OBSTACL:

A LANGUAGE WITH OBJECTS, SUBTYPING, AND CLASSES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Amit Jayant Patel
December 2001

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Mitchell
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Kathleen Fisher

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David Dill

Approved for the University Committee on Graduate Studies:

Abstract

Widely used object-oriented programming languages such as C++ and Java support software engineering practices but do not have a clean theoretical foundation. On the other hand, most research languages with well-developed foundations are not designed to support software engineering practices. This thesis bridges the gap by presenting OBSTACL, an object-oriented extension of ML with a sound theoretical basis and features that lend themselves to efficient implementation. OBSTACL supports modular programming techniques with objects, classes, structural subtyping, and a modular object construction system. OBSTACL's parameterized inheritance mechanism can be used to express both single inheritance and most common uses of multiple inheritance. In addition, it can be used to implement designs that are difficult to implement with conventional single or multiple inheritance.

There is a large space of possible designs for an object-oriented language. The design of OBSTACL is driven by program design and maintenance needs rather than simplicity or elegance. It explicitly supports both object and non-object forms of abstraction and does not attempt to reduce everything to an object. Although simplicity was not a goal, OBSTACL's objects and classes are simpler than those in many object-oriented languages because many of the problems solved by complicated features in a purely object-oriented language are solved instead by a rich set of non-object features such as modules, first class functions, abstract data types, and parametric polymorphism (generics). In addition several fundamental questions about abstractions (identity, deep vs. shallow equality, copying vs. cloning, and mutation) are answered by supporting both objects and non-objects in the same language. The resulting language is straightforward to implement efficiently and relatively simple to analyze mathematically.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the programming language foundations community, languages are often designed around elegant features that can express a wide variety of programming constructs. Examples include Scheme, designed around the lambda function, and Smalltalk, designed around the object. This dissertation shows that designing a language by examining program design and maintenance issues instead of reuse or mathematical elegance can lead to a clean object-oriented language that supports code reuse, a strong mathematical foundation, and a relatively simple type system. OBSTACL is our new approach to extending ML with objects. Focusing on design and maintenance issues, we end up with a language that is straightforward to implement, efficient, and relatively simple to analyze.

1.1 Approach

> Architects everywhere have recognized the need of ... a tool which may be put in the hands of creators of form, with the simple aim ... of making the bad difficult and the good easy.
>
> *–Le Corbusier (The Modulor)*

In general, we want our language to *encourage* good designs both by making good designs easier to program and by making bad designs more difficult to program. It is not possible to *prevent* bad design; we simply don't want to make it difficult to write good programs. We want a strong foundation—the language should be type safe, subject to analysis, and amenable to a precise specification. We do not attempt to reduce everything to an object (as in Smalltalk), maximize expressiveness, minimize features, or maximize possibilities for code reuse. The last of these may seem surprising, given that object orientation is often associated with code reuse. In OBSTACL, we want to promote the writing of highly reusable code. Reuse of well written, well-tested code is preferable to reimplementing the desired functionality. However, we should

distinguish between facilitating the *writing of reusable code* and the *reuse of code.* The former involves writing code with enough flexibility to allow it to be used in many different parts of a program or a set of programs. The latter includes making it easier to reuse code that was not designed to be used in a new situation. In OBSTACL we do not promote this latter goal; we believe that such reuse is highly error prone, because it introduces a fragile dependency: the author of the code being reused is not aware of the reuse, so he may change the code in a way that breaks any modules using the code. To reuse code robustly, the author of the code must know that the code will be reused, so that he can write a specification and preserve all behavior described in the specification when he makes changes. The signature and specification serves as a contract between the author of the code being reused and the programmer reusing the code. Our goal is therefore to support the writing of reusable code, and not the reusing of code that is not designed to be reused.

1.2 Scope

The focus of this work is exploring language design keeping program design and maintenance in mind. In addition, a calculus and a sketch of an implementation are presented. There are many decisions to be made in designing a language. To avoid spending time reinventing variables, functions, control flow, and so on, we take ML as a base language. ML is a rich language with a strong mathematical foundation. A side effect of choosing a rich but non-object-oriented language is that it is clear what features are object-oriented and what features are not. In languages like C++ [Str97] and Object Pascal, which are built on top of feature-poor languages, it is difficult to distinguish between those features added to support objects and those added for modularity and abstraction in general. It is important to note that our goal is not to extend ML—we use ML here to avoid reinventing the wheel, and our results are not ML-specific. Where needed, OBSTACL draws on ML, using a compatible syntax and assuming the presence of basic types (such as integer, function, tuple, list, and record).

We design objects primarily to represent "physical" objects like files, locks, and bank accounts. Abstract objects like points, strings, lists, and sets are already handled well in non-object-oriented languages like ML and Ada-83, except when dealing with multiple representations *for performance reasons*. We will not address adding features for performance or convenience; instead we use objects for design and maintenance reasons.

The types in ML are what we call ***algebraic types***. Algebraic types describe an unchanging set of values with predictable and useful properties. On these values one can build relations (more commonly, functions). For example, the type Integer is a set $\{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\}$. On this set we can build the addition relation. OBSTACL extends ML with ***object types***. Object types as we use them are dynamic heterogeneous sets with variable properties. Unlike algebraic types, object types may change over time. For example, the type Automobile will

have new elements added to it when a new car is built. An object type does not have to be homogeneous. For example, the Automobile type may contain both a Volkswagen Beetle and a Chevy Camaro, even though these objects have different properties. In addition, the set of variants in the type can change over time.[1] For example, next year there will be new models of automobiles in the set. In this work we treat object types as fundamentally different from algebraic types and do not attempt to make object types fit the algebraic type model. Instead we explore the space created by having both in the language. A List of Automobiles for instance is neither purely algebraic nor purely object. Like algebraic types, List of Automobiles is homogeneous (all its members are lists) and can have relations (`length`, `reverse`) defined on its elements. The set of List of Automobiles is unchanging, except when the set of Automobiles changes.[2] It is the author's experience that forcing all types to be algebraic (as in ML) or object (as in Smalltalk [Ing78] and Java[3] [AG96]) is less desirable than simply allowing a type to be either. Furthermore, unifying them by taking a union of their properties leads to a complex system.[4] OBSTACL preserves algebraic types and also provides a simple object type. Many features of other languages can be expressed as the combination of these types rather than as new features.

Expressiveness is not the only goal in language design. We also want guarantees on data and code. Take for example tuples[5] and lists[6]. A mutable heterogenous list is more expressive than both tuples and lists. Why does ML have homogeneous variable-length lists and heterogenous fixed-length tuples? When the data structure is *restricted*, we can make some *guarantees* on its values. In this example, with lists we know the types of its elements are all the same, but we do not know the number of elements. With tuples we know the number of elements, and their types can differ. Restrictions give us more knowledge at the expense of flexibility. Therefore we are not simply trying to make an expressive language—we are trying to make a tradeoff between flexibility and guarantees. The tuple vs. list example shows that sometimes we do not need a single construct that can handle all possible cases; sometimes, it is better to have complementary structures with different properties. That is our goal with objects and abstract algebraic data values.

[1] This is what distinguishes object types from the algebraic *union* types, described in 3.2.3.

[2] Changes to the mixed type reflect changes to the object type, and do not arise on their own.

[3] There is a proposal for adding value objects (corresponding to our algebraic types) to Java. [Gos97]

[4] In C++, for instance, algebraic operations such as `operator=` do not mix well with object concepts such as inheritance [Mey92].

[5] A tuple is a fixed-length sequence of values of different types.

[6] A list here is any variable-length sequence of values of the same type.

1.3 Language design

Designing programs is about space: one creates modules to separate program components and interfaces to connect components together into a program. The goals are to divide a problem into components that one can understand more easily. Program structures with small components and few connections between them are easier to understand. Maintaining programs is about time: one creates modules to allow smaller units of change and interfaces to specify what is stable over time. While designing a program one must take into account the initial structure of the system and how it may change over time. Abstractions factor multiple program components into a more general one. To support design and maintenance, we look in chapter 3 at forms of abstraction and characteristics of the dependencies they create. Reducing dependencies generally corresponds to better design and easier maintenance. We conclude that simple objects are not appropriate for all situations, and resolve to allow both object and non-object abstractions. We further explore the characteristics of objects and non-objects, in particular, issues of identity, equality, comparison, and copying. A simple object type looks similar to tagged union types or record types. Why can we not reuse those types for objects? Object types and record types both can expose a set of public functions that encapsulate private data. Objects however are the result of a distributed recursive definition that spans modules. Records lack this recursive form of definition. Object types and union types both allow the definition of a set of data variants and a set of functions defined on each variant. However the grouping of these definitions into modules is different: objects have one shared definition of a set of functions plus modular definition of data variants, while unions have one shared definition of a set of data variants, plus modular definition of functions. This difference directly impacts extensibility over time: objects are easily extended with data variants, while unions are easily extended with new functions. Neither is a suitable replacement for the other. It is not sufficient to consider only how a program is written initially; the ease of making changes over time must be taken into account. Maintainability plays a key role in the design of OBST-ACL's object system.

Since new data variants can be added when using objects, code that uses objects should not depend on which variant is provided but instead only on the behavior of that object. An illustrating consequence of this principle is that two objects with exactly the same definitions should be usable in the same places and behave the same. As simple as this sounds, many languages do not allow these two objects to be interchanged. The ability to substitute one variant for another is important for code reuse. A library designed for one implementation of objects will continue to work when a new implementation is used instead—the library is reused with a new kind of object. Since the code being reused is external to the object's implementation, we call this ***external reuse***. The other form of reuse common in many object programming systems is inheritance, which we call ***internal reuse***. With inheritance, parts of the definition of

one object can be used in defining another variant of object. Internal reuse is easy to recognize, but because there is more code outside an object definition than inside, external reuse has the most benefit.

In this work we focus on objects. In chapter 4 we consider tradeoffs in the design of an object system. First we look at the capabilities of objects and choose a simple system with message sends, comparison, imperative update, and structural types. We then examine the process of creating an object, which at first glance is a straightforward task, but in the presence of immutable fields is actually a difficult problem that requires more analysis. While popular object-oriented languages such as C++ and Java are overwhelmingly class-based, most previous theoretical work for object-oriented languages was based on objects without classes [AC96, FHM94], with classes added as an extra layer on top of the basic object system. The decision to use or not use classes in OBSTACL was based on design and modularity considerations, and not the desire to be similar to popular object-oriented languages or the theoretical work. We choose to use classes with single inheritance. Instead of multiple inheritance we use runtime mixin inheritance. Mixins were first introduced in the Flavors system [Moo86] and CLOS [Kee89], although as a programming idiom rather than a formal language construct. Mixins have become a focus of active research both in the software engineering [VN96, SB98, FF98] and programming language design [BC90, BL92, Bra92, LM96, FKF98] communities. Mixin inheritance has been shown to be an expressive alternative to multiple inheritance and a powerful tool for implementing reusable class hierarchies.

1.4 Evaluation

Evaluating a programming language objectively is difficult. Languages are often designed with some problem domain and some programming style in mind. Evaluation with respect to that problem domain will naturally favor a language designed with that problem domain in mind. A common approach is to use examples that demonstrate a language's features. However such examples are not convincing if there are other features or approaches that can solve the same design problem in a better way. We do not want to evaluate our language's ability to express individual features, such as those illustrated by the common *ColoredPoint extends Point* example. This example does not show good design—a *Point* represents a location, while a *ColoredPoint* represents both a location and a color. It is *not* a specific kind of location; it is instead a pairing of locations and colors (perhaps a mapping from locations to colors). These two classes show structural similarity, but not similarity in behavior or intent. We do not consider structural similarity sufficient for inheritance because it leads to problems during program maintenance. Instead of examples of language features we will use common design problems (including those from "design patterns" [GHJV95]) and show how they can be expressed in OBSTACL.

1.5 Theory

We study the type system and semantics of OBSTACL by designing a calculus for the language (see chapter 7).[7] As OBSTACL is an extension of ML, the calculus is an extension of a calculus for ML. For the core features of OBSTACL, we present a calculus in which both classes and objects are basic constructs. Many previous calculi for object-oriented languages have provided only objects, with classes encoded as objects. The decision to include classes directly in a core calculus reflects many years of struggle with object-based calculi. There is a fundamental conflict between inheritance and subtyping of object types [CHC90, Bru94, BL95, FM95]. Encoding classes as objects is therefore troublesome. Our calculus resolves this conflict by supporting class extension without class subtyping and object subtyping without object extension. The separation between inheritance (an operation associated with classes) and run-time manipulation of objects allows us to represent objects by records and keep the type system for objects simple, involving only functional, record, and reference types. Unlike many calculi, we do not need polymorphic object types or recursive MYTYPE [AC96].

In addition to classes, mixins are represented directly in this calculus. There has been a dearth of formal calculi to provide a theoretical foundation for mixin inheritance and few attempts have been made to use mixins as the basic inheritance construct. Although mixin inheritance is easy to formalize in an untyped setting, static type checking of mixins at the time of declaration (as opposed to the time of mixin use) is more difficult. In addition, many approaches to mixins do not address the modular construction of objects, including initialization of fields. In our calculus, mixins and mixin inheritance are used to produce all classes.

Although chapter 5 informally covers the semantics and types associated with OBSTACL objects, classes, and mixins, chapter 7 provides precise rules for understanding OBSTACL's type system and the semantics of its operations.

1.6 Practice

A language may look good on paper, but without a reasonably clean and efficient implementation it does not serve the needs of programmers. In chapter 8 we describe work on prototype implementations as well as a proposed full implementation of OBSTACL, with analysis of size and space efficiency issues. The two goals of the implementation are to generate efficient code and at the same time allow for separate compilation that matches the expectations of the programmer. At times these goals conflict; we choose separate compilation over efficiency. For example, when the implementation of one object changes, examining all uses in other modules of that object can lead to more efficient code. However, we will only consider recompiling other modules when an *interface* to the object changes—this is the only time the programmer may

[7] The calculus for OBSTACL is joint work with Vitaly Shmatikov and Viviana Bono, and appeared in [BPS99].

reasonably expect other modules to be recompiled. Such a restriction on compilation may lead to lower efficiency; we will argue that because of the design of OBSTACL, many cross-module optimizations needed in pure object-oriented languages are not necessary. It is possible to build an OBSTACL compiler that supports separate compilation and produces fairly efficient code.

1.7 Summary

In this work we will review concepts related to modularity and abstractions (chapter 2), examine the uses of objects with respect to program design and maintenance (chapter 3), evaluate language features with these goals in mind (chapter 4), and design an object system that promotes good program design (chapter 5). To study object systems, we will rely on a base non-object-oriented language, ML, upon which to build an object system, without relying on ML-specific features such as type inference or ML-style polymorphism. Where chapter 4 evaluated individual language features, chapter 6 examines OBSTACL as a whole in the context of expressing common design problems. To study the analysis of object systems, we will build a calculus (chapter 7), including both an operational semantics precisely describing how programs are executed and a type system describing how values in the system are described at compile time. The efficient implementation of object-oriented languages has been studied before; we instead will focus in chapter 8 on implementation strategies for the features of OBSTACL that are not present in most object-oriented languages. In chapter 9 we look at extensions of OBSTACL as well as features that are useful in other languages but are unnecessary in the context of this system. Finally, chapter 10 summarizes the work.

Chapter 2

Concepts

In this chapter we review the concepts that will be discussed and analyzed in the remainder of this work. The motivation behind these constructions is modularity. By reducing one part of a program's dependence on another part, we can make the program more modular. The benefits of building a program out of components are both structural and temporal. The structure of a program can affect how easy it is to write and understand: smaller components are generally easier to work with than larger components. In addition, each component can be assigned to a different person, allowing larger groups of developers to work on a project. The temporal aspect can affect how easy a program is to change and maintain. Individual components can be updated, replaced, or used in new projects.

To make a program modular, we need encapsulation, abstraction, and interfaces. ***Encapsulation*** is the grouping of tightly related parts of a program into a larger component. ***Abstraction*** is the viewing of only the important aspects of a component while ignoring its details. ***Interfaces*** are the boundaries at which components interact. Typically, an interface is not symmetric: instead of describing how two components interact, it takes into account what one

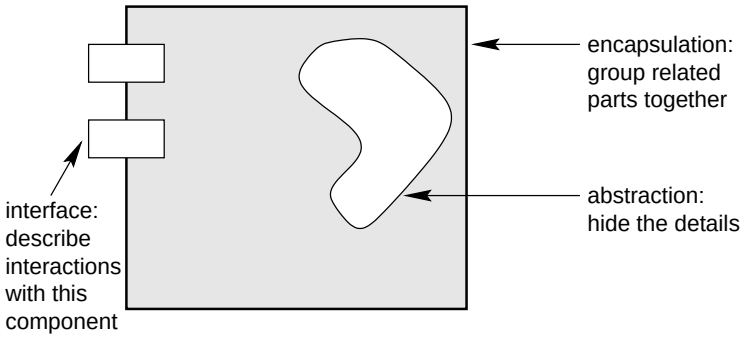


Figure 2.1: Abstraction, interfaces, and encapsulation

module (the ***provider***) is willing to ***export*** and what the other (the ***client***) needs to ***import***.

2.1 Modules

A ***module*** is a program component containing smaller pieces of a program: functions, types, and values. We can think of a module as a container into which declarations are put at compile time. Modules allow us to divide a program into separate units, which are connected together at well defined interfaces. In a modular program, the relationships among the pieces *within* a module are stronger than those *between* modules. We can look at the insides of a module without worrying about external relationships and we can also look at the relationships without looking inside the modules. Modules allow us to work with a program at a higher level than functions, types, and values.

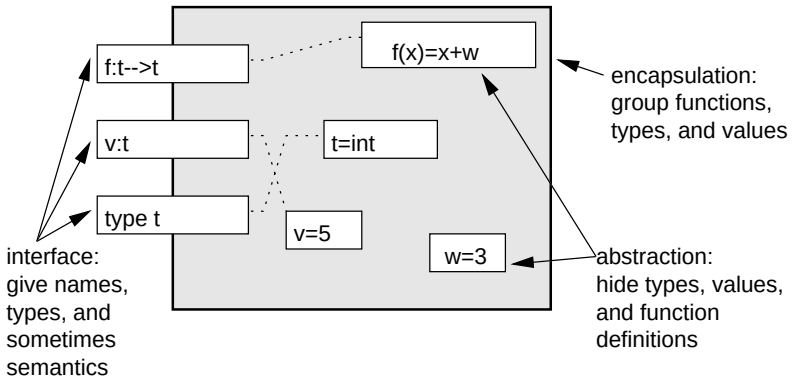


Figure 2.2: A module

2.2 Abstract Data Types

Given modules, we can build abstractions for data. An ***abstract data type*** (ADT) is a type describing a set of values without revealing the structure or representation of those values. An ***instance*** of an ADT is a member of the set of values described by the ADT. Since the representation is hidden, the module's clients cannot operate directly on values of an abstract data type. Instead, from the module we export a set of functions that manipulate these values. Of these public functions, "constructors" are those that create new instances of the ADT, "observers" are those that allow us to query instances of the ADT, and "operators" are those that allow us to manipulate instances of the ADT. With ADTs, the data (instance of an ADT) is separate from the code (functions in the module). When we want to manipulate instances of an ADT, we go back to the module that defined the ADT to find operators and observers.

2.3 Data Hiding

Interfaces can limit access to the component in two ways: ***accessibility*** and ***visibility***. Accessibility determines which parts of the program can access (read or write) elements of a structure (module, object, record, etc.). We use the C++ terminology: ***public*** means that any part of the program can access the element; ***protected*** means that parts of the program related to the structure through extension can access the element; ***private*** means that only other parts of the structure can access the element. Visibility determines which parts of the program can *see* elements of a structure. The same three levels (public, private, protected) can also be used to describe visibility. Accessibility-based protection adds a description of which parts of the program are allowed to access each item listed in the interface. The language implementation then ensures that private items are not access by unprivileged parts of a program. Visibility-based protection instead removes private items from the interface. Any items not listed cannot be accessed.[1]

2.4 Objects

An ***object*** is an entity combining data (a set of ***fields***) with operations (called ***methods***) that can be performed on it. An object is a black box with data inside and with methods forming an interface between the outside and inside of the box (see figure 2.3). The only form of communication with an object is by sending it a ***message***, which is the name of the requested operation. The object usually responds by selecting the method of the same name. The sender can then execute the method by calling it with arguments. Figure 2.4 shows the object receiving a message, selecting the appropriate method, and enabling the method call.

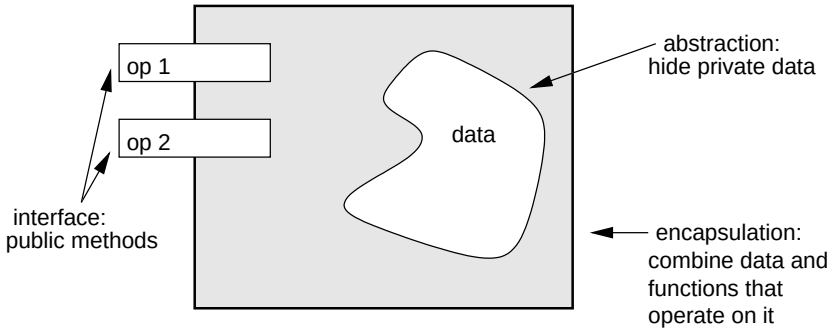


Figure 2.3: An object

As with modules, there is an interface between the object and other parts of the program. It

[1]There are two common schemes for protecting resources in an operating system. Access control lists are similar to accessibility-based protection in that a process can see any resource, but access to that resource is checked. Capability-based protection systems are similar to visibility-based protection in that any process that can see the "capability object" can access the resource; security comes from not allowing processes to see the capabilities in the first place.

