methods, method tables are used, and each instance includes a pointer to a method table, as explained above.

In general, however, a class has the form $C_{new} = C_1 C_2 \ldots C_n \Delta$, where the C_i are parents of C_{new}. Instances of C_{new} are represented by concatenating the representations of instances of $C_1 \ldots C_n$ with the representation of the fields added by Δ. An instance of C_{new} is thus composed of *subobjects*, where each *sub*object corresponds to a particular *super*class. Each subobject has its own pointer to a suitable method table.

In this case, it is no longer true that a class' representation is a prefix of the representations of all of its subclasses. Each subobject begins at a different offset from the beginning of the complete C_{new} object. These offsets can be computed statically, from the definition of C_{new}. Figure 7.1 defines $Offset(A, B)$ to be the relative offset of a B subobject within an A object, if A is a subclass of B (written $A <: B$). If $B <: A$, the function gives the negative offset from the A subobject to

$Offset(C, C_i \ldots C_j) =$
$-Offset(C_i \ldots C_j, C) =$
$\Sigma_{k=1}^{i-1} Size(C_k)$ where $C = C_1 \ldots C_n, 1 \leq i \leq j \leq n$.

Figure 7.1. The *Offset* function.

the B object. *Offset* relates any two classes where one is a subclass of the other. In any other case, it is undefined.

When a method f of C_{new} invokes a method g of some superclass C_i, an offset must be added to the beginning of the object, so that g operates on the correct subobject.

Virtuals once again complicate matters. When a virtual method is redefined by a subclass, the new definition may require access to all attributes of the subclass, and not only to the attributes of the superclass that originally defined it. As a result, the offset required when invoking such a method may be changed when a class is inherited from. The offset, like the identity of the virtual function itself, is an attribute of the actual class of the object, and must be available at run time. Typically, the offset is stored in the method table, alongside the address of the virtual method. When a virtual method is invoked, the offset from the table is added to the address of the instance, before it is passed to the method.

There is one further complication. A redefined method may be invoked from the superclass C_i or from C_{new}. The offset, in each case, is different. As a result, an additional table is generated for C_{new}. This table is constructed by concatenating all the method tables of $C_j, 1 \leq j \leq n$ and the table for Δ, and resetting the offset fields as needed. Space can be saved, by realizing that C_1 does share a common prefix with C_{new}, and that Δ will not be used independently of C_{new}. These tables need not be duplicated, and are best collapsed into the table for C_{new}.

For a class $C = C_1 \ldots C_n \Delta$, the table space required is $2\Sigma_{i=1}^{n} tablesize(C_i) - tablesize(C_1) + tablesize(\Delta)$. It is therefore linear in the size of the components, and always less than twice the sum of the component sizes.

```
type A =  object a: integer  methods Ma1():= Pa1  end;
   B =  object b: integer  methods Mb1():= Pb1  end;
   C = A B;
   D =  object d: integer methods Md1():= Pd1  end;
   E = D C;
   F = D A B;
   G = D A;
   H = G B;
```

Figure 7.2. Associativity

7.1.3 Basic Implementation of Operator-based Inheritance

This section is divided into three parts. First, section 7.1.3.1 illustrates why the technique used above to implement multiple inheritance is not directly applicable to operator-based inheritance. Then, the implementation techniques for primitive and composite object types are demonstrated.

7.1.3.1 Problems with standard techniques

The fundamental difficulty in implementing operator-based inheritance stems from the interaction between structural subtyping and the algebraic properties of the **merge** and **override** operators.

Consider the class definitions of Figure 7.2. Expanding the definitions of all names (as dictated by structural typing), one finds that by associativity, E = F = H. This equivalence dictates that all three classes have the same type, so that they can be used interchangeably. This in turn requires that all three have the same representation. However, using the techniques of section 7.1.2, these three classes have different representations.

If name-based typing were used, E would be a subtype of D,C,A and B (but not G), F would be a subtype of D,A and B (but not C), and H would be a subtype of G,D,A, and B (but not C), and the classes E,F and H would not be in any subtyping relationship. In this case, E,F and H would have to behave the same under attribute

accesses (they represent equivalent values, by associativity), but that would not necessitate identical representations, since these classes are not equivalent in the eyes of the type system.

A requirement for an implementation of operator-based inheritance is that the algebraic properties of operations be preserved when the operations are performed by the compiler upon the representations of object types and objects.

The system Stroustrup uses is not associative, but can be adjusted to be so. The essence of the Krogdahl-Stroustrup approach is to generate a table for every superclass (including the class being defined), but optimize so that superclasses sharing a common prefix share the same table. With *Modula-*π's definition of subtyping, this generates a table for every suffix of $C = C_1C_2 \ldots C_n$, resulting in tables for $C_1 \ldots C_n, C_2 \ldots C_n, \ldots, C_{n-1}C_n, C_n$. The size of these tables is
$n \times tablesize(C_n) + (n-1) \times tablesize(C_{n-1}) + \ldots + 1 \times tablesize(C_1) =$
$\Sigma_{i=1}^{n}(i \times tablesize(C_i)) = \frac{n(n+1)}{2} \times \overline{tablesize(C_i)}$
where $\overline{tablesize(C_i)}$ represents the weighted average table size. This is still unacceptable. Table size for a class must be linear in the sizes of its components.

Linear table size can be achieved if tables of component classes appear only once in a compound class' table. This, in turn, requires that tables be independent of the prefix of the class. If the prefix offset is statically incorporated in every call, the table offset fields only need to include the changes introduced by overriding methods, which are relative, not absolute. The details of this technique appear in the following subsections, which give an approach to implementation based upon the following inductive definition of object types.