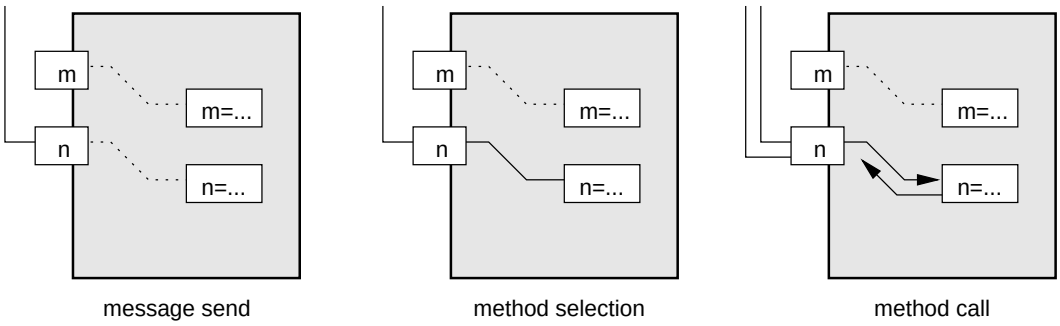


Figure 2.4: Method selection and call

typically describes the messages the object is willing to accept, and the types of the responses to a message. In some languages it also describes the behavior of the object when it receives a message.

2.5 Substitutivity

When using instances of ADTs, we must go back to the module that defines the ADT to find functions. By allowing the object (instead of its defining module) to export functions, we gain an important property called ***substitutivity***. Since we cannot "see" inside the black box object, objects that look different inside but present the same interface can be used interchangeably. Different kinds of ADT instances cannot be substituted for each other because the programmer statically specifies the module that defines operations. Given an instance of an ADT, its module must be known; therefore, data values from different modules cannot be put into a single container and treated uniformly.

For example, suppose there are two kinds of toys: Dolls and ActionFigures, each represented by a data value defined by an ADT in a module (see figure 2.5). If both are put into a toy box, and one is later taken out, then the defining module cannot be determined statically. As a result, no operations can be performed on the toy in a statically typed language.

With objects, however, the operations are located on the toy, not on the module (see figure 2.6). When the toy is put into and later removed from the box, the operations are carried along with it, so the programmer can call operations on the toy. When a message is sent to the object, the object selects the appropriate method—dolls will respond by selecting Doll operations and action figures will respond by selecting ActionFigure operations. The programmer does not have to know which kind of toy is being accessed. Substitutivity allows at run time an object to be replaced by another object with the same interface but different implementation.

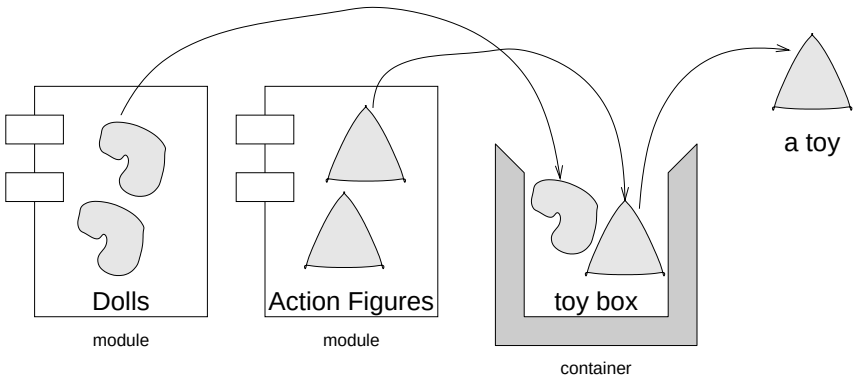


Figure 2.5: Removing a toy data value from a box

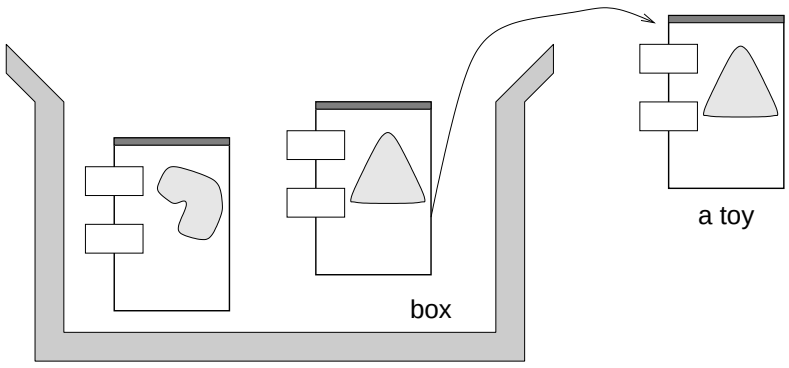


Figure 2.6: Removing a toy object from a box

2.6 Prototypes

With ADTs, the functions are defined in a module and the data is defined per value. However with objects, both functions and data are defined in each object. Data are necessarily different per object, but describing each object's functions in isolation is very tedious. An alternative is to build one object by using another object as an example. The example is called a ***prototype object***. The prototype can be extended with new methods, new fields, replacement methods, and replacement fields to build an ***extension***. We can build a family of prototype objects and use them to create normal objects.

2.7 Classes

An alternative to using prototype objects is to describe an entire family of objects at once. A ***class*** is a description of a family of objects. Each object in this family is an ***instance*** of the class. A class lists the fields and methods of each object. In addition, the class describes the routines used for object creation, called ***constructors*** and ***instantiators***, which are described in sections 5.2.4 and 5.2.5.

2.8 Inheritance

Like a prototype, a class does not have to be described in isolation. it can also be described as an extension of another class, called its ***superclass*** (sometimes called a ***base class*** or ***parent class***). The new class is called the ***subclass*** (sometimes called a ***derived class*** or ***child class***). Figure 2.7 shows the sharing relationship between subclasses and their superclass.

The subclass can ***inherit*** the definitions in the superclass. A subclass may also add new definitions or redefine some names. When a subclass redefines a name, both the old and new

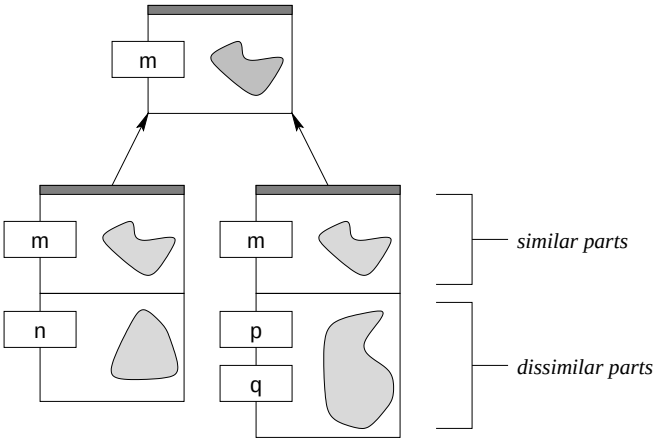


Figure 2.7: Inheritance

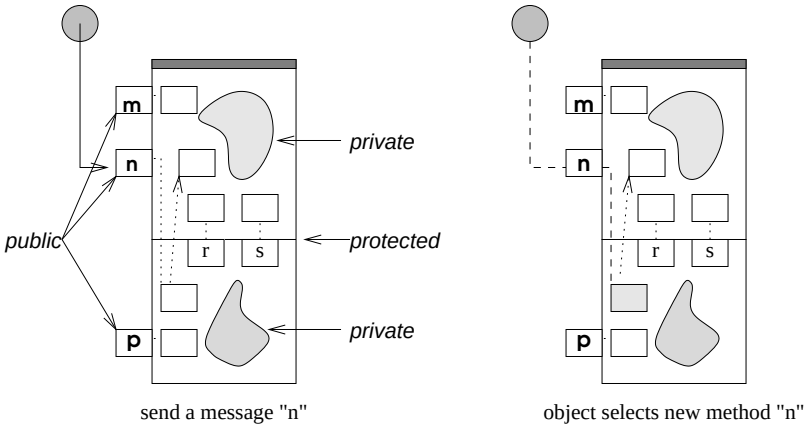


Figure 2.8: Dynamic lookup

definitions are present but the new definition is preferred when the object responds to a message (see figure 2.8). This property is called ***dynamic lookup***. The result is that the old definition is no longer accessible from outside the object.

The subclass definitions may need more access to the superclass definitions than a client needs, but not as much as the superclass definitions themselves need. With inheritance a third level of access, ***protected***, becomes useful. Private items are accessible only by the class that defined them; public items are accessible by anyone; and protected items are accessible by a class and its subclasses, but not to clients. With the distinction between protected and private, the author of a superclass can choose which items can be accessed by a subclass.

A variant of inheritance supported by many languages is ***multiple inheritance***. A class can be defined in terms of the definitions of several classes. These classes may themselves share some common implementation.

2.9 Types

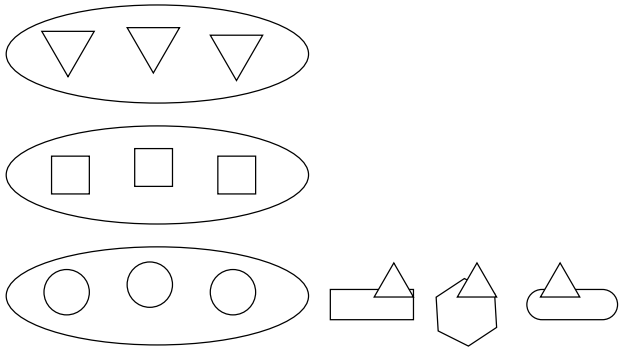
A ***type*** describes a set of values with common structure and properties. We can think of a type as a description of the "shape" of a value. One type is a ***subtype*** of another if its set of values is a subset of the other's set of values. Similarly, a ***supertype*** corresponds to a superset of values. In object-oriented programming, a *subclass* often produces a *subtype*; however, the two are distinct concepts and are not always linked.

2.10 Polymorphism

Polymorphism means "many shapes". ***Parametric polymorphism*** allows us to define program components with a type parameter. For example, we can define a triple of any type T, and can then use triples of numbers, triples of files, triples of strings, and so on. ***Subtype polymorphism*** allows us to define program components that have variants of a common, shared type.

Both kinds of polymorphism allow many shapes, but in different ways. With parametric polymorphism we choose a shape when we write the program, and at run time all the values must have that shape. In figure 2.9(a), triples can contain different shapes but all elements of each triple must have the same shape. With subtype polymorphism we specify what the values have in common, and at run time all the values can have different shapes, as long as they share something. In figure 2.9(b), the shapes share a triangular section. Subtype polymorphism cannot replace parametric polymorphism—to define containers that can accept circles, squares, and triangles, the container must accept any kind of object, but then there are no guarantees that objects removed from such a list support any interface.[2] Parametric polymorphism cannot

[2]Since Java does not support parametric polymorphism, objects removed from a list must be type-cast to recover



(a) Parametric (b) Subtype

Figure 2.9: Kinds of polymorphism

replace subtype polymorphism—if a list should contain objects of different types, then subtype polymorphism is required. These two kinds of polymorphism are orthogonal and can be combined. For example, each of the shapes in 2.9(b) could be put into a list of triangles, since the shapes share a triangular component.

type information.

Chapter 3

Program Design and Maintenance

How can one tell if a program is well-designed? We usually consider well-designed programs to be modular. However, all modular programs are not well-designed—the abstractions may be inappropriate for the problem being solved. In section 3.1 we look at the recognition and building of modular programs by examining dependencies. In section 3.2 we look at four kinds of abstraction and how they relate to our modularity goals. We then turn to issues with objects in sections 3.3, 3.4, and 3.5. In section 3.6 we look at more specific types of dependencies, which lead to design patterns.

3.1 Modularity and Dependencies

To discover modularity we look at a program's ***dependencies***, or ways in which one program component may need to be modified to account for changes in another program component. Typically a program has countless dependencies; here we look at two common forms:

- Program component A depends on program component B. If B's names, types, values, behavior, or structure changes, then A will have to be examined and possibly modified. For example, the `fopen` function in C depends on the representation of files (`FILE*`). If the `FILE` structure is altered, the `fopen` function will likely need changes.
- Program component A depends on a set S of program components. If components are added to or removed from S, then A will have to be examined and possibly modified. For example, the + function in Lisp depends on the set of numerical types. If a new numerical type is added, the + function will likely need changes.

Note that the dependency relation is transitive: if the type of disk can affect the `FILE` structure, and the `FILE` structure can affect `fopen`'s implementation,[1] then the type of disk may affect `fopen`'s implementation. The number of dependencies in a program can tell us something about its modularity: in general, reducing the number of dependencies makes a program more modular.

The question is then: how can we reduce dependencies? The primary tool for reducing dependencies is abstraction. If component A depends on component B, we can create an interface B_I for B that describes an abstract view of B but not the details of B.

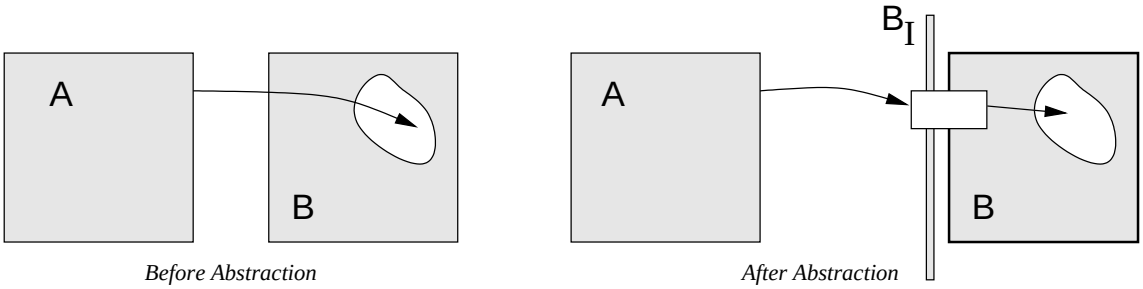


Figure 3.1: Creating an interface

Before abstraction, A depends on B, so the author of A must understand B. The author of B has no special responsibilities. Changes can be made to A without affecting B. However, changes to B can affect A. After abstraction, A depends on B_I and B depends on B_I, so the authors of A and B must understand B_I. Changes can be made to A or B without affecting the other component. However, some changes to B may require changing B_I. Changes to B_I affect both A and B.

The original version places more burden on the author of A: he has to understand both modules and react to any change in B. The modular version shifts responsibility to the author of B: he has to build the interface B_I, has to understand both B and B_I, and should avoid changing B in a way that requires changes to B_I. The author of A has to understand only B_I, not all of B.

Which version is better? The original program is smaller. It takes less effort to write if a single person is writing both A and B. The modular program is only slightly larger, but harder to write. It can take a lot of analysis to determine exactly which parts of B should be exported through its interface. However, once written, the program is easier to understand—we can understand the important aspects of B by reading B_I. In addition, it is easier to change—most changes to B do not affect B_I, so A does not have to be reexamined. Once we introduce other

[1] Note that it is possible that the changes to the type of disk do not require changes to `fopen`. We have defined dependencies as *possible* changes, not actual changes. Another way to look at dependencies is to think about which parts of the program must be examined (and possibly changed).

components that depend on B, it is clearer that the modular program is better. The cost of creating B_I is borne only once, but the benefit of depending on B_I instead of B is gained once for each component using B. As the program grows in size, the benefits increase while the costs remain fixed. The abstraction B with interface B_I can also be reused in other programs.

We have only looked at breaking the dependency of one component on another. In the next section we will explore abstractions that allow us to break dependencies on a set of components.

3.2 Abstractions

What form of abstraction should we use? We can reuse parts of the program that *don't change*; parts that change are updated or replaced. To benefit the most from modularity, the interfaces should rarely change. Therefore we should put what will stay the same in the interface, and hide what will change in the implementation. Discovering commonalities is part of the design process [Cop99]. We must consider what kinds of changes may be needed, how often those changes are required, and how expensive these changes will be. In a large programming project, it is also important to assign responsibilities. Every program component and interaction between components should be the responsibility of some developer. If there is an interaction for which no one is responsible, then unexpected behaviors can be produced. One case we will consider is a base system extended in two different ways (by different people) and combined by a fourth person; we would like the combination to work without the fourth person examining implementation details of all three modules to ensure there are no unexpected interactions. In this section we examine these issues using four different abstraction mechanisms.

3.2.1 Abstract Data Types

We start with the simple case: neither the data representation nor the set of operations needs to change in the future. Shown in figure 3.2 is an ADT for Points:[2] In this example there are two data variants (cartesian and cylindrical) and three functions. Clients depend on the set of functions so changes to that set will affect clients. Clients depend on the defining module but not on the set of data variants. The data variants or set of functions can be changed, but only by the author of the ADT.

With restrictions come knowledge: if the set of data variants cannot be extended from outside the ADT, the author of the ADT can rely on any Point being either cartesian or cylindrical—it cannot be some new variant. This knowledge allows for functions like `add()`: all combinations of variants can be accounted for. It also provides for security: if a client has a Point, he can be secure in the knowledge that it came from this ADT, and is not some kind of

[2]For Point to be useful, we would need to make the X and Y coordinates accessible. However, to demonstrate some issues with abstraction, we make them private.

```
signature PointSig = sig
      type Point;
      val Cart : real * real * real → Point
      val Cyl : real * real * real → Point
      val magnitude : Point → real
      val mult : Point * real → Point
      val add : Point * Point → Point
      (* axiom: magnitude(mult p x) = (magnitude(p) * x) *)
end

structure PointStruct = struct
      datatype Point =
                    Cart of real * real * real
                  | Cyl of real * real * real
      fun magnitude( Cart(x,y,z) ) = √(x² + y² + z²)
        | magnitude( Cyl(r,θ,z) ) = √(r² + z²)
      fun mult(p,v) = ...
      fun add( Cart(x1,y1,z1), Cart(x2,y2,z2) ) = Cart(x1 + x2, y1 + y2, z1 + z2)
        | add( Cart(x1,y1,z1), Cyl(r2,θ2,z2) ) = ...
        | add( Cyl(r1,θ1,z1), Cart(x2,y2,z2) ) = ...
        | add( Cyl(r1,θ1,z1), Cyl(r2,θ2,z2) ) = ...
end
```

Figure 3.2: Point abstract data type

impostor.

In the above definition, the ADT is built on top of union types. An alternate approach is to build the ADT on top of object types and a private class hierarchy. The ADT would wrap a private base class, `PointImpl`, plus two private derived classes, `CartPointImpl` and `CylPointImpl`. The ADT functions would simply forward the requests to the underlying object.

The remaining three forms of abstractions we consider will give us the ability to extend the set of data variants, the set of functions, or both. With extension however we lose knowledge—we can no longer rely on a fixed, known set of variants.

3.2.2 Objects

When we want extensibility of data variants, we turn to object-oriented programming. Like ADTs, we would like to have several data variants and several functions. Clients can depend on the set of functions. Unlike ADTs, clients of objects do not depend on the defining module. As a result, we have substitutivity (see section 2.5): it is possible for clients to transparently use new variants defined in a different module. With this sort of flexibility we lose knowledge of where an object was defined. Such knowledge is crucial for defining functions like `Point.add`, which

needs to know the data variants of both arguments. Standard forms of object-oriented programming let us know one variant only, so "binary" methods like `Point.add` cannot be written properly. There have been many attempts to solve the binary method problem [BCC+95]. Binary methods, in which the data variants *must* be known, are fundamentally incompatible with substitutivity, for which the data variant *must not* be known. Therefore it is not surprising that none of the solutions to this problem are satisfactory [BCC+95]. One must choose whether binary methods or substitutivity is more important; one cannot have both.

Thought Experiment

Suppose we are using Point objects in a program. If we create a class MyPoint that looked exactly like Point, would we be able to use MyPoint objects in place of some of the Point objects without changing the program (*i.e.*, does substitutivity hold)?

Having seen that Points (the most common example in object-theory literature) are better as ADTs than as objects, we should naturally ask for what situations objects are better than ADTs. An object abstraction works best when we want substitutivity—we want clients to be independent of the set of data variants. The Unix file system is a good example: the set of operations is fixed (`open`, `close`, `read`, `write`) and clients are not dependent on the data variant being used. Substitutivity is easy to demonstrate: if a program uses standard input (keyboard) and standard output (screen), we can use Unix *redirection* to make the program read from a file and write to the printer. We have replaced the keyboard and screen objects with file and printer objects without changing the program. Clients can work with files, pipes, network sockets, printers, tape drives, keyboards, screens, modems, and other devices. We also have extensibility: a new kind of object (such as an infrared receiver) can be added to the system, and as long as it uses the same interface, the object can be used without changing the program.

3.2.3 Union Types

Objects give us extensibility of data variants while essentially keeping the set of functions fixed.[3] Union types instead give us extensibility of functions while keeping the set of data variants fixed. We used a union type inside the Point ADT to give us two variants, cartesian and cylindrical. The Point ADT encapsulated a union type with a fixed set of functions, but we could instead use the union type directly to avoid fixing the set of functions.

When should we use union types? A union type abstraction works best when the set of data variants does not change and we want to add functions in a modular fashion. Clients can depend on the set of data variants but are dependent only on the functions they want to use, and are *not* dependent on the set of functions available. A good example of union types is the internal data structures of a compiler. A compiler stores an abstract syntax tree to represent

[3] Subtyping makes it possible to have more functions in some data variants. However here we are more concerned with adding functions to work on *all* variants.

the program. The nodes in the tree represent different kinds of constructs in the language, such as if/then/else, function calls, and variable declarations. The union type can contain one data variant for each language construct. One function can be created for each phase—parsing, debugging, profiling, visualizing, optimizing, and generating code. Adding a new optimization phase involves creating a new module and defining the optimization functions. It does not require changes to the existing modules. In contrast, had the parse tree been structured in an object-oriented fashion, a new optimization phase would involve modifying each of the existing modules to add a new method.

Union types make it difficult to extend the set of data variants but easy to extend the set of functions. In contrast, objects make it easy to extend the set of data variants but difficult to extend the set of functions, and ADTs make it difficult to extend either. In addition, union types allow code to inspect the variants being used. For example, `Point.add` invokes different code depending on a *pair* of arguments. Such functionality is also needed for optimization passes—a constant folder for example needs to know that both the left and right children of a tree are constants, and that the node at the top is an arithmetic type.

3.2.4 Multimethods

If we want both the set of data variants and the set of functions to be extensible, we can turn to an alternate object-oriented model used in CLOS [Kee89], Dylan [Com92], and Cecil [CDG97]. ***Multimethods*** are methods that are not encapsulated with data, but still enjoy the dynamic-lookup property—when a message is sent to an object, the most appropriate method is called. The "multi" in the name "multimethods" refers to methods being chosen based on more than one object. The technique is also called ***multiple dispatch***: messages are sent to a group of objects, and the system chooses the most appropriate method for the group.

There are two aspects of multimethods that make them interesting here. First, they can distinguish data variants of more than one argument. Second, they do not encapsulate data with functions.[4]

- Like union types, the multimethod paradigm gives up substitutivity by allowing any function to determine which data variant is being used. With this ability we can write binary methods like `Point.add`, which look at the data variants of two arguments.
- Like union types, the multimethod paradigm gives up encapsulation by separating the data from the functions. With this separation we gain the ability to extend the set of functions without modifying the definition of data variants.
- Like objects, we can define the functions for each data variant separately, so we can extend the set of data variants without modifying existing functions.

[4]Instead, as with ADTs, data and functions are encapsulated in modules.

	Cyl	Cart	Spher
Cyl	Bob	Bob	?
Cart	Bob	(orig)	Carol
Spher	?	Carol	Carol

(a)

	instr A	instr B	instr C
phase 1	(orig)	(orig)	Bob
phase 2	(orig)	(orig)	Bob
phase 3	Carol	Carol	?

(b)

Figure 3.3: Who defines functionality?

- Like objects, the multimethod paradigm gives up a central list of data variants. A decentralized definition of variants is more difficult to reason about than a centralized definition.

With multimethods we give up some encapsulation and substitutivity, but we gain the ability to extend both the set of data variants *and* the set of functions. Unfortunately it can be difficult to combine separately developed extensions into a single program. For instance, suppose in the Point example, Bob extends the system with cylindrical points, along with functions `add (Cart, Cyl)`, `add (Cyl, Cart)`, and `add (Cyl, Cyl)`. Carol extends the system with spherical points, along with functions `add(Cart,Spher)`, `add (Spher, Cart)`, and `add(Spher,Spher)`. The chart in figure 3.3(a) shows the code that is defined originally (marked *(orig)*) and which cases are defined by Bob and Carol. With either extension alone, `add` can handle all four cases, but when we combine the two extensions, there are nine cases, and two of them, `add (Spher, Cyl)` and `add (Cyl, Spher)`, are left undefined. Extensions involving binary methods cannot be combined easily. It is not the fault of multimethods in particular. This is an inherent limitation: the goal of modular extension (cylindrical points module defines addition involving cylindrical points) and the goal of completeness (binary methods such as addition are properly defined for all cases) are in conflict.

There are also limitations when combining a data extension with a function extension. Suppose to our compiler data structures example from the previous section, Bob adds a new kind of instruction. Independently, Carol has developed a new optimization phase. Combining these extensions is problematic: neither Bob nor Carol has described how the optimizer should treat the new instruction. Again, we see in the chart in figure 3.3(b) that each extension covers all six cases needed to deal with the original four plus the extension, but the combination of two extensions leads to nine cases, leaving one case undefined. Modular extension and completeness are at odds.

There is a case for which the entire matrix for `add` does not have to be defined. If the function's semantics are the same regardless of the data variants of the arguments, and if a single definition can be written to handle any arguments, then that definition can be used when more specific definitions have not been written. In the case of `add`, there would be a definition `add(Any,Any)` that could handle any types of points, and more specific definitions

	Client depends on...		Implementors can...		
	set of functions	set of data variants	combine extensions	extend functions	extend data variants
ADT	Yes	Yes	n/a	No	No
Objects	Yes	No	Yes	No	Yes
Union types	No	Yes	Yes	Yes	No
Multimethods	No	No	No	Yes	Yes

Table 3.1: Summary of abstractions

`add(Cart,Cart)`, `add(Cyl,Cyl)`, and `add(Spher,Spher)` provided by the authors of `Cart`, `Cyl`, and `Spher`, but no one is required to provide functions such as `add(Cyl,Spher)` because the generic `add(Any,Any)` can handle that case. One may wonder, if `add(Any,Any)` can deal with any kind of point, why write specific versions like `add(Cyl,Cyl)`? To preserve substitutivity, `add(Cyl,Cyl)` should do the same thing as `add(Any,Any)`. However, it is possible that `add(Cyl,Cyl)` can be implemented more efficiently than `add(Any,Any)`. This is especially true for container classes, where access to the container class internals can lead to more efficient code. In this work however we are concerned primarily with features for program design, rather than for efficiency.

3.2.5 Comparison

The above sections described four forms of abstraction. Table 3.1 summarizes the key properties of these four. Which one(s) should a language support?

There is evidence that all four of these abstractions are useful in practice. When the appropriate abstraction is not available in the language, programmers often create it through discipline and unusual program structure. For example, the Xt toolkit for The X Window System [You89] and the Presentation Manager library from IBM [Pet94] both introduce the notion of classes and subclassing to create object-oriented hierarchies in C. The eXene library for SML/NJ has a structure similar to a class hierarchy. For ML-like languages, Philip Wadler has proposed *views* [Wad87], a way to make union types more like objects while at the same time preserving the extensibility of the set of functions. For object-oriented languages, the Visitor design pattern [GHJV95] transforms object-oriented language constructs into a union-like structure, allowing for the extension of the set of functions ("visitors") while restricting the set of data variants. Type query constructs such as `dynamic_cast` in C++ and `instanceof` in Java also give some of the flavor of union types. The "double dispatch" technique is a way of implementing multimethods on top of conventional objects. In languages with multimethods, we see the opposite—it is more common to program in the union type or conventional object-oriented styles than to take full advantage of multiple dispatch.

Given any of these four abstraction techniques, programmers try to emulate another. This

is evidence that all four forms are needed in practice, and that no one form is the best choice for every problem. ML already supports ADTs and union types. Multimethods are incompatible with our goal of minimizing dependencies and preserving substitutivity. OBSTACL adds objects but not multimethods.

3.3 Objects vs. Values

More evidence that objects are not universal comes from the differing treatment of objects and values of algebraic types (see section 1.2). Objects have state and identity.[5] An object's state can change over time while its identity stays the same. For example, an automobile's state may be {loc = "Princeton, NJ"} at one time and {loc = "Pittsburgh, PA"} at a later time, but it is still the same automobile.[6] Two automobiles may have the same state but different identities. Values of algebraic types on the other hand have neither state nor identity. They can be treated as the *names* of objects. Names can be copied to create a new name for the same object, and compared to ask whether two names refer to the same object. For example, $\{1, 5, 3\}$ and $\{3, 1, 5\}$ are two names for the same set, and $^3/_5$ and 0.6 are two names for the same number. Copying an object such as an automobile is a non-trivial operation, and is often called ***cloning***. Unlike copying of values, a clone of an object has its own identity distinct from the original object. Two objects can look the same but be different. Two values can look different but be the same. Sections 4.1.3 and 6.1.1 examine this issue of "sameness" (equality) in more depth.

We might want to say that values are immutable and objects are mutable. However, conceptual immutability and implementation constraints are sometimes at odds, so values may have a mutable implementation for efficiency. For example, sets are conceptually immutable, but most implementations offer mutable sets for better efficiency. Although an object may be immutable in implementation, it is conceptually potentially mutable. For example, a student may be represented as an object with an immutable "name" field. Although changing the name is not supported in the program, a student conceptually could change his or her name in real life: the field is conceptually mutable but immutable in the implementation. The value/object distinction also affects distributed systems, multithreading, and serialization [BRS+98].

Objects and values are complementary and can be used together in the same program. Fundamental properties such as identity, copying/cloning, mutation, and comparison are different for objects and values. In OBSTACL we keep them distinct instead of choosing one over the other.

[5] Identity is the more fundamental aspect of objects [Pit93], but identity leads to objects having state, even if that state is not directly mutable in the language.

[6] Philosophers may argue this view of the world.

3.4 Polymorphism

The abstractions discussed in section 3.2 are sometimes called "polymorphic". Subtype polymorphism, which is associated with object-oriented programming, has been show insufficient to express parametric polymorphism [Str97]—more evidence that the object-oriented approach is not universal. At the same time, parametric polymorphism allows only compile-time variance of types and cannot express the run-time variance provided by subtype polymorphism. Both are required to express common programming structures. Consider for example a mutable array of objects. We can write this array's type as $\mathrm{Array}[\sigma]$, where σ is the type of the object contained in the array. Suppose B is a subtype of another object type, A. Should $\mathrm{Array}[B]$ be a subtype of $\mathrm{Array}[A]$? No. One operation on $\mathrm{Array}[A]$ arrays is to insert an A object into it. But $\mathrm{Array}[B]$ does *not* support insertion of A objects; it contains only B objects. We cannot say that $\mathrm{Array}[B]$ and $\mathrm{Array}[A]$ are related by subtyping polymorphism (in particular, one cannot say that $\mathrm{Array}[B]$ is a subtype of Array or of $\mathrm{Array}[\mathrm{Object}]$); they are instead related by parametric polymorphism. In dynamically typed languages, the two forms of polymorphism (parameteric, often used in containers, and subtype, used for object types) are not distinguished.[7] In a statically typed language, both are necessary to express programs.

3.5 Class Hierarchies

Section 3.2 included analysis of dependencies involving objects: the users of an object depend on the set of public functions it contains, but not on the set of data variants. As a result, new data variants can be defined without affecting program components that use objects. However, components that *create* objects are dependent on the mechanism used to create objects. In OBSTACL classes are used to create objects. (Section 4.2 explains why we chose classes over prototypes.) In addition, classes can be used to create subclasses. Thus there are three kinds of audiences:

1. Components that define subclasses depend on the superclass. They depend on the superclass's constructor, protected definitions, and public definitions. However they should not depend on the superclass's private definitions. For OBSTACL we impose stricter conditions: components that define subclasses should not depend on the set of ancestors of the superclass, nor should they depend on whether definition was inherited directly from the superclass or indirectly from one of its ancestors. Given these restrictions one can consider the ancestor graph to be an implementation detail that can be changed without affecting subclasses.

[7] Such a distinction of types is not unique to object-oriented programming. Lists (a varying number of elements of a fixed type) and tuples (a fixed number of elements of varying types) are the same in a dynamically typed language, but distinct in a statically typed language.

2. Components that instantiate a class depend on that class's object creation policy (see section 3.6.1) and public definitions. They should not depend on the protected or private definitions, nor should they depend on the class's ancestors.

3. Components that use instances of a class depend on the public interface defined by that class.

In each case the component should depend on what the class or object provides, not on how it is defined. For each audience there is a dependency in most object-oriented languages that we attempt to break in OBSTACL:

- Subclasses depend on superclasses. We introduce parameterized inheritance (see section 5.3) to allow subclasses to be independent of the superclass and its ancestor graph. However, they still depend on the interface exported by the superclass.
- Object creators depend on the creation policy. We introduce instantiators (see section 5.2.5) to allow object creation to be independent of the creation policy. However, they may still depend on properties of the creation policy.
- Object users depend on the class: We use structural object types (see section 5.1.3) to make object users independent of the class of the object. However, they may still depend on the behavior of the object.

As dependencies are broken, a program can be made more modular. For example, if a class is not dependent on the exact implementation of its ancestors, the class can be placed in an application and the ancestors can be placed into a dynamically loaded library (DLL). If the class does not depend on the implementation of the ancestors, the DLL can be upgraded (changing some implementation data structures to more efficient ones, without changing the interface) without causing link incompatibilities. Run-time dependencies, such as DLLs, are further discussed in chapter 8.

3.6 Design Patterns

In section 3.1 we saw the general pattern of increasing modularity by creating an abstraction and depending on an interface but not the implementation. If we refine "A depends on B" to "A depends on some aspect of B" then we can look at more specific ways of breaking these dependencies. In the popular literature these are often called Design Patterns. (Note however that not all design patterns are about breaking dependencies.) In our discussion of design patterns, we will mostly use well-known design patterns [GHJV95]. Creational patterns address the ways in which objects can be created. Structural patterns address the ways in which objects are related to each other. Behavioral patterns address the ways in which objects respond to messages.

3.6.1 Creational

Some classes have a policy on how and when objects are created. As mentioned in section 3.5, object creators may not need to depend on the creation policy. Many creational design patterns address this dependency by answering questions about object creation:

1. Singleton: (*How many* objects to create) Only one instance of the class should be created, no matter how many components want an instance of the object. Without this pattern, program modules that want to create an instance must communicate to decide which module will create the instance; then that module must give an object reference to the other modules. All modules creating objects of this class unnecessarily depend on each other.

2. Remote Creation: (*Where* an object should exist) An object may reside on another machine or in a separate process. Without this pattern, program components depend on whether the object should be created locally (and accessed through method calls) or remotely (and accessed through a network interface such as remote procedure call). Access to the object can be done with the *Remote Proxy* pattern (see section 3.6.2), as long as all access to the object is through an interface and no direct access is allowed.

3. Delayed Creation: (*When* an object should be created) When an object is expensive to build and may not be needed immediately, return a reference to the object but delay the actual creation of it. Without this pattern, the program may create objects to build a larger structure (*e.g.*, a graph) but those objects may not actually be needed, so resources are wasted. The object can be constructed using the *Virtual Proxy* pattern (see section 3.6.2).

4. Object Cache: (*Whether* an object should be created) When possible, an existing object is used instead of creating a new one. Without this pattern, a program component may make unnecessary instances that behave the same as those that been already been created. The object can be designed using the *Flyweight* pattern [GHJV95].

5. Prototype: (*How* an object is initialized) An object can be initialized with the state of another object (through cloning), similar to the approach used in prototype-based languages. Without this pattern, program components need to share initial values for new objects. The pattern encapsulates these initial values into an object.

6. Virtual Constructor: (*Which kind* of object to create) Based on arguments to the constructor, choose a class to instantiate. Without this pattern, each object creator would need to know the rules for choosing a class and would depend directly on the set of classes that could be chosen.

In addition to patterns in which the class sets a creation policy for its instances, there are patterns in which a third party contains a creation policy:

1. Factory Method: A superclass leaves to its subclasses the decision of which classes to instantiate for sub-objects. Without this pattern, the superclass chooses how to create its sub-objects, and the subclass has no influence over the choice.

2. Abstract Factory: Entire class hierarchies are chosen to create a set of objects. Without this pattern, the choice of class hierarchy is fixed and cannot be changed without modifying all components that create instances of classes in the hierarchy.

When the class defines an object creation policy, it must provide functions that implement the policy. The user must call those functions instead of creating objects directly. OBSTACL uses instantiators (described in section 5.2.5) to encapsulate object construction and the creation policy. Users must call an instantiator to request a new object; otherwise they might bypass the object creation policy. Instantiators can also be used as ordinary functions, so patterns like Abstract Factory can be built by collecting instantiators from classes in a hierarchy. Instantiators are a good tool for expressing many creational design patterns.

3.6.2 Structural

Objects are rarely used alone; they usually exist in relationships with other objects. Structural design patterns address dependencies between objects.

1. Adapter: Use objects from one class hierarchy H_1 when objects from a different class hierarchy H_2 are expected. Without this pattern, program components that expect objects from H_2 would have to be modified to use objects from H_1. With adapters, those program components can use objects from H_2 without depending on H_2. (Note however that if the component needs to *create* objects from H_2, the Abstract Factory pattern from section 3.6.1 is needed to break the dependency.)

2. Decorator: Add or modify functionality to a set of objects without creating new subclasses. Without this pattern, new subclasses must be created for each class being extended.

These two patterns have another advantage: they allow choices to be made or changed at run time. In chapter 6 we will see how parameterized inheritance (mixins) help us implement these patterns and also allow us to choose to make decisions at compile time (by building new classes) or at run time (by building new objects).

1. Remote Proxy: Use an object without knowing whether it is local or remote. Without this pattern, the program component must know whether an object is local or remote,

and access the object in different ways (message sends vs. a networking interface such as remote procedure call).

2. Virtual Proxy: Use an object that may not have been fully built yet. Without this pattern, the program component must know whether an object has been fully built, and build the object before it needs all of its services. It must also notify other program components that use this object that the object has been built and that it should not be built again.

3. Composite: Treat objects and groups of objects uniformly. Without this pattern, an object and a group of objects are treated differently, so there may be places where a single object can be used but a group cannot, because the component using objects depends on there being just one object.

These three patterns are about treating one structure (a remote object, a placeholder object, or a group of objects) just like an "ordinary" object. Structural subtyping can help us express these patterns without using inheritance and can also allow us to add these patterns to an system without modifying the existing classes.

3.6.3 Behavioral

Once an object is created and its relationships set up, we still may want to look at its behavior. Many design patterns address the behavior of objects.

1. Iterator: Provide an object to traverse a sequential container. Without this pattern, program components access a container directly. By depending on the container, code cannot be reused for other containers or for sequences that are not a container.

2. Memento: Encapsulate internal data that must be saved and restored. Without this pattern, a program component must be able to query and set the object's state, including private data. Both of these abilities reduce abstraction.

3. Visitor: Provide a way to extend the set of functions that can query the data.

The dependencies eliminated by these patterns are not specific to object-oriented programming. Iterators can be written in non-object-oriented libraries [Cop99, Aus98]. Mementos can be written using opaque types [MTHM90]. The extensibility provided by the Visitor pattern is available in union types. We will not address directly these forms of dependencies in our study of objects.

3.7 Summary

Programs can be made more modular by dividing them into components and defining interfaces between components. Reducing the number of dependencies makes the program easier to

understand and change. OBSTACL offers both non-object abstractions (ADTs, union types) and object abstractions; each has advantages. Design patterns express common structures that are found in real systems, and thus are useful for studying dependencies that arise in these systems. In OBSTACL, interface types, instantiators, and parameterized inheritance reduce dependencies when using objects. In chapter 6 we will examine the use of these features with respect to design patterns.

Chapter 4

Language Space

Our goal is to add objects to ML while supporting modular programming techniques and remaining consistent with the flavor of ML. In this chapter we examine design alternatives and our choices for OBSTACL. First we consider how objects are used: message sends, updates, and comparison. Then we consider how objects are created, and choose classes as a base construct for creating objects. Next we look at ways of creating classes using inheritance. Finally we examine the object initialization process, using constructors and instantiators.

4.1 Using Objects

4.1.1 Selection

Given an object, the most important operation one can perform is to send it a message. How should it respond? There are several options:

- Select: The object can select a method and return it.
- Select and Execute: The object can select and execute a method.
- Asynchronous Execute: The object can add the message to a queue and return. Messages in the queue are processed in a separate thread of execution.
- User-Defined: The object can execute user-defined code to respond to the message.

"Pure" object-oriented languages (*i.e.*, those in which everything is an object) usually do not return unexecuted methods, which are essentially functions, not objects. Most languages do not address concurrency of objects, or at least do not make each object a separate thread by default. Strongly typed languages do not offer user-defined message response because of typing issues. OBSTACL is a strongly typed language that does not address concurrency issues, so the options are to select and execute a method or to only select a method.

Selecting and executing methods is appealing because it gives the object the option to respond to all message sends. Field access and method call are given the same syntax—`a.x` for object `a` and field or method `x`—so it is easy to turn a field into a method or vice versa. However if the method takes parameters the syntax is `a.x(...)`. The use of `x` to invoke functions with no parameters and `x(...)` to invoke functions with parameters is used in languages like Pascal. ML and C however use `x()` to invoke functions with no parameters, so that `x` alone refers to the function itself and not a call to it. To be consistent with ML, we chose to make `a.x` refer to the method selection alone, and use `a.x()` to call the method.[1] Keeping the two operations (selection and invocation) separate allows for more flexibility. For example, the Xt library for The X Window System uses the notion of "callbacks", which are essentially functions; the InterViews library [LCI+92] is object-oriented and goes farther, by making callbacks an ⟨ object, method ⟩ pair—exactly what selection without invocation provides. The Command design pattern from [GHJV95] is intended to provide something similar. In addition to the program design issues, there is a language design issue—treating selection and invocation separately allows us to use the existing ML function call mechanism instead of introducing a new feature for method calls. Thus separating the two both increases flexibility and simplifies the language.

4.1.2 Updates

> It is a great mistake to think that an object can be purely functional.
>
> –*The Last Neanderthal* (Discovery Channel)

The second most important operation on objects is the update of fields or methods. There are three choices to be made:

1. Are updates imperative or functional? Imperative updates alter the object's state without changing its identity. For example an automobile's location may be changed from "Pittsburgh, PA" to "Boston, MA" without making a new automobile object. Functional updates create a new object with all fields and methods copied except one, which receives a new value. For example, an automobile in "Pittsburgh, PA" could be updated by creating a new automobile object that carries the fields (such as color, size, model) and methods of the old object except its location would be "Boston, MA". Any objects that refer to the automobile must be updated as well to refer to the new automobile. Any objects that refer to those object must also be updated, and so on.