A object type may take one of the forms given in Figure 7.3. Object types of the first form are known as *primitive object types* since they contain no other object types as components. Object types of the other forms are called *composite object types*. Note that the first composite form is actually a shorthand for $Otype_1$ **shadow** $Otype_2$, introduced for syntactic compatibility, as discussed in the

Otype = **object** *fieldlist* **methods** *methodlist* **end** |
Otype$_1$ *Otype*$_2$ |
Otype$_1$ **merge** *Otype*$_2$ |
Otype$_1$ **override** *Otype*$_2$ |
Otype$_1$ **show** *Otype*$_2$ |
Otype **restrict** *label* |
Otype **project** *labelList* |
Otype **copy** *label*$_1$ **as** *label*$_2$ |
Otype **rename** *label*$_1$ **as** *label*$_2$.

Figure 7.3. Constructors of object types

previous chapter (section 6.3.4). The next two subsections show how to represent primitive and composite object types.

7.1.3.2 Implementing Primitive Object Types

Instances of an object type of the form

type T = **object** f_1 ... f_n **methods** m_1 ... m_k **end**

are each represented by a header word, followed by storage for the fields. The header word points to a method table. There is one mtbl for every object type. Each entry in the mtbl corresponds to one of the methods $m_i, 1 \leq i \leq k$. There are two fields per entry - the address of the procedure implementing the method, and an offset field. All address fields are set to the address of the appropriate method value. Offset fields are usually, but not always, set to zero.

If aT is an instance of class T, an invocation of $aT.m_i$ is compiled into an indirect procedure call, using a statically determined entry in the mtbl. The address of aT is added to the offset stored in the same entry in the mtbl, and passed as the first parameter to the procedure. This is the same procedure used in [23].

A simple example is the class A in Figure 7.4, whose layout is shown in Figure 7.5 (Note that the method names shown in the figure are for expository purposes only; they are not present in the physical implementation).

```
type A =  object f1:  integer  methods m1() := p1, m2() := p2  end;

procedure p1(anA:A);
procedure p2(anA:A);
```

Figure 7.4. A primitive object type

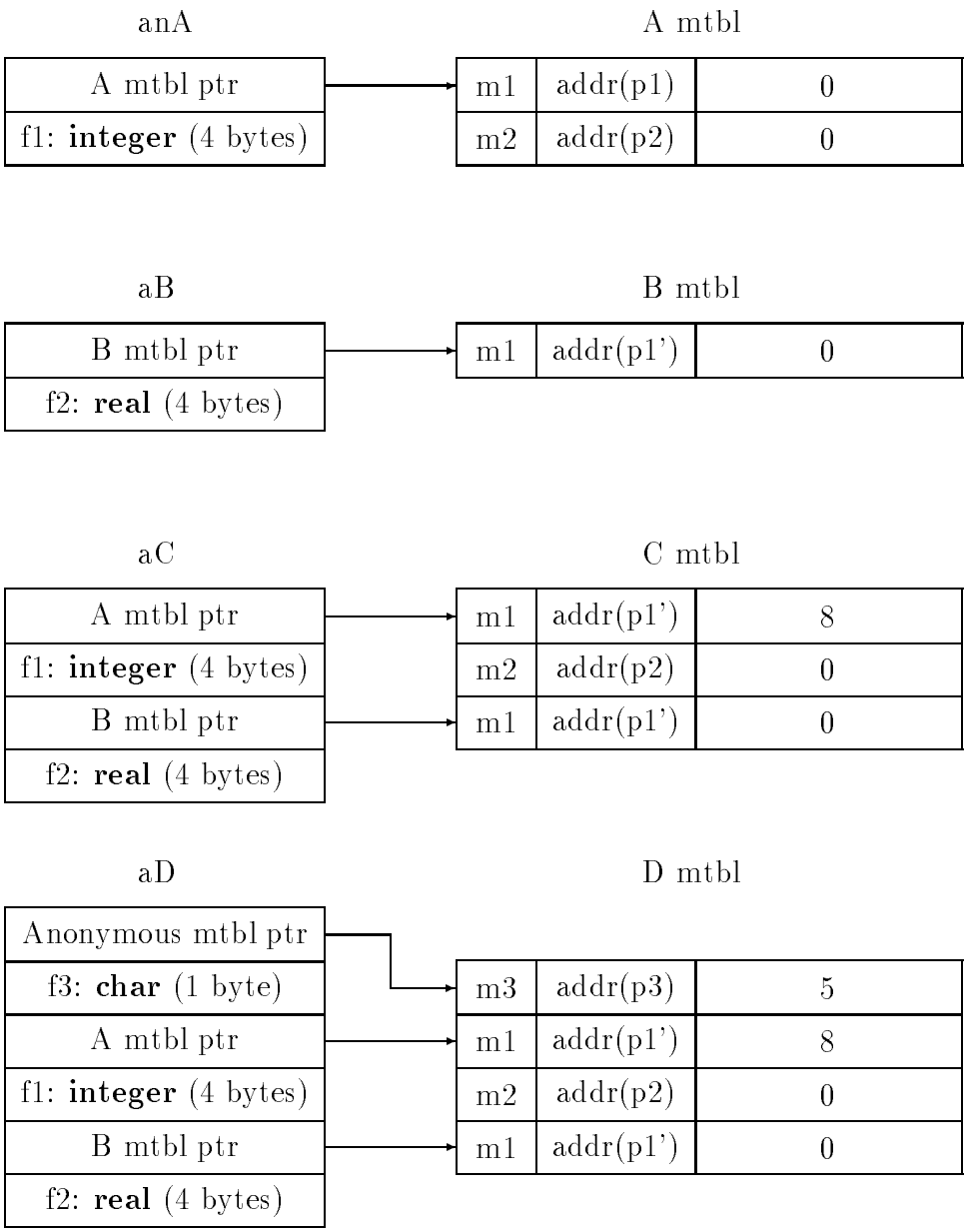

Figure 7.5. Layout of primitive and composite object yypes

7.1.3.3 Implementing Object Type Composition

The most involved operations on object types are those that compose object types, i.e., the **merge**, **shadow** and **override** operations. Apart from typechecking, **merge** is just a special case of **override.** Similarly, **shadow** is implemented just like **merge**. This presentation therefore focuses on the **override** operation. Everything said applies to the **merge** and **shadow** operations as well.

When composing two object types, as in

type C = A **override** B

objects of class C are represented by the concatenation of the A and B object representations, in that order.

As with ordinary multiple inheritance, the mtbl contains an offset field. As indicated above, the strategy is to separate the offset into two components - the static offset, which is inserted into any code referencing C objects, and the dynamic offset, which is stored in the mtbl.

The static offset is defined as :

StaticOffset(**object** $f_1 \ldots f_n$ **methods** $m_1 \ldots m_k$ **end** $, m_i) = 0, 1 \leq i \leq k$

StaticOffset(A **override** B, m) = $Size(A)$+*StaticOffset*(B, m) if m is declared in B else *StaticOffset*(A,m)

StaticOffset(C,m) is undefined if m is not declared in C. This poses no problem, since m cannot be referenced by C objects.

The dynamic offset will vary for each mtbl entry for m. A class C has an mtbl entry for m for every primitive component object type $S :> C$ that references m. That entry is referred to as $mtbl(C, S, m)$.

Let m be a method defined by some class C. Define *defaultValue*(C, m) to be the default value of m for class C, and let *defaultClass*(C, m) denote the class of the first parameter of *defaultValue*(C, m).

If $C <: S$ and m_i is a method referenced by S, the offset field of m_i for S in C is defined as shown in Figure 7.6.

$mtbl(C, S, m_i).offset=$
$Offset(C, defaultClass(C, m_i)) - Offset(C, S) - StaticOffset(S, m_i).$

Figure 7.6. Offset value in the mtbl.

References to methods of C are compiled as calls to procedures stored at particular entries in the C object's mtbl - that is, each method corresponds to a fixed offset from the C object's mtbl pointer, which is stored in the object. When such a method, m, is called, its first parameter (self) is the address of the C object, plus the offset for the subobject that declared it ($StaticOffset(C, m)$), plus the offset stored in the method's entry in the mtbl ($mtbl(C, C, m)$). This ensures that a B method is called with a *self* object corresponding to the B part of C. The extra offset from the mtbl ensures that if the method is overridden later, an appropriate adjustment is made so that the overriding method gets the correct *self* pointer. This same technique can be used if a method is overridden at object creation time.

Figures 7.7 and 7.5 show several object types, and the layout for their instances, respectively.

This example shows the use of an *anonymous object type*, a object type which has not been named, but is nevertheless used as a component of a composite object type. In D, an anonymous object type defines field f3 and method m3. Notice that $defaultClass(D, m3) = A$, and that the offset field for m3 in the anonymous object type's subtable reflects this. This is the only case where a primitive object type will have a nonzero offset in its mtbl.

```
B =  object f2:  real  methods m1() := p1'  end
C = A  override B
D =  object f3:  char  methods m3() := p3  end C

procedure p1'(aB:B)
procedure p3(anA:A)
```

Figure 7.7. Several composite object types

7.1.4 Pure Virtuals

A *pure virtual* method of a class C is a method that is declared, but given no definition (not even **nil**), the intent being that definitions may be supplied by other classes that are composed with C.

The above approach handles pure virtual methods as well. Note that if m is pure virtual in C, $defaultValue(C, m)$ is undefined, and so are $defaultClass(C, m)$ and $mtbl(C, S, m)$. This is of no consequence, since both the method value and the offset stored in the mtbl will never be used, since C must not be instantiated. What is important is that $StaticOffset(C, m)$ be defined. The original definition remains valid for pure virtuals.

In operator based inheritance, we have the unusual situation that a pure virtual definition may be supplied by either of the object types being combined. Contrast this with the asymmetric situation in conventional object-oriented languages, where only the modification may provide such a definition. If the actual method used comes from the modified class, a negative offset is used. Figure 7.8 gives examples.

The corresponding layouts are given in Figure 7.9. The symbol Φ denotes an undefined offset value.

The use of negative offsets is also useful when working with anonymous object types (see section 7.1.3.3 above).

```
E =  object  methods m1()  end
F = E  override B (* Pure virtual given value by modifying class *)
G = B  override E (* Pure virtual given value by modified class *)
H =  object  methods m1(), m4() := p4  end
I = E  override H (* Two pure virtuals combine *)

procedure p4(anH:H)
```

Figure 7.8. Examples of pure virtual methods.