   Functional updates are conceptually more complicated than imperative updates. Since other objects must be updated, the fact that an object is being updated must be reflected

[1] Python [Lut96] also makes this distinction, and calls the selected method a *bound method*.

in the object's interface, reducing abstraction and sometimes destroying substitutivity. In addition, creating new objects via updates bypasses the object creation policy set by the class (see section 3.6.1). Imperative update can be hidden from the object users, and more closely corresponds to real-world objects. Furthermore, evidence from object calculi and implementation of object-oriented languages suggest that functional updates are more expensive. For OBSTACL we chose imperative updates because they are simpler, better for modularity, and more closely correspond to the update of physical objects.

2. Can updates be made from outside the object? Updates made directly by the user of an object can lead to more flexibility and convenience. However, those updates may not preserve object invariants. For example, an object representing a finite state machine may require its state to be one of $(\text{false}, 0)$, $(\text{true}, 0)$, or $(\text{true}, 1)$. The user may update the state to $(\text{false}, 1)$, putting the object into an invalid state. In OBSTACL, the state is generally hidden from the user, and updates must be made through method calls. However, an object may export to the user the ability to make updates, allowing for user updates in cases that require it.

3. Can all fields be updated, or only some?

   In "mostly functional" languages such as ML and Scheme, assignment of new values to existing variables is discouraged. In ML, variables are immutable by default, and can be made mutable by using reference cells ("refs"). A ref is a box into which a value can be placed, and from which a value can be extracted. ML refs are first class values, and can be placed into data structures such as objects. C++ also distinguishes between mutable and immutable with the "const" qualifier, but makes mutability the default. In these languages, initialization must be a separate operation from assignment; the latter works only on mutable variables. For consistency with ML, we extend ML's treatment of variables and record fields to apply to object fields as well: each field is immutable by default, and can be made mutable by using a ref.

4. Can methods be updated, or only fields? Many object calculi support updates of both fields and methods. Most method updates are made to build objects, but there are also cases where updates to existing objects make sense. (See for example Cardelli's calculator object in [AC96] or the State pattern from [GHJV95].) However, method updates are more difficult to support in the presence of subtyping [AC96]. In OBSTACL, we use classes (which support method override) to build objects, so method updates in objects are uncommon. We chose not to support method update; instead, we can use updatable fields with function values to achieve a similar effect (see section 9.1.4).

In OBSTACL we support imperative updates of fields but not methods. We do so by utilizing ML reference cells ("refs"). Thus we can support field update without additional machinery.

4.1.3 Equality

Given two objects or values, one may want to ask *are they the same* (*i.e.*, are they equal)? What does this mean? We say a equals b if replacing one by the other has no effect on any program. This view suggests that equality and substitutivity are the same notion. However, substitutivity tells us that we can replace a by b and the program may work *differently* but will still work correctly. Equality is a stronger relation—it tells us the program will behave *identically* after the replacement. The equality relation from mathematics is reflexive, symmetric, and transitive, so we would prefer to preserve those properties in our programming language.

We consider four forms of equality:

1. Shallow Equality. Also called *pointer equality*, shallow equality compares only whether two expressions yield the same object in memory, not whether the objects have the same structure and contents. This form is fastest and simplest to implement, but is not always appropriate. It is also reflexive, symmetric, and transitive.

2. Deep Equality. Equality that takes into account a value's references to other values is called deep equality. For example, the list $[1, 2, 5]$ is equal to the list $[1, 2, 5]$ because the lists have the same length and the corresponding elements are equal. Deep equality can be significantly slower than shallow equality, and can be complicated by cycles in data structures. It is reflexive and symmetric but may not be transitive in the presence of subtyping (see below).

3. User Defined Equality. Since it is not always clear which form of equality is better, some languages allow the programmer to define a comparison function. User defined equality is typically slower than shallow equality but faster than deep equality, and can lead to anomalous behavior if the programmer is not careful (*e.g.*, the user's function may not be symmetric). User defined equality appears to be essential to handle cases where the same conceptual value may be represented in many ways. For example, the sets $\{1, 2, 5\}$ and $\{5, 1, 2\}$ should be considered equal but are not equal using either shallow or deep equality.

4. Equality At A Type. In the presence of subtyping, we may need a variant of deep or user-defined equality: compare objects at a common supertype of a and b. For example, $\{x = 3, y = 5\}$ and $\{w = 7, x = 3\}$ have a common supertype $\{x : \text{int}\}$ so we compare only the x field. However, it is not clear how to compare *private* data in objects. At any one type, equality is reflexive, symmetric, and transitive, but that one type is the user-visible type and does not include private and protected data items.

Of these four forms of equality, only shallow equality is compatible with our substitutivity and data hiding goals. ML provides shallow equality for mutable values (refs), deep equality for

most immutable values, and no equality for functions and certain other types. Deep equality is more appropriate for immutable values; shallow equality seems better for potentially mutable objects. In section 6.1.1 we will consider the implications of object comparison on substitutivity. Since shallow equality is useful and easy to implement, we support shallow equality rather than no equality for OBSTACL objects.

4.1.4 Object Types

In a strongly typed language, an object should have a type, and we should also be able to manipulate objects through less specific types (supertypes). What should these types look like?

- Classes: Languages like C++ link types to classes. There is a subtype hierarchy that mirrors the class hierarchy.[2] Each object type has a name and is associated with behavior.
- Explicit Interface Hierarchy: Languages like Java allow the definition of a hierarchy of object interfaces. These interfaces are independent of classes so subtyping can exist without inheritance.[3] Each object type has a name and can be given a specification.
- Structural Subtyping: Many research systems use structural object types, which describe the interface to objects. Object types do not have a name, nor are they associated with a specification. Subtyping relations are inferred, not given explicitly. A structural type corresponding to the `array` interface type would be $\{|\mathrm{get} : \mathrm{int} \to \tau, \mathrm{set} : \mathrm{int} \times \tau \to \mathrm{unit}|\}$.

Using classes for object types is in conflict with our goal of substitutivity, because we cannot substitute an equivalent object instantiated from an unrelated class. In languages that treat classes as types (C++, Java), the use of types with no associated implementation (abstract classes in C++, interface types in Java) is evidence that tying classes and types together is too restrictive. Hence we only consider explicit type hierarchies (specified by the programmer) and implicit typing relations (inferred by the compiler). It is also possible to use structural types explicitly specified by the programmer, but it is somewhat cumbersome, so we shall not explore it.

The advantage of a programmer-specified subtyping hierarchy is that types are given names and meanings. A name such as `DataStream` is more meaningful than a structural type such as $\{|\mathrm{read} : \mathrm{int} \to \mathrm{string}, \mathrm{write} : \mathrm{string} \to \mathrm{unit}|\}$. Furthermore, the declaration of `DataStream` can include a description of data streams and a specification of how they work. Classes must explicitly state that they satisfy some interface.

[2] It is possible to define a C++ subclass that is not a subtype, so, strictly speaking, the subtype hierarchy is a *subset* of the class hierarchy.

[3] In Java, all subclasses are also subtypes, so the class hierarchy is a subset of the subtype hierarchy. This is the reverse of the relationship in C++.

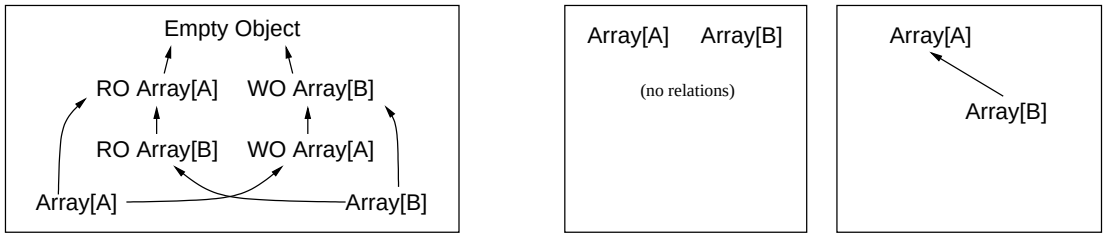
However there's no reason structural types cannot be given names. The more meaningful difference is how types are compared. Types are *name equivalent* if they refer to the same definition (where a name was given to it). Types are *structurally equivalent* if they have the same shape (record, tuple, object, function, etc.) and their components are structurally equivalent. Structural equivalence is more flexible but more error prone than name equivalence. As a result, many languages have both. In C and C++, record types are compared by name but most types are compared structurally [Str94, section 2.5]. In ML, abstract data types are compared by name but most types (including records) are compared structurally. The decision between an explicit subtyping hierarchy and structural object types is tied to type comparison—types from an explicit hierarchy are typically compared by name and explicit subtyping relationships, while structural types are necessarily compared structurally.

With structural object types there is a chance of "accidental subtypes", where two types are related through structural subtyping but weren't meant by the programmer to be related [BHJL86]. An example is `Point3D`$= \{|x : \text{int}, y : \text{int}, z : \text{int}|\}$ appearing to be a subtype of `Vector2D`$= \{|x : \text{int}, y : \text{int}|\}$, even though they should not be. Points and vectors have the same structural type but are not interchangeable. For instance, vectors can be added to vectors, but points should not be added to points. Another danger of structural types is that as interfaces evolve over time, subtyping relationships may be created or destroyed, through no fault of the programmer. For example, if two types are related in an earlier version of the program, then some parts of the program may rely on the subtyping relationship. If in a later version of the program, the two types are no longer related, the parts that relied on the relationship will break. Inferring structural types also reveals implementation information. For example, if a function f takes an object as its argument, then the inferred type of f includes the list of methods that f calls. Not only does this leak information, it is another form of brittleness—as the implementation of f changes over time, it may need to call another method, and the type (*interface* to f) must change.

Explicit subtyping helps verify the programmer's assumptions about the types of objects and the subtyping relations in the program. In addition, named object types support evolution of interfaces over time without accidental loss of subtyping relationships.

The main advantage of structural subtyping is its flexibility. There are many more subtyping relations than a programmer will specify, so explicitly listing subtyping relations will result in most relations being omitted. For example, suppose $\text{Array}[\tau] = \{|\text{get} : \text{int} \to \tau, \text{set} : \tau \times \text{int} \to \text{unit}|\}$ is a polymorphic array type, including objects which include a "get" method to read an element from the array and a "set" method to write to the array. When B is a subtype of A, $\text{Array}[B]$ is not a subtype of $\text{Array}[A]$ because putting an A object into $\text{Array}[A]$ objects is allowed, but we do not want the A object in a B array. Java allows this unsound subtyping relation for built-in arrays (but not for user-defined arrays) because it is quite useful to pass

arrays of B objects to functions that expect arrays of A objects, especially when those functions only *read* from the array. With structural subtyping, there is no subtyping between the array types but there *is* a relation on the read-only subset of them: $\{|\text{get} : \text{int} \to B|\}$ is a subtype of $\{|\text{get} : \text{int} \to A|\}$. Thus the function could accept read-only arrays of A, and a read-write array of B can be passed in as an argument. Figure 4.1 shows the rich set of subtyping relations resulting from just these two array types (RO indicates the read-only array; WO indicates a write-only array), and how they compare to the array class in C++ (`vector<T>`) and in Java (`T[]`).[4] Structural subtyping gives us the flexibility we want without tempting us to add unsound subtyping relations.



Implicit Subtyping (correct, flexible)

Explicit Subtyping, C++ (correct, inflexible)

Explicit Subtyping, Java (incorrect, flexible)

Figure 4.1: Structural subtyping produces more subtyping relations

Structural object types also replace "multiple interface inheritance" (the use of multiple inheritance to create additional subtyping relations). In OBSTACL, structural object types are useful with parameterized inheritance, which can produce objects with new types that have not been declared. ML uses structural types for most values. For consistency with ML and also for flexibility, especially with parameterized inheritance, we chose structural object types with structural subtyping for OBSTACL objects. In section 9.3.2 we show how to regain the advantages of an explicit type hierarchy—associating names and specifications with types to avoiding accidental subtyping.

4.1.5 Super Object Type

Given a hierarchy structure for both interfaces (subtyping) and implementations (inheritance), we must decide if there is to be a single root or multiple roots. Languages such as Smalltalk and Java have a class from which all others are derived. This class (often called Object) has two uses:

[4]Note that in C++, for each type `T` there is a supertype `const T` which is a read-only interface to `T` objects. The use of `const` can produce richer subtyping relationships than those produced only by inheritance. However, like C++ assignment and equality, `const` is *shallow*, only affecting contained objects and not objects referred to through pointers. Structural subtyping in OBSTACL can provide the equivalent of a `const` supertype, either shallow or deep, depending on the programmer's needs.

- Implementation. It allows for the definition of code that should exist in *all* objects in the system. Common code can include debugging, inspection, cloning, synchronization, serialization, comparison, and visualization. For example, Java defines methods such as `equals()` (comparison), `hashCode()` (hash tables), `toString()` (output), and `notify()` (synchronization) for all objects. OBSTACL does not have built-in cloning, serialization, synchronization, or visualization. Furthermore we do not assume that all objects can be placed into hash tables or formatted for textual output. More importantly, if we want to define operations such as output, hashing, debugging or comparison, we want them to extend to non-objects as well; implementing these operations in Object would limit their usefulness. Implementing common code in Object is often convenient but it tends to be domain specific and not flexible enough for programmers to adapt it to their own domains. In OBSTACL, there is no code that *all* objects must share. Each library can define its own domain-specific base class with common code.

- Interface. It allows for a single type representing the set of all objects. This type is commonly used for containers and other uses of genericity. ML already provides parametric polymorphism for genericity that extends to all types, not only object types. Structural object types provide a generic object type: $\{\!|\ |\!\}$ is a supertype of all other object types.

In OBSTACL, there is no need for an Object class at the root of a single class hierarchy. However, it does no harm, and having an empty class as the ancestor of all others is equivalent to having multiple class hierarchy roots. We will add an Object class to simplify the type theory (see chapter 7), but we will not add one to the language.

4.1.6 Access Rights

Protection of object data allows only some parts of the program to access private or protected fields. Where and how the line is drawn between privileged and unprivileged code varies from language to language.

We first consider *where* protection boundaries are set. Compile-time boundaries (modules, classes, functions) grant access rights to code based on where in the program it was defined. Run-time boundaries (processes, objects) grant access based on whether the code and data occur in the same unit of abstraction at run time. Often there is no practical difference between the two forms of boundaries, as many compile-time boundaries have but one instance at run time. In the case of objects, there is. Just as multiple invocations of a process cannot access the data of the others, with object-based protection, multiple instances of a class may not be able to access the data of the others. In contrast, with class-based protection, methods invoked on one object can access private data of different object instances from the same class. Such access breaks substitutivity. Consider the thought experiment in section 3.2.2. If a Point object were

```
class Base {
   private: int num_elements;
};

const int num_elements = 5; // some global variable

class Derived: public Base {
   void some_function() { ... num_elements+1 ... }
};
```

Figure 4.2: The protection mechanism affects name lookup

replaced by a MyPoint object, then all code that has access to Point internals will break. Also, to check access rights at compile time requires that the compiler can determine the class of an object. Our object types do not include the class name (see section 4.1.4) so this access scheme is not statically checkable in OBSTACL. Instead OBSTACL uses object-based protection.

The other dimension of access rights is *how* the line is drawn between privileged and unprivileged code. C++ and Java use accessibility-based protection, in which all information about the class is available but some of it is inaccessible. Access rights are granted by the class to other parts of the program, such as subclasses, object users, modules, or "friend" classes. Object creators can *see* the private and protected definitions, as well as the ancestors of a class, but they are not allowed to *access* these definitions.

In the example code shown in figure 4.2, the derived class references `num_elements`. Using visibility-based protection, `some_function` would use the global variable; using accessibility-based protection, it would see the private field, and thus cause a compile-time error. A disadvantage of visibility-based protection is that if the private field is changed to public or protected in a new version of the Base class, then the Derived class suddenly gets a different value for `num_elements`. The change is silent and therefore extremely likely to cause bugs that are difficult to track down. Therefore it may seem advantageous to use accessibility-based protection. However, the example shown in figure 4.3 shows an error that can occur when using accessibility-based protection instead of visibility-based protection. If the Base class is revised from the "Before" version to the "After" version, then the Derived class no longer compiles because it sees the private field from the Base class but cannot access it. It can be argued that this error is a compile-time error, and can therefore be detected, while the previous example leads to a logic error, which cannot be detected by the compiler; therefore the first situation is more important to avoid than the second. In the context of C++ and Java, accessibility-based protection is safer than visibility-based protection.

If we consider the change involved in each example, the former is a change to the *interface*, while the latter is a change to the hidden *implementation*. We should expect that changes to

Before:

```
class Base {
    private: vector<int> representation;
};

const int num_elements = 5;

class Derived: public Base {
   void some_function() { ... num_elements+1 ... }
};
```

After:

```
class Base {
    private: int representation[];
    private: int num_elements;
};
```

Figure 4.3: Private variable names affect program maintenance

the interface affect subclasses. We should *not* expect that changes to the hidden implementation affect subclasses. Visibility-based protection offers better hiding of implementation while introducing potential errors when changing interfaces.

OBSTACL uses visibility-based protection because names of private fields should never affect subclasses: the maintainer of the superclass cannot be expected to know what names all future subclasses rely on, and the subclasses should not be expected to know what names the superclass author used for local fields.[5] In the presence of run-time inheritance, mixins, and dynamic linking, private fields of the superclass may not be known at compile time, making static checks impossible. To avoid situations such as shown in the first example, OBSTACL does not provide direct access to names inherited from superclasses. Instead, they are accessed through "self", avoiding scoping confusion from new names being exported from the superclass. OBSTACL therefore supports visibility-based protection without the potential for error had visibility-based protection been used in C++ or Java.

4.2 Creating Objects

Now that we have chosen what sort of objects to use, we must decide how to create them. First we examine hierarchies and how to represent them. We explain why we chose to use classes to create objects and then look at ways to create classes.

[5] Dynamic scoping in early versions of Lisp and in current versions of Emacs-Lisp cause a similar problem: the local variable names used by the caller can affect the behavior of the function being called. The result is extremely error prone situations in which seemingly harmless changes to local variables in one function can affect the behavior of functions in other modules. Later versions of Lisp use static scoping, in which local variables can be renamed without fear of affecting the rest of the program.

4.2.1 Extensible Objects

Hierarchies are a natural way to organize information. Books (part, chapter, section, paragraph), outlines, businesses (CEO, vice president, manager, ordinary employee), and biological classification (kingdom, phylum, class, order, family, subfamily, genus, species) use hierarchies to organize information. However, with hierarchal definition of objects there are additional complications. For example, changes to definitions must be propagated to extensions, and the extensions should have some abstract view of entities they extend. Despite these complications, we chose to have hierarchal object definitions to increase modularity and promote code reuse.

Evidence from languages with extensible objects (SELF [US87], LambdaMOO [Cur97], Cecil) suggests that programmers distinguish between "generic" objects meant for extension and "proper" objects meant for ordinary use, including subtyping and method calls. Furthermore, extension and subtyping appear to be incompatible in a strongly typed language [Fis96] (see section 4.2.3). Languages with extensible objects are usually dynamically typed and therefore not concerned with detecting type errors in extension at compile time. One language that does combine static type checking with extensible objects is Cecil [Cha92]. In Cecil, one must distinguish between prototype ("template") and regular objects. Prototype objects are used for organizing a static inheritance hierarchy, while regular objects are created at run time and cannot be used for inheritance. Thus even in languages with extensible objects, the requirement of compile-time type checking leads to distinguishing between compile-time extensible objects and run-time subtypable objects. Since OBSTACL is statically typed, we chose to have two kinds of entities: those supporting extension and those supporting subtyping.

4.2.2 Prototypes vs. Classes

We now consider prototype objects and classes as potential extensible entities for OBSTACL. Class-based models are well known and widely used (C++, Smalltalk, Java). Recently, there has been a lot of interest in purely object-based models (also known as prototype-based models). The language most known for using a prototype-based model is SELF; the most widely used is JavaScript [Fla98]. The main advantage of these models is the conceptual simplicity of having only one kind of object, in comparison with class-based models, in which there are both objects and classes. How do prototypes and classes differ? Both are used to create proper objects. Both allow new definitions of fields and methods and redefinitions of methods. Prototypes however are like objects—their fields are given values and their methods can be called.

- Fields are given values. When a proper object is created, it inherits the field value from the prototype. For each object to have a different field value, fields must be mutable or there must be a mechanism to specify new field values when converting a prototype to a proper object. Forcing all fields to be mutable may be acceptable in an imperative

language like Java or C++ [6] but undesirable in a mostly functional language like ML. Furthermore, there may not be any field values that make sense for the prototype. For example, we may be building a set of objects with the invariant "each object corresponds to exactly one open network connection." The prototype however is an object that may be created before any network connections are opened. It therefore violates the invariant. Instead, we must add special case values and code to all the methods to handle the exceptional case (the prototype).

- Methods can be called. If methods defined in the prototype can be called before a proper object is created, methods must be written to accept either a prototype or a proper object as its host. This complicates the type system, requiring the maintenance of negative information. To see why, consider a prototype A and an extension B. A method defined in A must work in A and also in B. B's type however is not a subtype of A's because prototypes do not enjoy subtyping. Therefore the method must be polymorphic over all possible extensions of A. In addition to working in a family of prototypes, the method must work in proper objects. Even more complicated is when the method invoked on a prototype attempts to extend it. A more complicated form of object extension is needed to extend objects with imprecise type information.

Neither giving values to prototype fields nor calling methods in prototypes seems to be very useful, and both make the language more complex. In addition, defining and maintaining invariants can be more difficult when there is an additional prototype object that does not correspond to an entity being modeled. Prototype-based systems are also more difficult to statically type [Cha92]. We chose classes and class inheritance as a simple mechanism enabling extension without the full complexity of prototype objects and prototype extension.