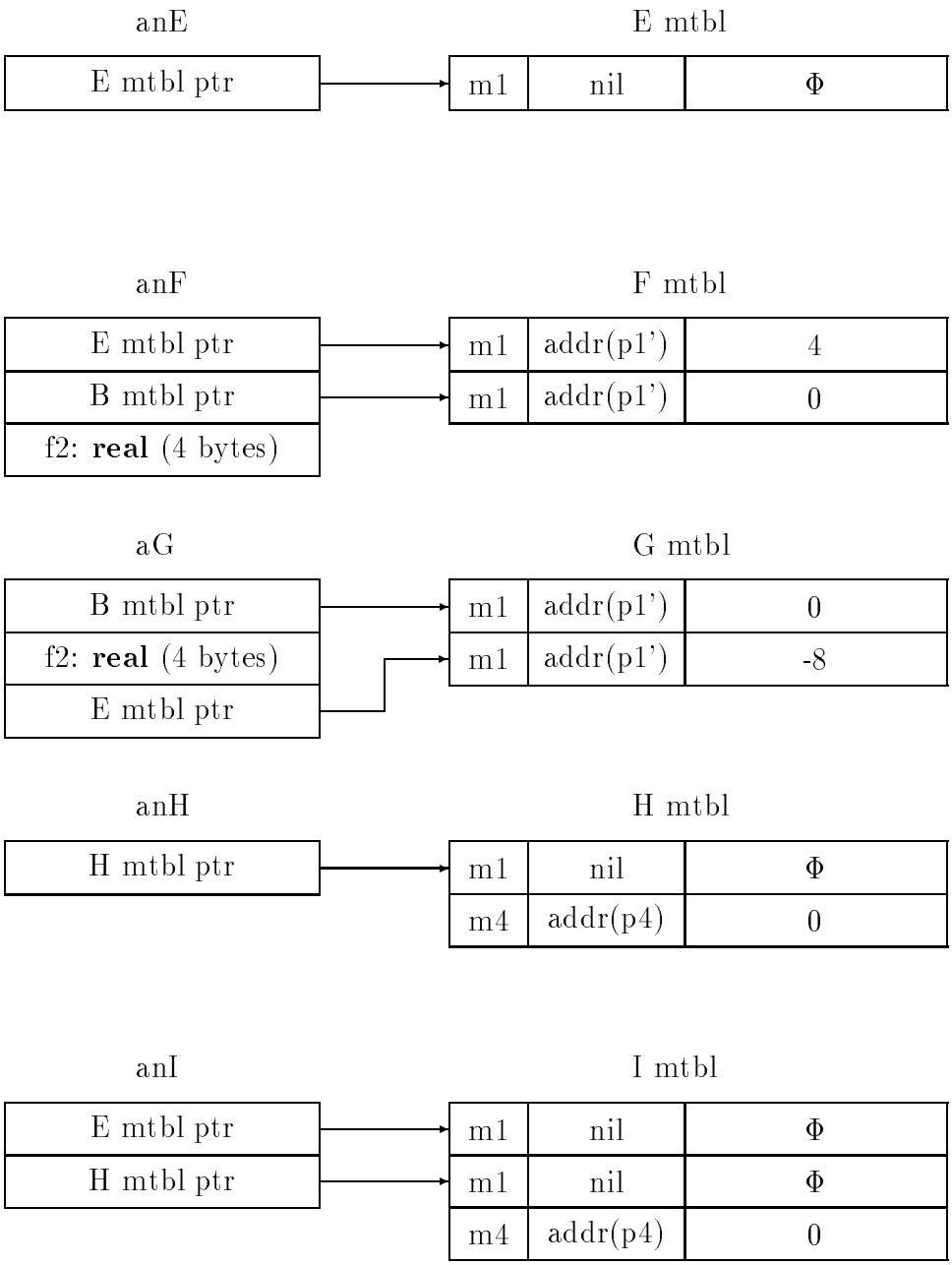

Figure 7.9. Layout of classes with pure virtuals

7.1.5 Other Operations

- **Restrict** and **Project.** The appropriate entries in the mtbl (those being made pure virtual) are nullified. The main effect of these operations is through the type system.

- **Copy As.** This operation has the effect of adding a new method to an object type. This is easily implemented by extending the method table with an additional entry for the extra operation. The contents of this entry are identical to those of the entry for the copied method. Of course, ifa the copied method is overridden, the entry for the new method will remain unchanged, and therefore available.

- **Rename As.** Renaming does not have any influence on a class' representation. Only the compiler's symbol tables are aware of the difference.

- **Show**. This operator has no effect except for typing.

7.1.6 Jigsaw Operations Not in Modula-π

Freeze operations are fairly redundant in Modula-π (see Chapter 6). However, they could easily be added. The freeze operation has no impact on the representation of a class. However, if a method has been frozen, it may be possible to generate code that uses it as an in-line function.

Hide and show operations are not needed, as Modula has its own encapsulation facilities. Furthermore, they do not fit in well with the notion that subobject boundaries are significant for typing.

7.1.7 Additional Details

There are many additional details that must be handled, including:

1. Assignment. Assigning an instance of class C to a variable of class $S :> C$, involves adding $\mathit{Offset}(C, S)$ to the address of the instance and assigning the result to the variable. The opposite case is when $S <: C$. This cannot be shown to be correct statically. However, Modula allows this, and generates a dynamic check to verify the correctness of the code. The offset is added, as before. The implementation of the dynamic check is discussed below.

2. Dynamic typechecking. In *Modula-3*, dynamic typechecks may be generated implicitly in some situations, or explicitly by the user using the constructs **istype, narrow** and **typecase**. The condition tested is whether one type is a subtype of another. All solutions discussed below associate a typecode with each distinct object type, which may be stored in the first word of each subobject. This typecode is also needed to support the **typecode** expression of *Modula-3*.

   For single inheritance, a dynamic typecheck may be implemented in constant time and linear space [12]. Under the new subtyping rules, this is no longer possible. It is possible to determine whether one type is a subtype of another in linear time and space, for instance by maintaining a list of each object type's supertypes, and testing recursively against this list. This is not an appealing option, since dynamic tests must be fast (especially since some are implicit).

   Alternately, a table, whose size is the square of the number of types in the program, could explicitly state if any type is a subtype of any other. This takes constant time, but quadratic space (though each entry only occupies a single bit). For most programs, the overhead would be acceptable. A program with a thousand object types would require a million bits for table space. Given

that a program with that many object types is clearly a substantial one, the overhead of 120 kilobytes of storage is not unreasonable. Still, reliance on a quadratic algorithm is worrisome.

A third option is described in [3]. A compressed transitive closure of the subtype relation is maintained. This structure allows testing in at worst $O(logn)$ time, but in practice gives constant time performance.

3. Garbage Collection. Each subobject should have a pointer to the beginning of the entire object. This is needed, because there may very well be pointers *into* objects (i.e., references to subobjects) while no pointer to the object as a whole exists.

4. Separate Compilation. The implementation scheme discussed above relies on knowledge of the size of objects. If the object type is a (partially) opaque type, its size is not known at compile time. Nevertheless, offsets need to put into code, and into tables, while the size of subobjects of opaque types is unknown. These offsets must be fixed at link time. If combinations of offsets need to be computed, and the linker cannot do this, then some initialization code might be needed. This is not a new problem; it already exists in *Modula-3*.

This concludes the discussion of operator-based inheritance and its implementation.

CHAPTER 8

FINALE

How many good ideas can there really be? *Luca Cardelli.*

There is one remaining task: to summarize the the contributions of this research, compare them with other work, and suggest directions for the future.