4.2.3 Subtyping on Classes

As mentioned in section 4.2.1, subtyping and extension appear to be incompatible: if we do not have precise type information about an object (or class), it is difficult to extend it. In this section we show an illustrating example for which we want to inherit everything from a class for which we do not have precise type information.

Assume we allowed subtyping on classes.[7] In the example shown in figure 4.4, `EncryptedFile`'s signature would be a subtype of `File`'s signature. The `EncryptedFile` class should

[6] Note that even in C++, all fields are not mutable. Like ML, C++ distinguishes between mutable and immutable variables and treats initialization separately from assignment.

[7] The same issues arise if we allow extension (inheritance or delegation) of objects. We use an example with classes to avoid any problems with the combination of extension and object features (method calls, fields, etc.).

```
let FileClassType = classtype
   method read : int->string;
end;

class File
   method read(nbytes) = ...
end;

class EncryptedFile extends File
   method read(nbytes) = self.decode(key, next(nbytes));
   method decode(key, string) = ...
end;

fun UUEncode(F:FileClassType) =
   class UUFile extends F
      method read(nbytes) = self.decode(next(nbytes));
      method decode(string) = (* uu-decoding of the string *)
   end;
   UUFile (* return value *)

let uu = new(UUEncode(File))   (* create a new object *)
```

Figure 4.4: Class subtyping example

be usable in place of the `File` class. Let us consider what happens if we attempt to call `UUEncode(EncryptedFile)`. By assumption, `EncryptedFile`'s signature is a subtype of `FileClassType`, so the function call is type correct. Function `UUEncode` creates a new class with a `decode` method. Statically, this function is type correct: it adds a decode method to a class that does not have a `decode` method. At run time, we see that it adds a decode method to `EncryptedFile`, which has a decode method already. What should be done?

- Error: An error can be signalled. This means that `EncryptedFile` cannot be used in place of `File`, and our subtyping relation is unsound.
- Replace: The new method should replace the old method. Essentially, we inherit only what is visible in the class signature. This approach is motivated by a similar approach in ML's module system: when importing a module, a client can restrict the structure to some signature, and items that are not in the signature are not imported. If we do this for classes, the resulting object (`uu`) has a single `decode` method that takes a string and returns a string. What happens to the `read` method? `UUFile`'s `read` method calls `next_method()`, which is `EncryptedFile`'s `read`. `EncryptedFile.read` in turn calls `self.decode` with two arguments, resulting in a type error. Therefore this solution is

unsound as well. One could argue that the subtyping could be allowed only if `EncryptedFile`'s methods do not call `self.read` or call any function that calls `self.read`. However, this introduces an unnecessary and unexpected dependency: the users of the class do not know how the class is implemented, but the subtyping relationship depends on the implementation details. In addition, determining whether `self.read` will be called somewhere requires extensive program analysis and cannot be computed in the general case.

- Add: The new method should coexist with the old method. `EncryptedFile`'s methods will call `EncryptedFile`'s `decode` while `UUfile`'s method and object users will call `UUFile`'s `decode`. This scope-sensitive method lookup preserves type safety but leads to anomalies and an expensive implementation.

  With "scoped" inheritance, the object responds to a message by returning a method that can be seen. The definition of a new method introduces a new scope, while the redefinition of a method does not. The hiding of a method via subtyping ends the scope. The diagram in figure 4.5 records the scopes for our example. We can see that users of object `uu` see scopes $\mathcal{A}$ and $\mathcal{C}$. The definition of `EncryptedFile.read` and the definition of `UUFile` occur in scopes $\mathcal{A}$ and $\mathcal{B}$, so it should call `decode` from $\mathcal{B}$. Method lookup therefore depends not only on the object but what "view" the caller has of that object. This can lead to unexpected behavior when we use interface types.

  Consider the code in figure 4.6. Under the rules for scoped inheritance, `self.decode` should be `EncryptedFile.decode`. At compile time we do not know that `x` is an encrypted file object, so `x.decode` should be the most recent definition of `decode` in object `x`; in this example it would be `UUFile.decode`. Intuitively, `self.decode` and `w.decode` should be the same since `self` and `w` are the same. Furthermore, `w.decode` and `x.decode` should be the same since `w` and `x` have the same type and point to the same object. Thus we have a dilemma—intuition tells us all three should be the same but scoped inheritance tells us they are not. The problem is that in scoped inheritance, method lookup depends on whether we view the object as an encrypted file or as a uuencoded file. The structural type does not distinguish between these two. Scoped inheritance requires more precise object types, and is at odds with our goal of using interface types to maximize substitutivity. In addition, an implementation using flexible interface types requires two pointers per object, one for the object itself and one for the "view" [FKF98].

Inheritance works best when precise type information is known. Subtyping gives us imprecise type information. As we have seen, subtyping and extension are incompatible unless we preserve context: the object and name of a method are not sufficient to select a method; we must also know how the caller views the object. The "scope" information (which can be large)

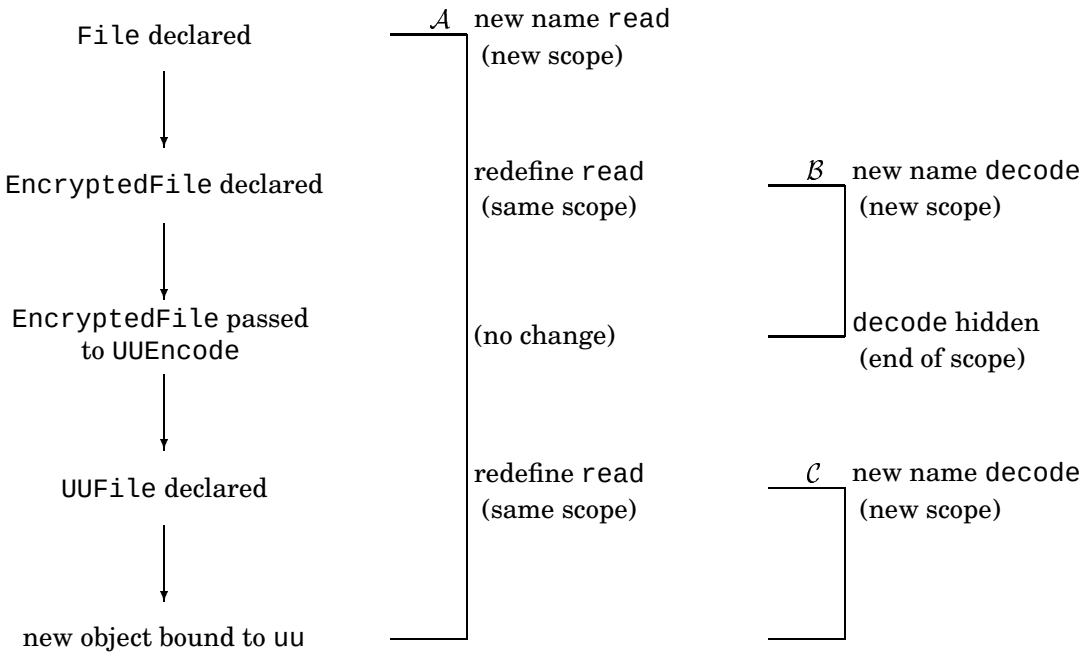


Figure 4.5: Scoped inheritance

```
class File
   method read(nbytes) = ...
   method test(x:{|decode:string->string|}) = raise Exn;
end;

class EncryptedFile extends File
   method read(nbytes) = self.decode(key, next(nbytes));
   method decode(key, string) = ...
   method test(x:{|decode:string->string|}) =
      let w:{|decode:string->string|} = self in
         self.decode;
         w.decode;
         x.decode
      end;
end;

let ef = new EncryptedFile();
ef.test(ef)
```

Figure 4.6: Example code using scoped inheritance rules

Original code:

```
class File
   method read(nbytes) = ...
   method name() = ...
end

class EncryptedFile inherits read from File (* partial inheritance *)
   method read(nbytes) = self.decode(key, next(nbytes))
   method decode(key, string) = ...
end
```

After blank filenames are treated specially:

```
class File
   method read(nbytes) = if self.name()="" then stdout.read()
                                           else File.read(handle)
   method name() = ...
end;
```

Figure 4.7: Example of partial inheritance

must be passed along with the object at run time or it must be encoded in to the object type, so that it can be used to perform method lookup. Although extension and subtyping might be combined in this way, scoped inheritance can lead to an expensive implementation if we use structural object types, and also results in unexpected behavior of programs, so we do not support inheritance with partial type information.

4.2.4 Partial Inheritance

We also consider partial inheritance with full type information. Like method replacement in section 4.2.3, partial inheritance is either unsound or reveals implementation information. Consider a variant of the `EncryptedFile` example shown in figure 4.7. `EncryptedFile` inherits the `read` method but not the `name` method from `File`. When `File.read` is invoked, it receives a reference to `self`, but `self` should not have `name` in it. Suppose `File` is then modified to treat blank filenames specially. Now inheriting `read` without `name` does not work: `read` needs `name` to be in `self`. `EncryptedFile` no longer compiles! `EncryptedFile` now depends on the *implementation details* of `File`. We do not want subclasses to depend on the implementation of a superclass; they should only depend on the set of protected and public definitions, the constructor, and the instantiator (see section 3.5). We therefore do not support partial inheritance in OBSTACL. However, in section 9.1.5 we consider the possibility of inheriting a definition and then hiding it; such a feature would serve approximately the same purpose as partial inheritance.

4.2.5 Run Time Inheritance

Classes are extended through inheritance. Should inheritance be a compile-time operation, as in C++ and Java, or a run-time operation, as in SELF and Smalltalk? Run-time inheritance would allow for more flexibility, but what are its costs in performance, language complexity, and ease of implementation?

- Performance: Compile-time inheritance may allow some optimization such as inlining of method calls and knowledge of object layout. In OBSTACL, parameterized inheritance and user-defined instantiators already keep us from applying these optimizations, so for run-time inheritance we may not pay any additional costs in terms of lost optimizations.
- Language Complexity: Run time inheritance implies classes can be created and passed around at run time. We therefore have to give class values a class type (or signature). Classes are extensible but not subtypable, so we do not need to define the subtyping relation on class types. A form of class types is already needed for parameterized inheritance (see class constraints in section 5.3.2), so supporting class types in the user-visible language may not be a significant addition.
- Implementation: Since classes can be created at run time, we have to store in memory any class information needed for inheritance and instantiation. It must include the size and layout of objects, a list of methods, a list of constructors and instantiators, and values of free variables.[8] Even with compile-time inheritance, the size and layout of objects is needed for parameterized inheritance; the list of methods is needed given structural object types; and values of free variables are needed if we allow nested class definitions as in Java.

Given the language features we would like to support in OBSTACL, it does not appear that run-time inheritance would incur much additional cost. We therefore support run-time inheritance. However, since classes do not support subtyping (see section 4.2.3), run-time inheritance is less useful than in languages such as SELF and Smalltalk, which do not have static type systems. We will use parameterized inheritance (section 5.3) to relax the constraints for run-time inheritance.

4.2.6 Multiple Inheritance

Inheritance allows a subclass to reuse code from a superclass. Multiple inheritance allows a subclass to reuse code from multiple superclasses. Multiple inheritance is more flexible than single inheritance, and seems essential to solving certain problems. We consider two forms of multiple inheritance:

[8] If a class definition occurred within the scope of a local variable definition, the method bodies may access the local variable, so that variable's value must be stored along with the class.

- Unrelated superclasses: The superclasses share no ancestors, and the resulting subclass supports more than one interface. There are two cases of unrelated superclasses we consider:
  - Interfaces: At most one superclass provides a non-trivial implementation, and the other superclasses are used to provide alternate interfaces. For example, an Editor could support a Stream interface so that the contents of the edit buffer can be accessed as a stream of characters. This idea is formalized as the Adapter pattern [GHJV95], and is explored in section 6.3.1.
  - Implementations: More than one superclass provides an implementation, and the subclass provides any necessary communication between components. For example, an edit dialog window may inherit from either XWindow or CursesWindow (system-specific implementations of an abstract Window class), and also from Editor (implementation of basic editing functionality); we will explore this example in 6.4.3. A more detailed example of this nature can be found in [Cop92]. Another example is provided by Smalltalk, where multiple inheritance would better express the structure of the collection hierarchy [Coo92].

  In languages that link subtyping to inheritance (such as C++), multiple inheritance is the only way to produce an object that supports more than one interface. Java allows a class to implement multiple interfaces but does not support inheritance of multiple implementations [GJS96].

  A problem that can occur when combining unrelated classes is that a name may be used in more than one superclass. These ambiguities must be resolved, either explicitly by the programmer (as in C++ and Eiffel [Mey87]) or automatically by the language implementation (as in CLOS [Kee89], Dylan [Com92], and Python [Lut96]). In C++, the programmer must write a new method that explicitly calls one of the superclass methods. In Eiffel, the programmer must rename one or more superclass methods when they are inherited. Automatic ambiguity resolution is error prone [BCH+96], as it is sensitive to slight rearrangements in the class hierarchy, and introduces a dependency on the entire ancestor graph instead of only one the immediate superclass [Sny86].

- Related superclasses: The superclasses share some ancestor, and the resulting subclass combines implementations that support the same interface. For example, if PepperoniPizza and SausagePizza each inherit from Pizza, we can create a PepperoniAndSausagePizza (see the class graph in figure 4.8 and code in figure 4.9). To support this "diamond" shaped inheritance structure, the language must support sharing of the Pizza ancestor. In C++, the shared class is called a *virtual base class*.

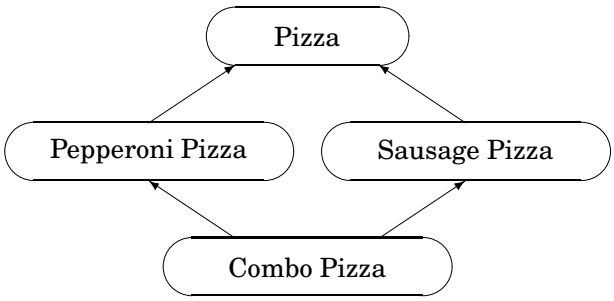


Figure 4.8: A diamond-shaped hierarchy

In addition to ambiguity resolution, diamond shaped inheritance structures introduce a subtle problem: methods that are correct in the superclass may not behave properly in the superclasses. Consider the Pizza example, in which we use the abbreviations Z, P, S, and C to refer to `Pizza`, `PepperoniPizza`, `SausagePizza`, and `ComboPizza`.

The code of $C.\mathrm{prepare}()$ is perfectly reasonable from the C's implementor's viewpoint: it simply calls the methods inherited from C's parents and adds some behavior of its own. However, the result is that $Z.\mathrm{prepare}()$ is called twice which, most likely, was not intended by the programmer. We do not want our pizza to have crust, sauce, cheese, pepperoni, another crust, more sauce, more cheese, and sausage. There is no way for $C.\mathrm{prepare}()$ to know that it should call a *sibling* instead of a *parent* method. There is a similar problem with object construction. The constructor of C should call a constructor of P and one of S. However, both P and S call a constructor of Z. The invariants of P or S may be violated if Z is not constructed properly.

C++ does not provide a good solution to either problem. In C++ the author of C must also call the constructor for Z with values that satisfy both the requirements for P and those for S. As a result, the internal invariants of P and S must be exposed to all subclasses. For ordinary methods, the language provides no such support. The approach suggested by the designer of the language [Str97, section 15.2.4.1] requires the implementor of $C.\mathrm{prepare}()$ to call special "helper" methods of all ancestor classes explicitly without relying on parent calls. Although this works initially, as the program is changed, it can lead to errors. Let us consider that $\mathrm{prepare}$ in each method calls one or more p methods, each of which does *not* call the parent version, but instead requires $\mathrm{prepare}$ to call p in each parent.

- Suppose S did not have p originally. The original version of $C.\mathrm{prepare}$ would call $Z.p$, then $P.p$. Now if in a later version of S, there is a p method, $C.\mathrm{prepare}$ is wrong.
- Suppose Z is factored into two classes, Z (Pizza) and H (CheesePizza). Both Z and H

```
class Pizza (* Z *)
      method prepare() = ...place crust, sauce, and cheese ... ;
end;

class PepperoniPizza (* P *)
      extends Pizza
      method prepare() = Pizza.prepare(); ...place pepperoni ... ;
end;

class SausagePizza (* S *)
      extends Pizza
      method prepare() = Pizza.prepare(); ...place sausage ... ;
end;

class ComboPizza (* C *)
      extends PepperoniPizza, SausagePizza
      method prepare() = PepperoniPizza.prepare(); SausagePizza.prepare(); ... ;
end;
```

Figure 4.9: Pizza class definitions

have a p method. However, $C.\mathrm{prepare}$ does not call $H.p$, and we are left with a pizza without cheese.

The problem here is that subclass C must change even though the *interface to the superclass did not change*. Only the implementation details of the superclasses changed. In situations like these, the subclass is dependent on the implementation of superclasses instead of only their interface. Even worse, the error is *silent*—there is no easy way for the programmer to know that there may be a problem.

To solve this problem, we need the language implementation to provide some sort of mechanism to call the ancestor methods in some order. CLOS provides "before" and "after" methods for some access patterns but not others. For example, we may want to call the superclass method in the *middle* of the subclass method, or we may want to call the superclass method conditionally. The stream example in section 6.4 has a `read()` method that fits neither the "before" nor "after" pattern. For these patterns, CLOS provides the `call-next-method` function to traverse the ancestor graph, or the `define-method-combination` macro to define new patterns of combining methods. However, with this approach, *local* changes to the class hierarchy can affect the *global* linearization order used to traverse the ancestor graph. Even seemingly harmless changes to the class hierarchy such as refactoring or inheriting superclasses at different levels in the hierarchy can lead to silent errors [BCH+96]. As a result, superclass changes that appear to be correct and invariant preserving can invalidate subclass code.

C++ supports multiple inheritance both for unrelated classes and for related classes (using virtual base classes [Str97]). C++ and Eiffel also support multiply inheriting the same class *without* sharing, and the Eiffel data structure libraries use this technique often [Mey94]. Inheritance typically expresses the *is-a* relationship instead of the *has-a* relationship. Multiple inheritance of unrelated classes usually indicates inheritance is being used for a *has-a* relationship. Both inheritance of unrelated superclasses and of related but unshared classes are often misused for convenience, and they can be expressed in languages without multiple inheritance by using composition and forwarding [VRTB98]. However, related shared classes cannot be combined in this way—multiple implementation inheritance is needed to express diamond-shaped hierarchies.

Multiple inheritance is useful for writing certain kinds of programs. However it is a complex and controversial language feature [Car93] that introduces new problems. In OBSTACL we chose not to support multiple inheritance, because we believe a combination of language features can be used instead. In section 6.4 we show how to combine single inheritance, structural object types, object composition, and parameterized inheritance to replace most uses of multiple inheritance.

4.3 Initializing Objects

Having chosen classes to produce objects, we now look at ways to initialize objects. From our experience with non-objects (such as records, lists, and unions), we might expect initialization to be easy and not worth thinking about. For these types, the initial values provided by the user are exactly the ones that are stored in the larger structure. Given an abstract data type or a very simple object system such as the "record of functions" approach, initialization is still fairly simple: the user provides initial values, which are used to compute values to go into the larger structure. However, objects are built in a distributed way: each ancestor class defines fields that must be initialized, and there is no single program component that knows how to initialize all the fields properly. One approach would be for initial values to be provided when the class is defined, but we would then have to update any fields that are different for each object. (Most fields fall into this category.) In addition there may not be any initial values that make sense before an object is initialized (see section 4.1.2). We instead want to provide initial values when a class is *instantiated*. Each class should provide values for the fields it defines so that subclasses do not have to depend on the private fields of the superclass (see section 3.5); each class therefore provides a *constructor* to initialize fields. Sections 4.3.1 and 4.3.2 cover issues with constructors. In section 4.3.3 we look at how to package various stages of object construction into constructors and instantiators.

4.3.1 Multiple Constructors

Should a class be able to provide more than one constructor? Many objects can start their existence in more than one state. For example, a Unix network socket object may start listening on a port, connect to a port on another machine, or remain unconnected. It is more elegant to start the object in the correct state than to start it in an incorrect state and later change it. Furthermore, it does not greatly complicate the language to support multiple constructors, so we support them in OBSTACL.

C++ and Java support multiple constructors through overloading. A class may have constructors with different types for their parameters, and a constructor is chosen by examining the types of the arguments passed at instantiation time. For example, `new Socket(3333)`, `new Socket("stanford.edu", 80)`, and `new Socket()` could create a socket listening on port 3333, a socket connected to port 80 on `stanford.edu`, and an unconnected socket, respectively. However, overloading does not handle all cases. For example, if we want to build both cartesian and polar points, both constructors would take two real numbers, so `new Point(90.0,45.0)` does not tell us or the compiler whether a cartesian or polar point is desired. An alternative, used in Smalltalk, is to give each constructor a name. We can then call `Point.cart` or `Point.polar` to create the right kind of point. ML does not support function overloading, and it uses named constructors for union types and abstract data types. In OBSTACL we provide named constructors, both because they are consistent with ML and because they are more general and easier to understand than overloaded constructors.

4.3.2 Initialization Phases

The main role of a constructor is to provide initial values for the fields of the object being built. Since the author of the subclass does not know what private fields are defined by the superclass, the subclass constructor should also call a superclass constructor. Figure 4.10 shows a C++ constructor, in which local fields are initialized and then a parent constructor is called.

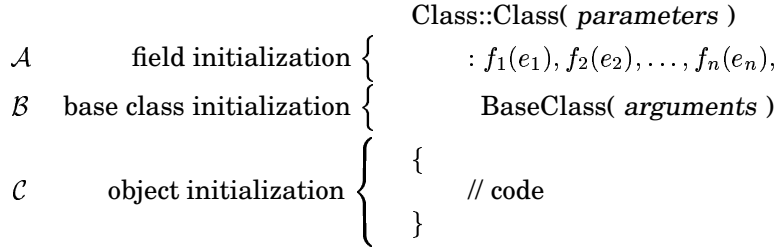


Figure 4.10: Parts of a C++ constructor

One disadvantage of this form is that it is not possible to perform some computation that is used to initialize two or more fields. The workaround is to leave some fields uninitialized in

part $\mathcal{A}$ and then assign values to them in part $\mathcal{C}$. We would like to avoid this problem in OBSTACL without resorting to using uninitialized mutable fields. All fields should be initialized in part $\mathcal{A}$ before part $\mathcal{C}$ is executed.

The other role of a constructor is to establish invariants associated with the object's existence. For example, a window object may bring up and draw the contents of a window, a database object may establish a connection with a database, or a background task object may start a new thread of execution. In part $\mathcal{C}$, the constructor can invoke arbitrary code, including methods of the newly created object.

For type safety, the two phases of construction (field initialization and object setup) should be separate. During field initialization, it is unsafe to call the object's methods, since they may access uninitialized fields. During object setup, the constructor must be allowed to call methods that set up the object's invariants. C++ allows fields to be initialized in either part $\mathcal{A}$ or $\mathcal{C}$ so it is possible for methods to be called before all fields are initialized. For example, in the program fragment shown in figure 4.11, the constructor of `X` correctly initializes fields before calling method `m`, but does not take into account the redefinition of `m` in a subclass. The result of constructing a `Y` would be access to an uninitialized field (`lore`). To avoid this problem, C++ specifies that the constructor call the superclass method, not the subclass method (*i.e.*, the redefinition is ignored). Not only is it expensive to implement this behavior,[9] it violates the meaning of method redefinition: that the new method is always called instead of the old one. In addition this behavior is a frequent source of confusion for a new C++ programmers, and also eliminates a useful pattern: the constructor might request that the subclass perform some action during execution of the superclass constructor. We therefore seek a solution in which the correct methods are called during object setup.

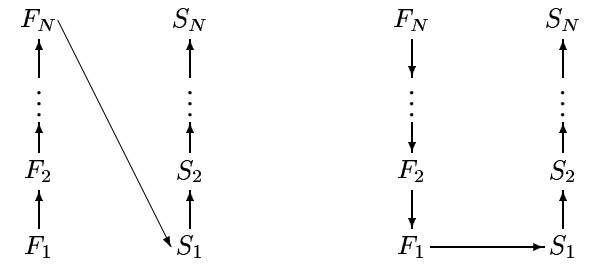
In addition to the phases that occur *during* object construction, we may want to perform actions before and after object construction. The beginning of construction is already too late to make decisions required to implement creational design patterns in [GHJV95]. These sorts of decisions can instead be made in the *instantiator*. Languages that force the use of an instantiator give control of creation policy to the author of a class, while languages that offer public access to the constructor give more control to the user. There are advantages to both, but in OBSTACL we generally favor giving control to the class, and allow the class to export abilities such as field update or object construction to the user if desired. Additionally, having instantiators in the language allows us to simplify the treatment of protected methods (see chapters 7 and 8) and eliminate the object creator's dependency on the class's creation policy (see section 3.6.1). We therefore chose to have both instantiators and constructors in OBSTACL.

[9] C++ compilers typically implement this by switching method dictionaries at run time; see chapter 8 for a description of how method lookup is implemented. This switch needs to be thread safe in a higher level language, and thus would introduce locking for all method calls, even though all method calls after the construction phase do not need it.

```
class X {
   int data;
   X(int i) {
      data = i;
      m();
   }
   virtual void m() { cout << data; }
};

class Y: public X {
   int lore;
   Y(int j): X(j) {
      lore = j;
   }
   virtual void m() { cout << lore; }
};
```

Figure 4.11: Constructor calling redefined method



(a) Two Pass Ordering (b) One Pass Ordering

Figure 4.12: Ordering of initialization phases

4.3.3 Order of Construction

In OBSTACL, the user calls an instantiator, which *may* call a constructor. Alternatively, it can return an existing object (Singleton and Object Cache patterns), call a different instantiator (Virtual Constructor and Private Class patterns), or clone an existing object (Prototype pattern). A constructor has three duties: initialize fields, call a superclass constructor, and execute any additional code required to set up the object. To fully construct an object from a class with $N-1$ ancestors requires calling N constructors. Suppose for class i we label field initialization F_i and object setup S_i. In what order should these steps be performed? The most important constraint is that all fields should be initialized before any methods are called, so all the F steps should be performed before any S step. Second, if public and protected fields of a superclass ($F_1 \ldots F_{i-1}$) are needed to compute the initial value of a subclass field (F_i) then superclass fields should be initialized before subclass fields. Third, superclass invariants should

be set up before subclass invariants, so that the subclass can rely at least on the subclass portion of the object being set up properly. Thus the ideal order in which these steps are performed is for fields to be initialized first (superclass before subclass) and then all object setup code executed (superclass before subclass), as shown in figure 4.12(a). However, a two pass system is harder to understand than a one pass system because we cannot view the subclass constructor as "calling" the superclass constructor. The two pass system is also less efficient both because of the overhead of a second set of calls and because the temporary values generated during the first set of constructor calls must be stored or recomputed in order to be used in the second part of the constructor. Since all fields in OBSTACL are private, we are not constrained to initialize superclass fields before subclass fields. We therefore chose the simpler, more efficient *one pass* ordering shown in figure 4.12(b).

4.4 Summary

OBSTACL's design supports modular programming while maintaining consistency with ML. Objects and classes are simpler in OBSTACL than in many object-oriented languages because they reuse rather than replace non-object language construct. Objects are similar to records of methods, and support message sends, imperative field updates, and shallow (identity-based) equality tests. Methods are ML functions with access to the host object (*self*) and private fields. Updates to fields are supported with ML references. Object types are similar to ML record types, but support subtyping. Classes are used primarily to create objects. Classes do not support subtyping, and can be extended with single inheritance. Multiple inheritance, partial inheritance, scoped inheritance, and method renaming are not supported, but classes can be parameterized over their superclasses, and inheritance is allowed at run time. Objects are created by calling a class's instantiator, which usually responds by creating an object and calling a constructor. The constructor initializes fields, calls a superclass constructor, and then sets up object invariants. OBSTACL objects and classes remain simple because they work with rather than replace existing ML features such as functions, abstract data types, and modules.

Chapter 5

Language Design

> The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.
>
> *–George Bernard Shaw*

In chapter 3 we looked at how modular programs can be built by reducing dependencies between program components. In chapter 4 we looked at language features and how they relate to our modularity goals and how consistent they are with the design of ML. In this chapter we turn to the design of OBSTACL. In general, we do not consider language features that are primarily *local* in nature. If a programmer must perform extra work that only involves local changes (*e.g.*, within a class or module), then a language feature to reduce that work is primarily for convenience. We consider such features useful but not for the core language; they will be considered in chapter 9. Instead we consider features that would otherwise require the programmer to perform non-local work (*e.g.*, throughout the program).

5.1 Objects

An OBSTACL object combines data (fields) and code (methods). Operations are message sends, field updates (only from within the object), and comparisons. We treat objects as being similar to records produced by functions, as shown in figure 5.1. The record returned by this function is not a real OBSTACL object, but rather an ML record that looks somewhat like an object. The `let` at the beginning declares identifiers that are similar to private fields in an object; the record (marked by $\{\ldots\}$) contains fields that are similar to public methods in an object. The `open` function is an "instantiator", a function that creates objects. The behavior of objects that contain only private fields and public methods is similar to that of ML records of

```
fun open(filename) =
  let fd = ref NONE,
      buffer = ref "Erase before reading"
  in
    {
      open = fn() => (fd := SOME(IO.open_in(filename))),
      read = fn() => (case !fd of
                        NONE => raise IO.io
                      | SOME(f) => (buffer := IO.input(f,1); !buffer)),
      close = fn() => (case !fd of
                        NONE => raise IO.io
                      | SOME(f) => (IO.close_in(f); fd := NONE))
    }
  end
```

Figure 5.1: A function producing a pseudo-object

public functions with the private fields defined in the enclosing scope.

In OBSTACL, one may not define a stand-alone object like the one above. Instead, a class must be defined, describing the state and behavior of a family of objects. When the class is instantiated (see section 5.2.5 for details), a new object is created. The distinction between classes and objects is similar to that between an ADT definition, a description of an abstract type in terms of operations that are applicable to it, and ADT values, specific instances of the abstract type. Unlike ADT values, OBSTACL objects do not keep any ties to the class from which they were instantiated.[1]

5.1.1 Fields

As in the pseudo-object above, an OBSTACL object may have private fields that are accessible in the method bodies but not from outside the object. Support for public and protected fields is examined in section 9.1.2. Each object has its own fields, but they are immutable by default. The fields above are mutable because they are bound to ML references, which provide imperative assignment with the `:=` operator.

5.1.2 Methods

The stream pseudo-object shown above exports methods in the record. OBSTACL supports both public and protected methods in objects. Support for private methods is explored in section 9.1.3. The methods can be thought of as being defined in the "scope" of the fields, so they have access to private fields. OBSTACL does not support binary methods or multimethods. In the OBSTACL object model, the methods are part of the *object*, not part of the class.

[1] In chapter 8 we shall see that an efficient implementation may maintain a connection between the object and the class. However this connection is not seen by the user of the object.

5.1.3 Object Types

Since objects are similar to records of methods, we expect object types to be similar to record types. OBSTACL object types are *structural*: they list the names and types of methods in the object. To distinguish object types from record types, which are written $\{x_i : \tau_i\}$ in ML, we write object types $\{|x_i : \tau_i|\}$, where x_i are names of methods and τ_i are the function types associated with those methods. As with ML record types, the order in which names are listed in object types is insignificant. Thus, $\{|\text{read} : \text{int} \to \text{string}, \text{write} : \text{string} \to \text{unit}|\}$ is equivalent to $\{|\text{write} : \text{string} \to \text{unit}, \text{read} : \text{int} \to \text{string}|\}$. An object type serves as an abstract interface that describes the messages the object is willing to receive.

Object types support subtyping: one type σ_1 can be considered a subtype of another type σ_2 (written $\sigma_1 <: \sigma_2$) if objects of type σ_1 can be used in place of objects of type σ_2 without causing a type error. There are two kinds of subtyping supported by objects:

- *Width subtyping* allows σ_1 to contain at least all the components of σ_2. More precisely, if $\sigma_2 = \{|x_i : \tau_i|\}$ for $1 \le i \le N$, then the subtype σ_1 can be $\{|x_i : \tau_i|\}$ for $1 \le i \le M$ (where $M \ge N$). For example, $\{|m : \alpha, n : \beta|\} <: \{|m : \alpha|\}$, but $\{|m : \alpha|\} \not<: \{|n : \beta|\}$. Width subtyping allows *all* object types to be a subtype of $\{|\ |\}$.

- *Depth subtyping* allows the components of σ_1 to be subtypes of the corresponding components of σ_2. More precisely, if $\sigma_2 = \{|x_i : \tau_i|\}$ then σ_1 can be $\{|x_i : \tau_i'|\}$ where for each i, $\tau_i' <: \tau_i$. For example, if $\alpha <: \beta$ but $\alpha \not<: \gamma$, then $\{|m : \alpha|\} <: \{|m : \beta|\}$ but $\{|m : \alpha|\} \not<: \{|m : \gamma|\}$.

The two forms of subtyping can be combined. Components that σ_1 and σ_2 share are compared with depth subtyping and components that are not shared are compared with width subtyping. Combining width and depth subtyping is a consequence of the transitivity of subtyping. In addition to transitivity, the subtyping operator is antisymmetric and reflexive, making it a partial ordering on types.

With subtyping the static type of an identifier may not match the actual type of the object. The type system ensures that the object supports *at least* the methods listed in the static type. Each client can have its own view of the object.[2] In figure 5.2, the only restriction on the interface types τ_1 and τ_2 is that $\tau <: \tau_1$ and $\tau <: \tau_2$, where τ is the true type of the object.

Object types, unlike record types, tend to be used in ways that lead to mutually recursive types. We therefore allow object types to be declared as recursive, illustrated in figure 5.3. Subtyping on recursive types is potentially uncheckable. However for common cases we can use a simple algorithm to test if $\alpha <: \beta$. First, assume that $\alpha <: \beta$. Then, check if $\alpha <: \beta$ by the usual subtyping rules, except if the usual rules require checking of $\alpha <: \beta$ at some deeper level,

[2] Factoid: With subtyping on structural object types, there are an infinite number of subtypes of any given object type, as expected. However, with subtyping on function types, there are also an infinite number of *supertypes* of any given object type.

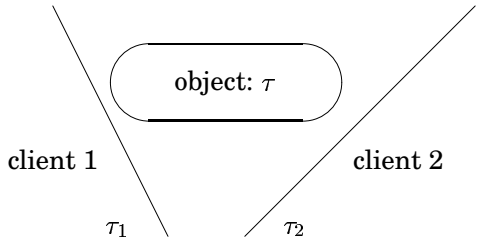


Figure 5.2: Multiple views of an object

```
objtype Automobile = {|
           get_tires : unit -> Tire list
           |}
and Tire = {|
           get_automobile : unit -> Automobile option
           |};
```

Figure 5.3: Recursive type definitions

we assume $\alpha <: \beta$ already. If $\alpha \not<: \beta$, we will discover it in some way other than by assuming $\alpha \not<: \beta$. Although this algorithm is sound, it is not complete. Subtyping on recursive types is not addressed in our work on OBSTACL; it is an independent problem that has been studied in other work [HM95, Pot98].

Depth subtyping extends naturally to non-object types such as tuples and record. Functions are an odd case: $\alpha \to \beta <: \gamma \to \delta$ iff $\beta <: \delta$ and $\alpha >: \gamma$. Note that the requirement on the argument to a function is opposite of what one might expect [Fis96, AC96]. Refs too are an odd case. A τ ref can be thought of as a pair of functions, $\text{get} : \text{unit} \to \tau$ and $\text{set} : \tau \to \text{unit}$. For $\tau \text{ ref} <: \sigma \text{ ref}$, we need $\text{unit} \to \tau <: \text{unit} \to \sigma$, which simplifies to $\tau <: \sigma$. But we also need $\tau \to \text{unit} <: \sigma \to \text{unit}$, which requires $\tau >: \sigma$. If $\tau <: \sigma$ and $\tau >: \sigma$, then $\tau = \sigma$. Another view of refs is as single element arrays; see section 4.1.4 for an example showing the subtyping relationships for arrays.

5.1.4 Substitutivity

Since object types do not include the class from which the object was instantiated, objects from unrelated classes can share an object type. As a result, we have substitutivity: we can replace one object by another object, which may be implemented differently but presents the same interface. In chapter 8 we will see the underlying machinery that makes substitutivity possible.

5.1.5 Operations

OBSTACL objects support two operations: message sends and comparison for equality. A message consisting of only a name (which must be known statically for type checking reasons[3]) is sent to an object; it responds by selecting and returning a method with that name. For example, given object `x` of type $\{|m : \text{int} \to \text{string}|\}$, `x.m` sends the "m" message to `x`, which responds by returning a function of type $\text{int} \to \text{string}$. To then call the method with an argument 5, one would write `x.m(5)`. (It could also be saved and used just like any other function value in ML.) Note that in OBSTACL the arguments are *not* part of the message.

The other operation supported by objects is comparison. Two objects `x` and `y` can be compared with `x=y`. The result is `true` only if `x` and `y` are the same object. Each use of the `new` operation on a class results in an object that is unequal to all existing objects. If `x` and `y` refer to different objects but have the same values for fields, they compare unequal. (See section 4.1.3 for a discussion of why OBSTACL objects use this form of equality.) Another operation we would expect on objects is field update. OBSTACL objects do not support update directly; instead, fields can be bound to ML references, which support imperative update. OBSTACL objects remain simple, supporting only message sends and comparisons.

5.2 Classes

Objects in OBSTACL are not defined in isolation. Like abstract data type definitions, a single definition can be used to create an entire class of objects. This class definition describes both the representation and behavior of objects. A class may be defined to be an extension of another class, containing all the definitions of the other, plus additional definitions and redefinitions of inherited methods.[4] OBSTACL classes do not provide modularity or encapsulation; ML modules provide these at compile time and the objects produced from classes provide these at run time. OBSTACL classes do not serve as object types, although an object type can be derived from a class definition. OBSTACL classes do not form a layer of protection (access control). Compared to classes in other languages, the classes in OBSTACL play a lesser role.

The primary purpose of a class in OBSTACL is to create objects. The components of a class are:

- Fields describe data that goes into each object.
- Methods describe behavior that goes into each object.
- Constructors initialize objects as they are being built.
- Instantiators respond to requests for new objects.

[3] Dependent types [Aug98] may relax this restriction, but static type checking still requires that we know which message is being sent, since the type of the result depends on the message name.

[4] However, unlike classes in some languages, classes in OBSTACL may not *omit* or *rename* definitions.

Most object languages have constructors but not instantiators. In these languages, a class user may create an object directly, and then the class is given a chance to initialize it. By the time the constructor is called it is too late to influence the process, other than to abort it altogether. An instantiator is a form of encapsulation of the object construction step. Just as hiding data behind accessor methods increases flexibility for the class author, hiding object creation behind instantiators increases flexibility of construction. OBSTACL requires that an instantiator for class C return an object of the proper type (including all public items of C) but does not ensure that the returned value is actually a new instance of C. It may return an existing instance of C, corresponding to the Singleton and Object Cache patterns (see section 3.6.1), or an instance of a different class, corresponding to the Virtual Class and Remote Proxy patterns. However the usual case is to create a new instance of C. If a new object is created, the system invokes a constructor to initialize it.

A constructor initializes an object in two phases. The first phase initializes fields of the new object. Since a class may be an extension of another, the new object may contain fields from multiple class definitions. A constructor from each class definition is invoked to initialize the fields from that class. The second phase is used to set up invariants for the object. Again, each class definition involved can contribute to the second phase. Fields are initialized in the first phase of initialization; methods can only be invoked in the second phase.

Classes in OBSTACL are run-time values. Like other values, classes have a type. Class types include the components of the class used to create the object (constructors and instantiators) and the components used to define the contents of the object (fields and methods). Sections 5.2.1 and 5.2.2 describe fields and methods; section 5.2.3 deals with class extension; sections 5.2.4 and 5.2.5 describe constructors and instantiators; and section 5.2.6 describes class types.

5.2.1 Fields

Each class can define private fields, which are initialized in a constructor and accessible to only the methods defined by this class. In figure 5.4, `handle` is an immutable field and `count` is a "mutable" field. Field values are defined per object, so the constructors bind initial values for `handle` and `count`; note that `count` is initialized to a ref value. All fields in OBSTACL are immutable; we build "mutable" fields by using ML refs. As a result the language is simpler—there are no new operators for field updates. The methods in each class can access the fields defined in that class simply by naming them. The `read` method in `InputFile` uses the `handle` field, and the `read` method in `CountingFile` uses the `count` field. Since `count` is bound to a ref, access to it uses the ML ref operations: `!` for dereferencing and `:=` for assignment.

5.2.2 Methods

Each class can define new methods and redefine methods inherited from the superclass. New methods are defined with the `method` keyword; redefinitions use the `redefine` keyword. The two are distinguished in OBSTACL for type safety and to avoid coding errors (see section 6.1.2). Although methods are *defined* in a class, they can be *invoked* only in objects. Given an object `x` and a method `m`, `x.m()` invokes the method. Within `m`, the identifier `self` is bound to the object, `x`.[5] In figure 5.4, the `read` method in `CountedFile` uses `self` to call the `increment` method. The type of `self` is an object type with all public and protected methods; this type is longer than the type exported to the user, which includes only public methods. In `InputFile`, the type of `self` is $\{|\text{read} : \text{int} \to \text{string}|\}$. In `CountedFile`, the type of `self` is $\{|\text{read} : \text{int} \to \text{string}, \text{increment} : \text{int} \to \text{unit}|\}$. Private fields are not accessed through `self`, nor are they listed in the type of `self`. Instead, all private fields are bound to identifiers within method bodies. For example, the `read` method in `InputFile` can access the field `handle` simply by naming it. If the method is a redefinition, its signature must be a subtype of the signature of the old definition, and the new definition can access the old definition by referring to the identifier `next`. The `read` method of `CountedFile` invokes the old method by calling `next(nbytes)`. In addition to the distinction between new and redefined methods, each method can be public (available to object users) or protected (available only to subclasses). Protected methods such as `CountedFile.increment` are typically helper functions, and are marked with the `protected` keyword.

5.2.3 Inheritance

OBSTACL classes can be extended using single or parameterized inheritance. Parameterized inheritance is performed with linear mixins, which will be described in section 5.3. A class declared as extending another class will inherit the definitions in the superclass. The new class (the subclass) may add new methods, redefine inherited methods, and add new private fields. Inherited methods cannot be renamed or removed. If a method is redefined, the redefinition must have a subtype of the type of the inherited method. Since all fields are private, the subclass definitions cannot see the fields defined in the superclass or its ancestors. However, those fields are present in objects instantiated from the subclass, and may be accessed by methods inherited from the ancestors. To initialize these fields, the subclass constructor must call a superclass constructor. Like fields, all methods from the superclass are present in instances of the subclass. Therefore, at some level, inheritance implies subtyping: the actual types of the subclass objects will be subtypes of the actual types of the superclass objects. However, the user is presented with a smaller type than the actual type (in particular, protected methods are hidden), so the user may not see a subtyping relationship where there is inheritance.