Section 8.1 surveys various studies related in some way to the research reported on in this dissertation. Future work is discussed in section 8.2. Conclusions are given in section 8.3.

8.1 Related Work

8.1.1 Jade

Jade is a module manipulation system based upon *Emerald*. In many ways, *Jade* is *Jigsaw*'s closest relative. *Emerald* and *Jade* clearly distinguish subtyping from inheritance, and support only the former. *Jade* rejects inheritance due to the many difficulties it has traditionally raised, as described in Chapter 2. As an alternative, *Jade* defines parameterized abstractions called *components*. Like *Jigsaw* modules, components have no free variables, so they are "self-sufficient" constructs. External dependencies are expressed using *habitats*, a compile time parameterization mechanism. This is similar to the use of declarations in *Jigsaw* for module interconnection. However, habitats support parameterization of components but not inheritance. Modifications to components must be done either manually or with environmental support. In [58], the idea of automating such operations using "simple set theoretic operators" is suggested, but not explored. The essential difference between such

operators (if they were developed) and those of *Jigsaw* is that the bidirectional relationship between abstraction and parameter characteristic of abstract classes is not available.

8.1.2 CommonObjects

The definitive study of inheritance with respect to encapsulation is [65], which has been cited extensively in this dissertation. In conjunction with that study, Snyder developed an object-oriented *LISP* dialect called *CommonObjects*[63]. *CommonObjects* was the first object-oriented language that did not violate encapsulation. It also allowed the definition of mixins. However, the language was dynamically typed, reflecting its *LISP* heritage. Though encapsulation is a key aspect of modularity, it is not the only one. Other issues, such as hierarchy, were not considered.

Mixins were recognized as an important programming idiom in [65] but were not considered as a full fledged construct. *CommonObjects* employed a formulation of inheritance called *tree inheritance*. The terminology reflects the operational, graph-oriented approach to inheritance prevalent at the time the study was undertaken. Despite the differences in terminology and outlook, tree inheritance is very similar to mixin based inheritance. In both cases, ancestors of a class that are reachable by multiple paths in the inheritance hierarchy are reflected multiple times in the class' instances. However, in tree inheritance a class has multiple immediate ancestors, and has direct access to all of them. Tree inheritance is therefore not a purely linear approach, but rather, as its name implies, a tree structured one. Classes in *CommonObjects* could be viewed as mixins with multiple arguments.

8.1.3 Mixins

This work grew out of an earlier study of mixin-based inheritance [7]. Some of the limitations of mixin based inheritance have been addressed here. These include the absence of fine-grain sharing, of renaming facilities and of a symmetric merge operation.

Until now, mixins have been modeled as parametric abstractions called wrappers. Cook used an operator combining a generator and a wrapper in his compositional semantics of inheritance [18]. This operator was also used by Hense [28]. In [7], the override operation was defined as a binary operation on wrappers, enabling composition of mixins. Here, an alternate formulation of wrappers as functions from generators to generators has been given. The main purpose of wrappers was to allow access to overridden definitions. The required functionality can be achieved using an explicit operator for this purpose. This allows the use of generators instead of wrappers, simplifying definitions. This reflects the strategy of simplifying the structure and pushing more functionality into the operator set.

8.1.4 Generator Operations

Many of the operators presented here were first proposed by Cook in [18]. There, a general mechanism for deriving generator operations from record operations was described. However, the operators defined by Cook were used to illustrate the principle of manipulating self-reference by means of generators. In modeling programming language constructs, more elaborate operators were used. In particular, it was necessary to introduce wrappers, as discussed in section 5.1.3.

The novelty here is in providing a comprehensive suite of operations, and making them explicit linguistic constructs. In addition, the uniform use of generators to model all definitional structures is new. The operator suite also includes new operations (namely **hide, show, freeze, freeze-except** and **copy-as**).

8.1.5 Mitchell

Mitchell, in [53], presented an extension to the *ML* module system that is in some ways similar to this work. Mitchell also chose to incorporate inheritance into a module language, an extension of the *ML* module system [44]. Some similar operations are supported, embedded in a more conventional syntax. Underlying both systems are denotational models involving the manipulation of self-reference, and typing based on bounded quantification. There are many differences, however.

Central to this thesis is the notion that inheritance itself can be used as a modularity mechanism. Inheritance is an essential part of the module language, "gluing" modules together by merging self-reference. Such a formulation of inheritance must preserve encapsulation. This contrasts with Mitchell's view of inheritance as "a mechanism for using one declaration in writing another." Even though inheritance is part of the module system, it is not essential to it. Instead, the *ML* notions of structures and functors are used to define and interconnect modules. Some of the inheritance constructs defined in [53] violate encapsulation (*viz.* **copy except, copy only**). These constructs inherently require knowledge of the internal structure of the "parent" module.

A consequence of the semantics of **copy except, copy only** is that separate compilation is compromised. A parent module must always be compiled before its use, and any change to it requires recompilation of its heir modules [46]. *Jigsaw* supports inheriting from separately compiled modules without restriction.

The approach presented in this dissertation has the benefits of simplicity and modularity. It does not rely upon dependent sums or products, or on multiple universes of types. It is explicitly formulated as a framework for manipulating modules where all functionality is supported by operators. Making its structure explicit facilitates applying the framework to a broad spectrum of languages. Lan-

guage designers may easily add or modify operations as necessary. An expression based language also allows users to compose operations more freely.

The *Jigsaw* framework supports abstract classes and mixins.[1] Mixins cannot be expressed in the framework of [53], and there is no explicit support for abstract classes (though the traditional device of giving dummy definitions for pure virtual methods is always available, with its concomitant disadvantages).

On the other hand, Mitchell's approach supports modules implementing abstract data types. This allows for combining traditional algebraic (or higher order) data types with object-oriented formulations. *Jigsaw* supports only the pure object-oriented approach. It would be desirable to extend the framework with an analogous set of operators for abstract data types. However, there are technical difficulties related to the typing of existential data types.

A related issue is the use of structural subtyping, in contrast to "name-based" subtyping in [53]. Both forms are useful; here, the focus is on structural subtyping, which is more appropriate between different modules or programs [12].

Finally, unlike [53], precise semantic definitions of all operations have been given.

8.2 Future Work

8.2.1 Name-based Typing

Jigsaw, as presently formulated, does not support name based typing. However, this does not seem to present a serious difficulty. In Chapter 6, the brand mechanism of *Modula-3* was used to obtain the functionality of named types in a structural-type setting. This is a generally applicable solution. Brands are viewed as parameters to type constructors, and are components of a type's structure [55]. The uniqueness of brands can be enforced syntactically, as in *Modula-3*. Within a *Jigsaw* module, a brand can be given by the user only once. When modules are combined, it is

[1] Abstract classes are mentioned in [53], but only as substitutes for interfaces.

necessary to guarantee that brands are unique across modules. In effect, the brands introduced by a module are part of its interface, and may not be duplicated by other modules. This can be checked when modules are combined.

An alternative is to provide an intrinsic notion of name-based typing in *Jigsaw*. This could be done by adding types as components of interfaces and modules. Formalizing this would mean that generators would become dependent products and records dependent sums. The details of this (especially with respect to recursive types) have not been studied carefully, however.

8.2.2 Abstract Data Types

Abstract data types are both more useful and more problematic than named types. A formalization based on existentially quantified types is problematic, because of type abstraction. In particular, creating new abstract data types by combining the abstract types form two modules runs into the same difficulty that has arisen time and again in this dissertation - how to typecheck inheritance in the presence of type abstraction. A rigorous definition of inheritance on ADTs is an important and substantial research issue.

8.2.3 Formal Specification of Inheritable Modules

A primary motivation for *Jigsaw* is reusability. One of the possible side-effects of increased reuse is an increased emphasis on formal specification and verification of software components. The reason for this is economic in nature. The larger the market for a component, the more feasible it is to invest in the expensive process of formally verifying a software component. Conversely, users of "off-the-shelf" components may begin to demand more precise specifications of the software they purchase.

The preceding observations draw attention to a problem not yet addressed by the formal methods community. While there is an abundance of work on specifying

how software behaves when used, there is a dearth of research into how to specify how software behaves when inherited.

There is a need to specify how a class will respond when modified, which implies knowledge of method interdependencies. In addition, if a revised method is changed, even if it preserves the previous version's specification, it may induce other changes to the object's state. It may be desirable to specify that certain methods do not have any additional effects (a form of frame problem).

One reason that this problem has not yet come to the forefront of attention is that in most programming languages, there is no way to inherit from a separately compiled class or module. This implies that source code is always available, and this source is the specification used to understand how the class will behave when modified. An exception is *Modula-3*, where one can inherit from a separately compiled object type. Specifying how to do this is challenging, and is done informally, in English. For some excellent examples, see [55, Chapter 6].

The semantic framework of *Jigsaw* may suggest a starting point of attacking this problem. Traditional specification deals with the behavior of records with function valued attributes. The problem just posed may be thought of as specifying how generators behave.

8.2.4 Prototypes

Jigsaw was originally designed to deal with inheritance among classes. Though there are some differences, the framework can be carried over into the world of delegation.

Typing of delegation raises the same acute problems that inheritance in any polymorphic context does. Therefore, typechecking will be ignored here.

Assuming *Jigsaw* modules are first class entities, and *Jigsaw* operations are executed at run time, the effect of say, a **override** operation is to produce a new

module, which contains copies of the two modules that were arguments to the **override** operation, with the attributes of the dominant module overriding those of the other module.

In contrast, no new object (module) is created under delegation. The delegating object *references* the delegate. As a consequence, the state of the delegate is shared with all its delegators.

The principle difference between *Jigsaw*'s semantics and those of a delegation based language like *SELF* is that between copy semantics and reference semantics. An equivalent conclusion is reached independently by Taivalsaari in [66].

Based on this insight, *SELF* style delegation can be supported with a suite of inheritance operators. The description will have an uncomfortably operational flavor, but remember, delegation is an inherently operational notion.

At this point, a single example will be shown, to give some insight. Deriving a full denotational semantics of a modular form of delegation based on this insight seems fairly straightforward.

o_1 **override** o_2 produces a new object, whose only function is to forward messages to o_2, with a revised self (client). If the messages are not understood by o_2, they are forwarded to o_1.

The space of values being manipulated does not really consist of objects, but of references to functions of type $Filter$, such that

$$Filter = Filterref \rightarrow Msg \rightarrow Value$$

In other words, $Filter$'s are functions that take a reference to a $Filter$ (representing $self$), a message, and produce a value. $Filter$s are analogous to generators.

The conventional syntax $o.m$ really stands for $o(o)(m)$. In other words, a message send in a delegation language really invokes a generator.

All operators can now be defined in a manner completely analogous to their generator versions. The results are always references to $Filter$ functions, which

perform the necessary manipulations upon *self* and filter messages as appropriate before sending them to the original operands.

Implementation of a delegation based language along these lines is an interesting variation on the *Jigsaw* framework, in which interface checking would be overridden by *true*, generators replaced by *Filters*, and module operators redefined accordingly (including dynamic interface checking).

8.2.5 Nested Modules Revisited

In *Beta*, nested classes can be virtual, as shown in section 3.1.3. The same applies in *Jigsaw*. However, *Jigsaw* adopts a purely static type system, restricting subtyping (subinterfacing) on modules to type (interface) equivalence. *Beta* supports subtyping on patterns, and relies on dynamic typechecks to guarantee safety. This flexibility is what enables *Beta* to express mixins as shown in Chapter 3. Useful mixins are *polymorphic* class abstractions. In *Jigsaw* modules are treated monomorphically. Similarly, *Beta* allows entire class hierarchies to be modified by inheritance. This is not well supported in *Jigsaw*. Of course, *Jigsaw* supports mixins more directly, as shown in section 4.2.9. Inheriting entire hierarchies seems valuable however. If *Jigsaw* adopted dynamic typechecking to augment its type system, this could be supported, though it would be costly.