[5] There is nothing special about the identifier `self`, other than that it is automatically bound to the host object in the method body. In particular, it is not a new keyword and therefore is not treated specially by the parser.

```
class InputFile
   field handle: FileDescriptor;
   method read(nbytes) = IO.input(handle, nbytes);
   constructor make(filename)
      fields {handle = IO.open_in(filename)};
   instantiator make(filename) = new InputFile make(filename);
   instantiator makeTemp() = new InputFile make(IO.gettempname());
end;

class CountedFile extends InputFile
   field count: int ref;
   redefine read(nbytes) = (self.increment(nbytes); next(nbytes));
   protected method increment(nbytes) = count := !count + nbytes;
   constructor make(filename)
      fields {count = ref 0};
   instantiator make(filename) = new CountedFile make(filename);
end;
```

Figure 5.4: Example classes in OBSTACL

5.2.4 Constructors

An OBSTACL constructor is shown in figure 5.5. Each constructor has a name, which is used to distinguish it from other constructors for the same class.[6] The body consists of three parts: field initialization, a call to the parent constructor (if the class has a parent), and optional initialization code.

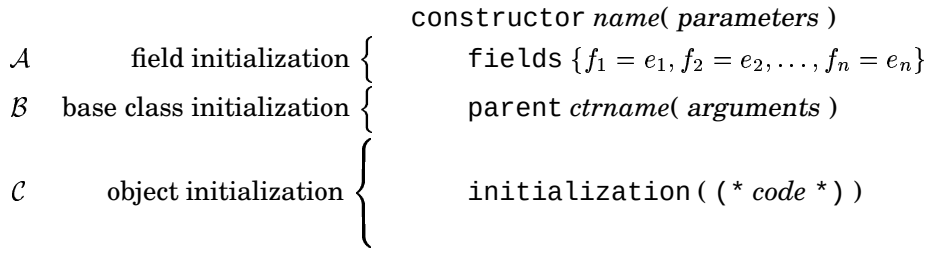


Figure 5.5: Parts of an OBSTACL constructor

Constructors are responsible for initializing the fields and setting up invariants for each object. Constructors in OBSTACL are given names, in contrast to C++ constructors. A constructor definition consists of the following three parts:

fields This clause must contain an expression that evaluates to a record containing a value for each field defined in the class. More formally, if fields $f_1, \ldots, f_n$ are defined in class C,

[6] C++ uses overloading for this purpose, but it can be confusing and it makes it difficult to have multiple constructors with the same types of arguments.

```
constructor fraction(num:int,div:int)
  fields
    let (q,r) =
      IntInf.divMod(num,div)
    in
        { units=q,
          num=r,
          div=div }
    end
```

(a)

```
fun init_calc(i:int)
      : {index:int, prime:int} =
          {index = i,
           prime = (* i^th prime *)}

constructor fifth()
  fields init_calc(5)

constructor thirteenth()
  fields init_calc(13)

constructor i^th(i:int)
  fields init_calc(i)
```

(b)

Figure 5.6: Flexibility in field initialization

then every constructor defined in C should include

`fields` *expr*

where *expr* has the type $\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$. Note that this is an ordinary record and not an object.

Values for fields $f_1, f_2, \ldots, f_n$ are given in a field record $\{f_1 = e_1, f_2 = e_2, \ldots, f_n = e_n\}$. By using this form instead of field assignment syntax (*e.g.*, `f1:=e1`), it is clear to the programmer that we are not providing access to an uninitialized object—from the programmer's perspective, the object does not yet exist. We use ML record syntax instead of some special field syntax for simplicity and flexibility. For example, without side effects a single computation can be used to compute values for more than one field.[7] For example, in figure 5.6 *(a)* both the quotient and remainder are computed in a single call, and the values are stored in two fields. Another benefit is that multiple constructors can share code that computes initial values.[8] Figure 5.6 *(b)* shows an example of sharing initial-value code. If one goes further and views an object as being a set of these field records, it remains consistent with our modularity goals, such as allowing different classes to provide fields with the same name and having only private fields (see chapter 6). OBSTACL's approach to field initialization uses existing ML constructs to provide flexibility without complicated language extensions.

[7] A criticism of C++ is that this is not possible. A C++ programmer has to resort to side effects or repeated computation to initialize multiple fields with the results of a single computation.

[8] Again, the C++ programmer cannot do this directly, and must resort to side effects or code duplication.

parent If C inherits from another class P, the constructor must call one of P's constructors to initialize the inherited fields. This clause gives the parent constructor name and the arguments to be passed to it. The arguments are evaluated in a scope that includes the object's fields but not a reference to "self". We can therefore use the initial field values but not call methods in this object. The restriction is necessary because we cannot safely call methods until all of the object's fields are initialized.

initialization This optional clause contains the code that is to be executed after the object has been fully constructed but before it is returned to the instantiator. The `init` clause is commonly used to set up invariants. For example, a network file server object may be registered in a global filesystem registry; a tree node may notify its parents and children of its existence; a window object may establish a connection to the windowing system; and a file object may set up a finalizer[9] to flush and close the file. The scoping rules are the same as for methods: the self keyword and the private field identifiers are bound in an enclosing "scope", and therefore are available to the initialization expression. The code is treated as the body of a special method, so there are no restrictions on what it may do. Unlike C++, any methods called by the initialization code are looked up using the object's true class, not the class defining the constructor.

Notice that constructors do *not* create objects—they only provide the initial values for the object's fields and define the initialization code.

Any class which is intended to be instantiated must have at least one constructor and can possibly have more than one (each with a different name). Some of the constructors may provide default values for certain fields or accept arguments in different formats and convert them to the internal representation that can be stored in the object's fields. For example, a `FileStream` class may have a constructor to create a closed stream and another to create an open stream.

5.2.5 Instantiators

Constructors are a way for a class to set up an object being created. Instantiators are invoked by a class user to request a new object. How the class responds to this request is described by the class and is not controlled by the user.

An instantiator is an ordinary function, except that it resides in the class definition and has access to object creation. Instantiators are the only functions in a class; constructors describe object initialization but are not functions, and methods are described in a class but actually

[9] C++ has *destructors*, which are run before the object is destroyed. In OBSTACL, objects are destroyed by the garbage collector. A *finalizer* can be used to perform clean-up operations just before the object is destroyed.

reside in instances of that class. Instantiators can evaluate ***instantiation expressions***, introduced by the `new` keyword:[10]

`new` *C ctr-name args*

Such an expression can only occur within an instantiator and is used to create and initialize an object of the surrounding class.

The following operations are performed by the run-time system before a value is returned:

1. The constructor named *ctr-name* is invoked in class *C*. If *C* is derived from another class, the constructor will then invoke one of its parent's constructors, and that will invoke one of its parent's constructors, and so on. As a result, a constructor will be invoked for *C* and each of its ancestor classes so that all the fields are initialized.

2. An object is put together by combining the records containing initial values of the fields (provided by the constructors) with a class pointer, which points to the run-time data structures of class *C*.

At no point can the class writer access an object that is not fully formed, so type safety is ensured. It *is* possible, however, for the class writer to access the object before it is fully initialized by the initialization clause in constructors.

An instantiator does not have to call `new`. If the design of the system requires that only one copy of the object is created (the Singleton pattern), the instantiator can instantiate a new object the first time it's called, and return a reference to the existing instance in response to all subsequent calls. An instantiator *I* may also call another instantiator, possibly creating objects of a different class, as long as the type of the object eventually returned to the user is a subtype of the object type associated with the class in which *I* is defined. This approach is useful for abstract classes whose instantiators return objects of concrete subclasses. In general, the goal of instantiators is to encapsulate the process of object creation.

One may need multiple instantiators, so that clients have a choice of which instantiator to call when creating a new object. Some of the instantiators may provide default values for the object's fields, others may accept values in a format different from the object's internal representation (for example, a point object may have an instantiator that accepts polar coordinates from the user and converts them into rectangular coordinates when constructing an object). Different instantiators may be used to create an object in different modes (a file descriptor object may be created in a "read" or "write" mode). In general, multiple instantiators provide flexibility for the clients of the class.

Notice an important difference between the motivation for multiple constructors and the motivation for multiple instantiators. Every class has two kinds of clients: derived classes and

[10] It is not necessary for `new` to be a keyword. In an alternate syntax, `new` could be a predefined identifier bound to a record containing functions corresponding to constructors. In such a syntax, the user would write `new.ctr` instead of `new C ctr`.

```
  instantiator Point.polar(r, t) =
    if r=0 then
      new Point origin ()
    else
      new Point rect (r * cos t, r * sin t);
```

Figure 5.7: One instantiator, multiple constructors

```
type InputFileType = classtype
   field handle: FileDescriptor;
   method read : int -> string;
   constructor make : string;
   instantiator make : string -> InputFile object;
   instantiator makeTemp : unit -> InputFile object;
end;
```

Figure 5.8: A class type

users of the objects instantiated from the class. Multiple constructors are needed to enable flexible object construction by the derived classes. Multiple instantiators are needed to enable flexible object creation by the class users. For example, a file object may be created in "open" or "closed" mode. Alternatively, a single instantiator may invoke one of several constructors. In the code fragment shown in figure 5.7, the arguments to the instantiator are used to determine which constructor to call. If r is zero, a point at the origin is created. If it is non-zero, the polar coordinated is converted to rectangular, and the constructor for rectangular coordinate points is called. A more sophisticated instantiator might call other instantiators. For example, if there is another Point class that is optimized for polar coordinates, this instantiator could "forward" the object creation request to the other class's instantiator. Section 6.2 explores more uses of instantiators; section 9.4.4 considers the possibility of not having instantiators in the language.

5.2.6 Class types

Each class has two types, one for public methods and instantiators and another also including protected methods and constructors. A class type looks similar to a class definition, except that in place of constructor, instantiator, and method definitions are constructor, instantiator, and method types. As an example, the class `InputFile` from section 5.2.2 can be described by the type shown in figure 5.8.

Structurally similar object types are related by subtyping. Width subtyping allows some definitions to be omitted to create a supertype. Classes in OBSTACL do not support width subtyping, for reasons explained in section 4.2.3. Depth subtyping allows subtyping on the types of components. Although depth subtyping on class types seem reasonable, it is not compatible

```
class B
   method m() = self.n().catMethod();
   method n() = Cat.new();
end;

type A = classtype
   method m : unit -> unit;
   method n : unit -> Animal;
end;

let a' : A = B;
class C extends a
   redefine n() = Animal.new();
end;
```

Figure 5.9: Class subtyping example

with class extension, as the example in figure 5.9 shows. If depth subtyping is allowed, class `B` can have type `A`. Extensions of classes with type `A` are allowed to redefine `n` as long as it returns an `Animal`. But class `C` now contains an inherited method `m`, which requires that `n` return a `Cat`. Resolving this conflict requires some way to allow `A.n` and `C.n` to coexist. Such a solution introduces much complexity (see section 4.2.4). OBSTACL class types do not support subtyping in general. However, contexts in which class types are used (extension, mixin application) have rules in which some limited subtyping is allowed.

Class types are similar to object types. Since objects are created from classes, we expect the types to be related. With each class type we can associate two object types, $pub(C)$ and $prot(C)$. All public object-level items are included in $pub(C)$. In the core version of OBSTACL, this includes only public methods. However, given extensions to support public fields (see section 9.1.2), public fields would be included as well. Public *and* protected fields and methods are included in $prot(C)$. Note that $pub(C)$ is the maximal type that an instantiator may return; one can use the object with smaller types. The type $prot(C)$ is the "self type", the type of the object as seen by itself, and is a subtype of $pub(C)$. Private fields are not included in any object type.

Given a definition of class C, the type checker computes the corresponding maximal public and protected object types $pub(C)$ and $prot(C)$. The public type $pub(C)$ is accessible in a program with the syntax `C object`, and is used as the default return type of instantiators of C. For example, the above class uses `InputType object` to refer to the inferred public type of objects instantiated from `InputType`. The protected type $prot(C)$ is used as the type of the `self` identifier in methods.

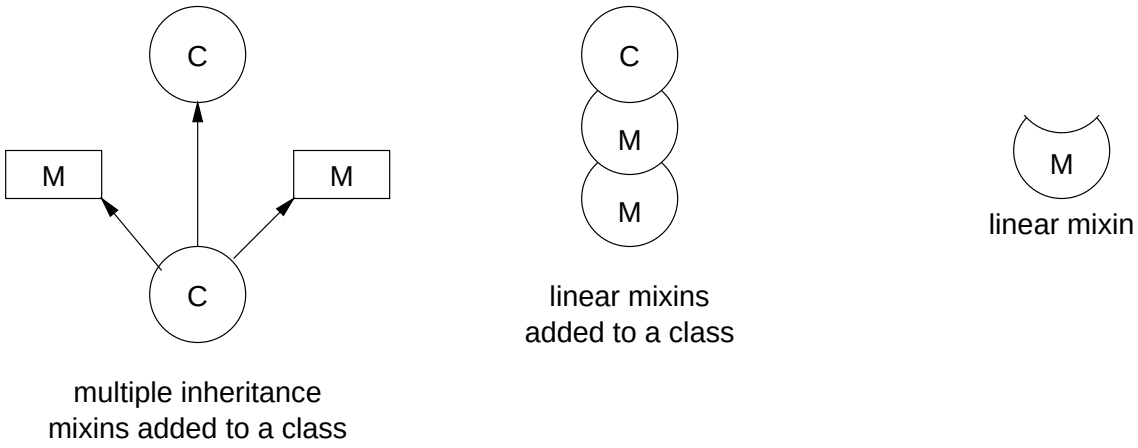


Figure 5.10: Linear mixins vs. multiple inheritance mixins

5.3 Mixins

A mixin is a set of definitions added to another class. It differs from definitions introduced by subclassing in that a mixin is not attached to a specific superclass; it can be added to any number of classes. Mixins are usually "mixed in" to a class with multiple inheritance or similar facility. OBSTACL supports a variant of mixins we call *linear* mixins (but we will drop the "linear" modifier when it is clear we are referring to OBSTACL's mixins), which are essentially subclasses parameterized over a set of superclasses. The advantage of linear mixins over multiple-inheritance-based mixins include multiple application, control over ordering, and a way to statically describe and check constraints on their combination. The advantage of linear mixins over conventional inheritance is that they leave both the superclass and subclass open for modification, instead of only the subclass. The decomposition of ordinary inheritance into linear mixins plus mixin application is similar to the decomposition of let binding in lambda calculus into functions plus function application. A linear mixin can also be viewed as a function that takes a class and derives a new subclass from it.[11] The same mixin can be applied to many classes, obtaining a family of subclasses with the same set of methods added and/or replaced. By providing an abstraction mechanism for inheritance, linear mixins remove the dependency of the subclass on the superclass, enabling modular development of class hierarchies—*e.g.*, a subclass can be implemented before its superclass has been implemented. Linear mixin inheritance can be used to model single inheritance and many common forms of multiple inheritance [BC90, BLS94].

The general idea of a linear mixin can be made clear by an example (in C++ syntax) shown

[11] In ML, functions take values and produce values; functors take modules and produce modules; OBSTACL mixins take classes and produce classes. The three are not unified because they have different forms of type relationships between input and output types.

```
template <class I> // I must be IceCreamCode or a subclass
class chocolate: public I {
  public: void eat() {
    if (amount_left <= 0)   I::eat();
    else                    --amount_left;
  }
  private: int amount_left;
};
```

Figure 5.11: Linear mixins simulated with C++ templates

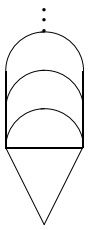


Figure 5.12: There is no limit to the number of times a mixin can be added.

in figure 5.11. If we are given any class that represents an ice cream cone, we can produce a new class that has an extra scoop of chocolate ice cream on top (see figure 5.12).

Unlike multiple inheritance mixins in C++ and Python, where the `chocolate` mixin can be added only once, we can add the same linear mixin as often as we want, as long as the class interface matches. We can do this because the subclass `chocolate` has a name for what it's extending (`I`), so methods like `eat()` have a clear path up the inheritance chain. Having an arbitrary number of additions of a mixin makes them much more useful than mixins that can be added only once. With our example, how good would our ice cream class be if we could not add two scoops of chocolate?

5.3.1 Definition

A mixin definition lists the definitions that will be mixed into a class. It follows the same syntax as class extension except that the supertype is a parameter with a constraint. The constraint, which we will describe in section 5.3.2, is essentially a class type. The example in figure 5.13 shows a constraint definition and two mixin definitions. `Encrypt` is a mixin that takes any class whose type conforms to `WritableStreamType` (*i.e.*, any class that has `read` and `write` methods of an appropriate type) and derives a new class from it that implements encryption on top of the basic stream functionality. Similarly, `UUEncode` creates a uuencoded stream from any stream class that implements `read` and `write`.

```
let classtype WritableStreamType
    method read: unit->string;
    method write: string->unit;
end;

mixin Encrypt extends S with WritableStreamType
   ... // Private cryptographic methods: encrypt and decrypt
   redefine read() = decrypt(next());
   redefine write(data) = next(encrypt(data));
end;

mixin UUEncode extends S with WritableStreamType
   ... // Private encoding algorithm: encode and decode
   redefine read() = decode(next());
   redefine write(data) = next(encode(data));
end;
```

Figure 5.13: Mixin example

5.3.2 Constraints

Looking at the type checking of ordinary inheritance (described in detail in section 7.4), it is clear that the rules depend on checking the existence, non-existence, and types of methods and constructors in both the subclass and superclass. Mixins do not have the superclass available. Instead, we abstract the superclass out of the type checking rules for ordinary inheritance, resulting in rules for mixin application, plus a set of *constraints* that restrict the set of superclasses to exactly those that would pass the type checking rules for ordinary inheritance.

The constraint should be given by the programmer as a pair of clauses. The positive clause (following the `with` keyword) should be a set of typed method and constructor signatures. It specifies which methods and constructors a class *must* have in order for the mixin application to be type correct. The negative clause (following the `without` keyword) should be a list of untyped method names. It specifies which methods a class *must not* have in order for the mixin application to be type correct.

Positive clause. The positive clause is a list of typed method and constructor signatures.

`classtype` `method` $m_1 : \tau_1$; ... `method` $m_k : \tau_k$; `constructor` $c_1 : \sigma_1$; ... `constructor` $c_\ell : \sigma_\ell$; `end`

To ensure that all method calls and field references can be resolved correctly, the positive clause must include all of the parameter's methods and constructors that are used in the mixin's body. The three kinds of uses are method calls, method redefinitions, and constructor invocations. A subclass does not use private fields or instantiators defined in the superclass,

so these are not needed in the positive clause. For each method call `self.m` in the subclass, if m is not defined in the subclass, then it must be defined in the parameter, and thus appear in the positive constraint.

For each call site i, let τ_i be the type of m. Then the positive clause must include `method` $m : \hat{\tau}$ where for all uses i, $\hat{\tau} <: \tau_i$. The type $\hat{\tau}$ should be the most general type that satisfies the requirements, but may be more specific to further constrain mixin application.

The positive clause must also include the names and types of all methods that are redefined in the mixin using the `redefine` keyword. Redefining a method implies that the superclass has a method with the same name and a compatible type. This is ensured by including the signature of the redefined method in the positive clause of the constraint. In addition, the type may be further constrained by uses of `next` within the method.

The subclass also uses constructors of the superclass. Thus constructors must be included in the constraint. Recall from section 5.2.6 that the type associated with a constructor is the type of the argument only, and from section 5.1.3 that subtyping on arguments is contrary to what one might expect. For that reason the type $\hat{\tau}$ is a supertype of each τ_i instead of a subtype, as in the case of methods.

An example of a mixin parameter constraint can be found in the streams example above. The parameter of `UUEncode` mixin is constrained by `WritableStreamType`. Any class that has `read` and `write` methods can be used to subclass a `UUEncoded` stream from it.

The programmer who implements the mixin may require that the parameter have certain methods even though they are not called in the mixin's body. The appropriate signatures can be added to the constraint list by hand. Additional constraints may be useful for stating future needs. For example, if the programmer knows the mixin will require a method in the future, adding the method to the constraint prevents the client from using the mixin on a class without the method, and then later complaining that a library upgrade broke the program.

Negative clause. The negative clause lists all methods that the parameter may *not* have. It should include the names of all new methods defined in the mixin (those defined with the `method` keyword as opposed to `redefine`). This is necessary to avoid type incompatibility problems arising when the superclass and the subclass contain unrelated methods with the same name. The type checker signals an error if a method is included both in the positive and the negative clauses.

Although using a negative clause instead of providing for method renaming reduces flexibility of mixins, the cases it prohibits are those that lead to type safety violations or to coding errors (see section 6.1.2).

5.3.3 Application

The type checker rejects any attempt to create a new class by applying a mixin to a class whose class type does not conform to the mixin's constraint. When mixin M is applied to class C, for each method signature $m_i : \tau_i$ in P's positive clause C's class type must contain a method m_i with type τ_i', where $\tau_i' <: \tau_i$. If the method is public in the constraint it must also be public in C. The reverse is not true—a method may be protected in the constraint but public in C. For each constructor $c_i : \tau_i$ listed in the positive clause, C's class type must include a constructor c_i with type τ_i' where $\tau_i <: \tau_i'$.

Also, C's class type must not include any methods with names listed in M's negative clause. It is important that C's class type lists all of C's protected and public methods without hiding any of them. If a method m is hidden in C's class type, then a subclass derived from C may define a new method m with an incompatible type (see section 4.2.3). This will lead to run-time type errors when m is called from other methods defined in C, but the type checker won't be able to detect these errors at compile time since it does not have access to the type of the original m when compiling C's subclasses.