Another distinction is that *Beta* identifies classes and types. This has the disadvantages mentioned in Chapter 2, but allows *Beta* to support type abstraction using the same virtual pattern mechanism used for inheritance [49].

Nested modules in principle also support the notion of *class variables* found in languages like *Smalltalk*. Class variables are variables shared by all instances of a class. A module that nests another module inside it, can serve as a "factory" [20] and produce initialized instances of the nested module. The surrounding module serves as a repository of shared data among all instances of the nested module.

Again, module subtyping restricts the usefulness of such designs. A richer notion of module subtyping would allow *Jigsaw* to support these highly expressive constructs. Use of dynamic typing, as in *Beta* is one option, but a costly one. In [19] static type systems that address some of these problems are discusses.

8.2.6 Process Calculi

Object orientation presents a natural model of concurrency, and concurrent object-oriented programming has been the focus of considerable attention [75]. The operator based approach advocated in *Jigsaw* seems to fit well with process calculus models of concurrency in the style of *CCS* [61]. Nierstrasz has investigated such calculi in an object-oriented context [57]. More recent work by Nierstrasz investigates the integration of process and λ calculi [56]. In [56], it is shown how to express the fixpoint operator in such an integrated calculus. It should therefore be possible to integrate *Jigsaw* style generator definitions into this framework. This leads toward the exciting possibility of an expression based language for composing modular, concurrent object definitions.

8.3 Conclusion

This dissertation has provided a framework for modularity in programming languages. In this framework, known as *Jigsaw*, inheritance is understood to be an essential linguistic mechanism for module manipulation. The framework is unusually expressive, theoretically sound, efficiently implementable and language independent.

Specifically, the dissertation has made the following contributions:

- Inheritance has been characterized as a module manipulation mechanism.
- The notion of mixins has been identified as an important abstraction missing from current object-oriented programming languages, in contravention of established principles of language design.

- For the first time, a broad array of linguistic features has been integrated in a cohesive manner, including multiple inheritance, mixins, encapsulation and strong typing.
- A clean, modular semantics for multiple inheritance has been developed.
- A linguistic framework based directly on the semantics has been constructed. This serves as a framework for modular language specification, and as a specification of a framework for modular language implementation, independent of a particular computational paradigm.
- The applicability of the framework to existing programming languages has been demonstrated.
- An efficient implementation scheme for the constructs introduced has been described.

Beyond the specific contributions, the dissertation demonstrates once again the importance of denotational semantics to programming language design.

REFERENCES

[1] Reference manual for the Ada programming language. ANSI/MIL-STD-1815 A, 1983.

[2] Agha, G. *Actors: A Model of Concurrent Computing in Distributed Systems.* MIT Press, Cambridge, Massachusetts, 1986.

[3] Agrawal, R., DeMichiel, L. G., and Lindsay, B. G. Static type checking of multi-methods. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 1991), pp. 113–127.

[4] America, P. A parallel object-oriented language with inheritance and subtyping. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990), pp. 161–168.

[5] Bentley, J. L. *More Programming Pearls.* Addison-Wesley, Reading, Massachusetts, 1988.

[6] Borning, A. H. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference* (1986).

[7] Bracha, G., and Cook, W. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990).

[8] Canning, P., Cook, W., Hill, W., Mitchell, J., and Olthoff, W. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture* (1989), pp. 273–280.

[9] Canning, P., Cook, W., Hill, W., and Olthoff, W. Interfaces for strongly-typed object-oriented programming. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1989), pp. 457–467.

[10] Cardelli, L. A semantics of multiple inheritance. In *Semantics of Data Types* (1984), vol. 173 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 51–68.

[11] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report (revised). Tech. Rep. 52, Digital Equipment Corporation Systems Research Center, Dec. 1989.

[12] Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G. The Modula-3 type system. In *Proc. of the ACM Symp. on Principles of Programming Languages* (Jan. 1989), Association for Computing Machinery, pp. 202–212.

[13] Cardelli, L., and Mitchell, J. C. Operations on records. Tech. Rep. 48, Digital Equipment Corporation Systems Research Center, Aug. 1989.

[14] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17*, 4 (1985), 471–522.

[15] Cargill, T. Controversy: The case against multiple inheritance in C++. In *Usenix Winter Conference* (Jan. 1991).

[16] Chambers, C., and Ungar, D. Making pure object-oriented languages practical. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 1991), pp. 1–15.

[17] Connor, R., Dearle, A., Morrison, R., and Brown, A. An object addressing mechanism for statically typed languages with multiple inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1989), pp. 279–285.

[18] Cook, W. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[19] Cook, W., Hill, W., and Canning, P. Inheritance is not subtyping. In *Proc. of the ACM Symp. on Principles of Programming Languages* (1990), pp. 125–135.

[20] Cox, B. J., and Novobilski, A. *Object-oriented Programming: An Evolutionary Approach*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991.

[21] Dahl, O., and Nygaard, K. Simula: An Algol-based simulation language. *Communications of the ACM 9* (1966), 671–678.

[22] Ducournau, R., and Habib, M. On some algorithms for multiple inheritance in object-oriented programming. In *European Conference on Object-Oriented Programming* (1987), pp. 243–252.

[23] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[24] Goldberg, A., and Robson, D. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[25] Guimaraes, N. Building generic user interface tools: an experience with multiple inheritance. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 1991), pp. 89–96.

[26] Harper, R., MacQueen, D., and Milner, R. Standard ML. Internal Report ECS-LFCS-86-2, Edinburgh University, Mar. 1986.

[27] 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.

[28] Hense, A. V. Denotational semantics of an object oriented programming language with explicit wrappers. Tech. Rep. A 11/90, Fachbereich Informatik, Universitaet des Saarlandes, Nov. 1990.

[29] Hense, A. V. Wrapper semantics of an object oriented programming language with state. Tech. Rep. A 14/90, Fachbereich Informatik, Universitaet des Saarlandes, July 1990.

[30] Hense, A. V. Explicit wrappers and multiple inheritance, Feb. 1991. Unpublished manuscript, Fachbereich Informatik, Universitaet des Saarlandes.

[31] Holzle, U. Why static typing is not important for efficiency, or why you shouldn't be afraid to separate interface from implementation. Position paper in ECOOP'91 workshop on Types, Inheritance and Assignments, J. Palsberg and M. Schwartzbach, editors.

[32] Jensen, K., and Wirth, N. *Pascal User Manual and Report*, second ed. Springer-Verlag, 1978.

[33] Johnson, R. E., and Russo, V. F. Reusing object-oriented designs. Tech. Rep. UIUCDCS 91-1696, University of Illinois at Urbana-Champagne, May 1991.

[34] Kamin, S. Inheritance in Smalltalk-80: A denotational definition. In *Proc. of the ACM Symp. on Principles of Programming Languages.* Association for Computing Machinery, 1988, pp. 80–87.

[35] Keene, S. E. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, 1989.

[36] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1978.

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

[38] 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.

[39] Kristensen, B. B., Madsen, O. L., Moller-Pederson, B., and Nygaard, K. The Beta programming language – a Scandinavian approach to object-oriented programming, Oct. 1989. OOPSLA Tutorial Notes.

[40] Krogdahl, S. Multiple inheritance in Simula-like languages. *BIT 25* (1985), 318–326.

[41] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1986), pp. 214–223.

[42] Linton, M. A., Calder, P. R., and M.Vlissides, J. InterViews: A C++ graphical interface toolkit. Tech. Rep. CSL-TR-88-358, Stanford University, July 1988.

[43] Liskov, B., and Guttag, J. *Abstraction and Specification in Program Design*. MIT Press, Cambridge, Mass., 1986.

[44] MacQueen, D. Modules for Standard ML. In *Proc. of the ACM Conf. on Lisp and Functional Programming* (Aug. 1984), pp. 198–207.

[45] Madany, P. W., Campbell, R. H., Russo, V. F., and Leyens, D. E. A class hierarchy for building stream-oriented file systems. In *European Conference on Object-Oriented Programming* (July 1989), S. Cook, Ed., British Computer Society Workshop Series, Cambridge University Press, pp. 311–328.

[46] Madhav, N., September 1991. Personal communication.

[47] Madsen, O. L., November 1990. Personal communication.

[48] Madsen, O. L., Magnusson, B., and Moller-Pederson, B. Strong typing of object-oriented languages revisited. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990), pp. 140–149.

[49] Madsen, O. L., and Moller-Pederson, B. Virtual classes, a powerful mechanism in object-oriented programming. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 1989), pp. 397–406.

[50] Manna, Z. *The Mathematical Theory of Computation*. McGraw-Hill, 1974.

[51] Meyer, B. *Object Oriented Software Construction*. Prentice-Hall International, Hertfordshire, England, 1988.

[52] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, 1990.

[53] Mitchell, J., Meldal, S., and Madhav, N. An extension of Standard ML modules with subtyping and inheritance. In *Proc. of the ACM Symp. on Principles of Programming Languages* (Jan. 1991), pp. 270–278.

[54] Moon, D. A. Object-oriented programming with Flavors. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1986), pp. 1–8.

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

[56] Nierstrasz, O. Towards an object calculus. In *ECOOP'91 Workshop on Object-based Concurrent Computing* (July 1991).

[57] Nierstrasz, O., and Papathomas, M. Viewing objects as patterns of communicating agents. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990), pp. 38–43.

[58] Raj, R. K., and Levy, H. M. A Compositional Model for Software Reuse. In *European Conference on Object-Oriented Programming* (July 1989), S. Cook, Ed., British Computer Society Workshop Series, Cambridge University Press, pp. 3–24.

[59] Reddy, U. S. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conf. on Lisp and Functional Programming* (1988), pp. 289–297.

[60] Remy, D. Typechecking records and variants in a natural extension to ML. In *Proc. of the ACM Symp. on Principles of Programming Languages* (1989), pp. 77–88.

[61] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1989.

[62] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An introduction to Trellis/Owl. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1986), pp. 9–16.

[63] Snyder, A. CommonObjects: An overview. *SIGPLAN Notices 21*, 10 (1986), 19–28.

[64] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1986), pp. 38–45.

[65] SNYDER, A. Inheritance and the Development of Encapsulated Software Components. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 165–188.

[66] TAIVALSAARI, A. Towards a taxonomy of inheritance mechanisms in object-oriented programming, September 1991. Licentiate thesis.

[67] TENNENT, R. *Principles of Programming Languages.* Prentice-Hall, 1981.

[68] UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HOLZLE, U. Parents are shared parts of objects: Inheritance and encapsulation in SELF, 1990. In The SELF papers, compiled by Urs Holzle.

[69] UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HOLZLE, U. The SELF manual, version 1.0, July 1990.

[70] VLISSIDES, J., AND LINTON, M. Unidraw: A framework for building domain-specific graphical editors. Tech. Rep. CSL-TR-89-380, Stanford University, July 1989.

[71] WAND, M. Type inference for record concatenation and multiple inheritance. In *Proc. IEEE Symposium on Logic in Computer Science* (1989), pp. 92–97.

[72] WEGNER, P. The object-oriented classification paradigm. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 479–560.

[73] WEINAND, A., GAMMA, E., AND MARTY, R. ET++ - an object-oriented application framework in C++. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1988), pp. 46–57.

[74] WIRTH, N. *Programming in Modula-2.* Springer-Verlag, 1983.

[75] YONEZAWA, A., AND TOKORO, M., Eds. *Object-Oriented Concurrent Programming.* MIT Press, 1987.