The key rule of subtyping is that if a context requires a value of type α and the value used in that context has type β, then β must be a subtype of α. For mixins, the contexts appear in the mixin body. As described in the previous section, the type of methods in the positive constraint is a subtype of all the types required in the mixin body. Class C must contain a method with a type that is a subtype of the types listed in the positive constraint. Since subtyping is transitive, the type in the class (which is the value) is a subtype of the types required in the mixin body (which is the context). An additional context is C's definition, which can call methods redefined in M. For this context the type of methods in M must be a subtype of the type of the method in C.

The rules for mixin constraints and the rules for mixin application jointly ensure type safety for mixins.

5.4 Summary

OBSTACL objects are fairly simple, providing only selection and comparison. Object types are also simple, consisting of a set of name/value pairs. Object types may be related through subtyping, which comes in two flavors, width and depth. Objects are produced by instantiating classes, which contain all the definitions necessary to initialize and populate an object with fields and methods. Instantiators encapsulate the construction process; constructors define modular initialization. Classes can be built by extending another class or by applying a mixin to a class. Mixins are an abstraction for inheritance and allow the same extension to be applied to a family of classes. Objects, classes, and mixins form the core of OBSTACL.

Chapter 6

Evaluation

In this chapter we will look at how programmers can express designs in OBSTACL. In sections 6.1 and 6.2 we look at how objects can be used and created. In sections 6.3 and 6.4 we look at how classes can be used and created, including how systems using multiple inheritance can instead be expressed with parameterized inheritance, composition, and interface types. Along the way we will look at design decisions in OBSTACL that encourage good programming practices and make more difficult certain kinds of errors. In section 6.5 we summarize guidelines for using OBSTACL's features to help write maintainable, well-designed programs.

6.1 Using and defining objects

6.1.1 Equality

Which form of equality is best for objects? Section 4.1 lists four types of equality. Shallow equality considers only object identity, and seems inappropriate for many situations. Why does OBSTACL support only this form of equality for objects? Equality in mathematics is a reflexive, symmetric, transitive relation. We would like to preserve these properties when defining equality in our language. In the presence of subtyping, data hiding, and substitutivity, some of these properties no longer hold when using forms of comparison other than shallow equality.

An equality relation that automatically takes into account the structure and values inside an object may seem to be ideal. Deep equality is the form of equality used for immutable ML values, and it is intuitively what one would expect from an equality relation.

However in the presence of data hiding, deep equality faces some oddities. Are only public items of two objects compared, or also private fields? If only public items are compared, then the compiler would have to compare methods (functions), which cannot be compared in a finite amount of time.[1] Thus we must compare only the private fields.

[1] To say $f = g$, we must show that $f(x) = g(x)$ for all values of x. Since there may be an infinite set of values of x,

How to compare private fields of objects of different classes is not clear in the presence of substitutivity. Suppose there are two classes `Point2d` and `DebugPoint2d extends Point-2d`, where `DebugPoint2d` has no additional definitions. We require that if an instance of `Point2d` is replaced with an instance of `DebugPoint2d`, where all the methods and fields remain the same, there be no change in the program's behavior. In other words, the implementation detail (which class was used to instantiate the object) should not dictate behavior; only the fields and methods can change behavior.

The consequence of the above requirement is that object comparison must allow comparisons between objects instantiated from related classes. Suppose for example that `Point3d` is a subtype of `Point2d`, and that there are objects `a = Point3d(1,2,31)`, `b = Point2d(1,2)`, and `c = Point3d(1,2,42)`. By the requirement for substitutivity, a=b and b=c. However a≠c. Thus transitivity is violated.

We may wish to define an equality relation at each type. In such a system, $a =_{\mathrm{Point2d}} c$ but $a \neq_{\mathrm{Point3d}} c$, and transitivity is restored, at least for a single equality relation.

A more serious problem with deep equality (whether a single relation or one for each type) is that examining private fields invades the privacy of a class. Suppose instead of being a subclass, `DebugPoint2d` is a copy (*i.e.*, has the same definition). The classes `DebugPoint2d` and `Point2d` are no longer related by subclassing. Can instances of one be equal to instances of another? If instances of one can be equal to `Point2d` objects and instances of the other cannot, then these two classes *with exactly the same definition* do not produce objects that are comparable. Thus we require that instances of exactly the same class definition behave the same.

The consequence of the above requirement is that object comparison must allow comparisons between objects instantiated from unrelated classes. The `x` and `y` fields of `Point2d` and the corresponding fields from `DebugPoint2d` are compared when an instance of `Point2d` and an instance of `DebugPoint2d` are compared. However, how can the compiler determine what is an equivalent field? If it looks at fields with the same name, then the *names of private fields* have an effect on the behavior of the object! Thus the principle of data hiding is violated.

Deep equality, which must involve inspecting private fields of multiple objects of different classes, runs into trouble with subtyping, substitutivity, and data hiding. One approach taken by many languages (including C++ and Java) is to push the problem of equality to the user. Each class can define an equality relation. Java offers an `.equals()` method on objects that breaks symmetry. C++ offers an `operator ==` function, which is dispatched at compile time and does not take into account the true dynamic type of an object. To handle both symmetry and dynamic dispatch requires some form of binary methods or multiple dispatch. However, when the two objects are instantiated from unrelated classes, neither class's equality relation

function equality cannot be determined in a finite amount of time.

is suitable, even in the presence of binary methods. Allowing binary methods leads to violating our requirement of identical class definitions producing identical results. Suppose for example that `DebugPoint2d` defined an equality test with other `DebugPoint2d` objects, and that `Point2d` defined an equality test with other `Point2d` objects. When a `Point2d` instance and a `DebugPoint2d` instance are compared, neither class's user-defined method will be invoked, and either an error will be signaled or the objects will compare `false`.

From the above examples it is clear that shallow equality must be used to preserve substitutivity and transitivity. Yet, for two `Point2d` objects at the same coordinates not to compare equal seems rather undesirable. What seems like a serious problem is not much of an issue in OBSTACL because not everything must be represented as an object. Points for example, as well as other algebraic types (see section 3.3) are best represented as ADTs. For "physical" entities like automobiles, bank accounts, and files, shallow equality is generally preferable. For example, two bank accounts may have equal balances but that does not make the accounts equal (the same). Two files may have the same contents without being the same file. In general, "values" are equal when their contents are equal. In OBSTACL these are represented with ML types like lists, tuples, and records, which use deep equality. The equality of "objects" depends not on the contents, but on identity. In OBSTACL, these are OBSTACL objects, which use shallow equality. Forcing all abstractions to be values or to be objects leads to the anomalies described above. OBSTACL keeps these separate and avoids the associated problems.[2]

6.1.2 Redefinitions

In OBSTACL, new method definitions are distinguished from redefinitions. Early version of C++ had the distinction but later versions dropped it. This distinction seems unnecessary, and sometimes reduces the potential for reuse. The code in figure 6.1 shows a mixin `M` that could be applied to both classes `A` and `B` if only the distinction between new and redefined methods (and correspondingly, the negative constraint on the mixin parameter) were dropped. In this case, maximizing reuse is at odds with other traits of the language.

In a statically typed language, the type of a redefinition should conform to the type of the method being redefined. In contrast, a new method is unconstrained in its type. If `M.m` has a type compatible with (*i.e.*, a subtype of) the type of `A.m`, then it can be considered a redefinition. But what if it does not? The types of `M.m` and `A.m` are known at compile time (at the time of inheritance–mixin application), so the compiler could reject the code with a compile-time error. Or it could accept the code but treat `M.m` as a new method, hiding the old definition of `A.m`. (Note that `A.m` must still be part of the object, so that `A.q` works properly.) Hiding complicates

[2]Copying and equality are interlinked. Generally, when a copy is made, the copy should be equal to the original. Objects using shallow equality (like bank accounts) tend to be those for which casual copying does not make sense. Objects using deep equality (like matrices, sets, points, and strings) tend to be those for which casual copying is reasonable.

```
mixin M
  protected method m() = ...;
end;

class A
  protected method m() = ...;
  public method q() = self.m();
end;

class B
  public method q() = ...;
end;
```

Figure 6.1: Redefinitions vs. new methods

the language but is a possible solution; see section 4.2.4. Either solution is reasonable when the program is considered a static system. However, it is when the program is changed that there can be trouble.

Suppose `M.m` and `A.m` have the same type at first but a later version of `A` changes the type of `m` slightly, so that `M.m` is no longer a redefinition. Either the compiler rejects `M.m`, which means the program no longer compiles, or it accepts `M.m` and hides `A.m`, in which case the behavior of `A.q` has silently changed. The change is to the interface of `A.m` so any code that refers to `A.m` must be checked and possibly modified. We should expect that the program may break when an interface changes. However we should also expect that some programmer should be responsible for fixing it.[3] The author of `A` does not know about `M`, but can announce to all users of `A.m` that things may need to change. The author of `M` does not know about `A`, and has no reason to change anything. The programmer who applied `M` to `A` is the only remaining party, and sees nothing in `M`'s constraint suggesting that `M` depends on `A.m`. Each programmer did the right thing, yet the program broke.

Although in this example the compiler could warn that the type is incompatible, the same situation can arise when the semantics of `A.m` changed without changing the type. In this scenario, the compiler cannot detect a problem, and the program will break at run time (possibly returning erroneous results instead of signalling an error). The program is that the author of `M` has not stated whether `M.m` should redefine `A.m`. Any guess by the compiler will be wrong some of the time.

The reason for distinguishing new methods from redefinitions boils down to expressing intent. Associated with a new method is a meaning—a description of current behavior ("this method draws a shape in the current window"), expected behavior ("redefinitions may draw

[3] We saw this issue come up with *multimethods*. Code could be extended in a way where no one was responsible to fix things that went wrong.

any shape in the current window in the default color and pen width"), and restrictions[4] ("redefinitions should not close the window, open new windows, change the font, reboot the machine, ..."). [5] A redefinition on the other hand must conform to an existing type and meaning.

Defining a new method and redefining an existing method are fundamentally different. Making them use the same syntax prevents the language system from catching errors. In the presence of mixins the distinction is more valuable, as it helps avoid improper application of mixins.

6.2 Object Creation Patterns

In this section we explore how constructors and instantiators can be used to hide the details of object construction from users of an object. Instantiators can be used to hide additional initialization and bookkeeping code from users, but they also hide exactly which class is instantiated. With this flexibility we can substitute an equivalent object with a more efficient implementation, return an existing object instead of creating a new one, or return a set of objects that work together to act as one. Instantiators are "first-class" functions, so we can also export them from the class to build collections of instantiators in a larger data structure.

Many creational design patterns allow a class to control the object creation policy [GHJV95]. In OBSTACL, these patterns are implemented using instantiators. Since instantiators are always called by object creators, it is easier to add a creational design pattern to a class without affecting object creators. Other creational patterns put object creation somewhere other than the class being instantiated. For these, OBSTACL allows instantiators to be manipulated at run time and put into data structures, including builder and factory objects. Using instantiators in every class and allowing them to be used as first class functions facilitates the implementation of common creational design patterns.

6.2.1 Modular Construction

The author of a subclass is in some sense a user of the superclass, albeit with more access and knowledge of the class workings. The subclass author does not know all the details of the superclass. Minimizing the level of detail that he must know helps improve maintenance—the superclass author can change some details without affecting the subclasses. Private fields can be changed in this way. This hiding is only possible if object creation is modular. A subclass implementation should not be responsible for initializing inherited fields when a new object is created, since some of the inherited fields may be private and thus invisible to the subclass. Also, the definitions of inherited fields may change when the class hierarchy changes, making

[4] These are usually implied but not explicitly stated.

[5] One reason for non-virtual methods in C++ is that the program is easier to reason about if one has to consider only current behavior and not an infinite variety of possible future definitions.

```
class File
    ...
    constructor make(filename) ...;
    instantiator make(filename) =
        if substring(filename, 0, 1) = "|"
            then Pipe.open(substring(filename, 1, length(filename)))
            else new File open(filename);
end;
```

Figure 6.2: A virtual constructor

the subclass implementation invalid. In OBSTACL, each class's constructor is responsible only for initializing the definitions new to that class. It then invokes a constructor of the superclass. This approach is used in many object-oriented programming languages, including C++ and Java. When superclass implementations can be changed without requiring changes to the subclass, changes remain local to the superclass, making maintenance easier. In OBSTACL, this form of constructor is essential for mixins. The mixin definition does not have any knowledge of the (private) fields provided by the subclass, which may not even be written yet. Thus the mixin constructor cannot initialize these fields. Modular construction is desired for ease of maintenance with conventional inheritance, and is necessary for implementing mixin inheritance.

6.2.2 Virtual Constructor

A virtual constructor is not a constructor in the language, but a function that chooses a class to instantiate based on its arguments [Cop92, sections 5.5 and 8.2]. It then either creates an object directly, calls another function to create an object, or raises an exception if no object is to be created. For example, the Perl `open` function opens a file unless the filename begins or ends with "`|`", in which case it sets up a subprocess and returns the stream used to communicate with that subprocess. Figure 6.2 shows how such a rule might be written in OBSTACL. If the filename begins with "`|`", the instantiator calls another instantiator; otherwise, it creates a File object.

Virtual constructors are also useful for persistent or network objects. When writing the object to disk or network, a "tag" representing the object's class is written and then the object's fields are written. A superclass instantiator can read the tag and call an instantiator for one of its children, which reads data from the stream and calls an appropriate constructor to create the object. Another use of virtual constructors is in implementing parameterized factory methods as described in [GHJV95].

Similar in structure to virtual constructors, private classes are classes that can be instantiated but are never seen by the object creator. In Dylan [Com92], a class can be abstract

```
class BackgroundDNS
    field working : CondVar;
    field answer : string ref;
    method is_ready() = not working.blocked();
    method answer() =
        working.wait();
        !answer;
    constructor start(hostname:string)
        fields {
            working = CondVar.make(),
            answer = ref " "
        }
        inititialization
            Thread.make(
                fn() => (
                  answer := DNS.resolve(hostname);
                  working.signal()
                )
            )
    instantiator lookup(hostname:string) = new start(hostname)
end;
```

Figure 6.3: DNS lookup class

(meaning it cannot be instantiated) but still have an instantiator, which creates an instance of a private class that is not exported from the module. The object creator can create instances of the private class without being able to see it. In OBSTACL, this pattern is straightforward to implement. An alternative approach is to dispense with the abstract class and to export the instantiator alone.

6.2.3 Object Cache

An instantiator is not required to create an object—only to return one. Particularly useful with objects that do not keep state, an object cache keeps a set of commonly requested objects and allows reusing them instead of creating new objects.

An example where an object cache may be useful is DNS lookups, which can be very slow, and may be run in a separate thread (see figure 6.3). A typical user would create a `BackgroundDNS` object to spawn a lookup, then later call `answer()` to get the answer. Later, the author of `BackgroundDNS` may notice that users often look up the same host multiple times. Instead of spawning a new thread, the instantiator can return an existing lookup object (see figure 6.4). Users need not change their code.

Instantiators are useful for other creational design patterns as well. An object cache is a generalization of a singleton. Instead of creating only one instance, an object cache creates one

```
    instantiator lookup =
      let cache = Cache.new() in
         fn(hostname : string) =>
            if cache.has_key(hostname)
            then cache.get(hostname)
            else
               let obj = new start(hostname)
               in (cache.put(hostname, obj); obj)
               end
```

Figure 6.4: Caching DNS lookup instantiator

```
class Singleton
    ...
    instantiator make =
        let val r = ref (fn() => new Singleton make())
        in
            fn () =>
                let val x = (!r)() in
                    (r := fn() => x; x)
                end
        end
end;
```

Figure 6.5: A singleton

instance for each "key" (which often is the set of arguments to the instantiator). For example, an object cache may create at most one network connection to each database machine, and then return existing connections for reuse. The returned objects typically have no mutable state (see the Flyweight pattern [GHJV95]) so that they can be shared among many clients.

A class that should have at most one instance is a singleton class [GHJV95]. A singleton class is a special case of an object cache, where the one object is *always* reused. The instantiator for the class will create a new object the first time it is called, but each subsequent call to the instantiator will return the same object. Figure 6.5 shows a singleton instantiator that makes a local function to create objects, and replaces it with a function that returns an existing object. The local ref `r` is created once, when the class is declared, so it can be used to store shared values shared from one call of the instantiator to the next.

6.2.4 Factory

A collection of factory methods can be placed in a separate object, called an Abstract Factory. The factory object can be shared by many program components that need a shared set of classes. For example, a windowing toolkit might offer more than one look and feel (Java metal

```
class AWTFactory
   method button = AWTButton.make;
   method scrollableEditor() =
      let scrollbarPos = Subject.make(0) in
         AWTRow(AWTEditor(scrollbarPos), AWTScrollbar(scrollbarPos))
      end
   ...
end;
```

Figure 6.6: Abstract factory

[Zuk97], Tcl/Tk [Ous94], and Motif [OSF91]). A factory class for each appearance can direct each window creation request (dialog box, menu, button, etc.) to the appropriate windowing library. In figure 6.6 the methods of `AWTFactory` direct requests to instantiators. Object creators no longer depend on the entire set of classes (`AWTButton`, etc.) but only on `AWTFactory`. In addition, some methods may create objects by combining other objects—for example, `ScrollableEditor` combines a scroll bar object and an editor object. Object creators are insulated from the details of creating related objects, so different factories can be used instead.

6.2.5 Remote Object

In the remote proxy pattern, an object is created in a separate address space and communicates with a proxy object created in the original address space. The communication is hidden from the user of the object. In OBSTACL, the instantiator can choose whether a service should be local or remote, and instantiate the appropriate object. This decision can be made by the class implementor without changing the interface to the rest of the program.

6.2.6 Prototype

In the prototype pattern, each new instance of a class is not initialized directly, but instead initialized to be a copy of an existing object. One aspect of prototypes is to be able to encapsulate a class and initial values for objects. OBSTACL supports two alternatives to express this aspect of prototypes:

- At run time, create functions to encapsulate a call to a class's instantiator and the arguments to the instantiator.
- At run time, create classes that use the local environment for field values. Figure 6.7 shows an example of creating classes dynamically. Each time `mkclass` is called, a new class is returned. Instances of that class can access the arguments passed to `mkclass`.

These alternatives use OBSTACL's support for creating functions and classes at run time to play the role of a prototype object. However, neither alternative matches the second aspect of

```
fun mkclass(x,y,z) =
   class C
      method get_x() = x
      instantiator make() = new C make();
   end;
```

Figure 6.7: Dynamic class creation

prototypes: to be able to use the prototype as an object, and *then* create another object from it. (See section 9.4.9 for issues related to cloning.) OBSTACL directly supports some aspects of prototypes but requires the programmer to write a `clone()` method for the full power of the Prototype pattern.

6.2.7 Multi-stage Construction

In some windowing libraries, a widget requiring extensive customization is constructed using a two-step process [Int94, OSF91]. First, its constructor is called. Then it is customized by allowing the client to set fields such as *x*, *y*, *fg* (foreground), and *bg* (background). Finally, a method such as `RealizeWidget` or `SetupWindow` is called to complete the construction process. Since the constructor and setup functions must occur in pairs (*i.e.*, an object construction without calling the setup function or calling the setup function more than once is an error), it makes sense to abstract these into one unit. However, between the two calls is user code. C++ has no convenient way to pass code to a function, so the construction process must be split into several methods. In contrast, in a language with first class functions, we can pass in a function that maps a record (e.g., {*x, y, fg, bg*}) with default values to a record with new values. The function is able to customize the widget during the construction process without exposing a partially initialized object to the client.

In the Subject/Observer design pattern [GHJV95], the observer must register itself with the subject at the time of construction. Even if we can guarantee that the observer's constructor does not call any virtual functions, we don't know anything about the subject. It may try to call a virtual function before the construction process completes. Even if we can guarantee that `Subject.register()` does not call any virtual functions, a multithreaded program may have another thread that calls a virtual function on the unfinished object.

An object that is constructed in stages (alternating between class code and user code) is typically partially constructed in the constructor, leaving the object in a "partial birth" state that may not satisfy the object invariants. Alternatively, the object invariants must be loosened to deal with the extra partial construction state. In a language with first class functions, the user stages of construction can be passed in as arguments to the constructor or instantiator. When the instantiator returns an object, it is in a fully constructed state with all its invariants satisfied.

6.3 Class Creation Patterns

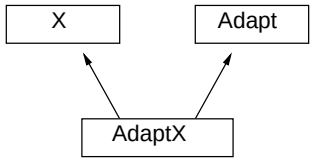
Structural design patterns describe relationships between objects or between classes. In OBSTACL, interfaces between objects are expressed in structural object types, which allow the use of new objects with implementations unrelated to the original implementation being used. Substitutivity makes easier the addition of design patterns to existing systems. In particular, adapters, composites, decorators, and proxies require program components to use an interface instead of a specific class. Using interface types by default lowers the resistance to using object-based structural patterns.

Class-based structural patterns use inheritance as the relation between classes. In OBSTACL, mixins can be used to make these patterns more flexible. The class-based version of the Adapter pattern is used to adapt a class from one type hierarchy for use in a different type hierarchy. The object-based version of Adapter is preferred over the class-based version when "you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one" [GHJV95]. The class-based version offers two advantages: subclassing allows one to override behavior for self calls properly, and there is less overhead (the object-based version introduces a new object and extra levels of indirection). As described in [GHJV95], object-based Decorator pattern has three main advantages and two disadvantages when compared to the alternative, subclassing. Mixins give us two of these advantages without either disadvantage. In addition, mixins can be combined with *forwarders* to produce the object versions of the class-based patterns. The programmer can easily choose the most appropriate form. In section 6.3.1 we look at adapters and in section 6.3.2 we look at decorators.

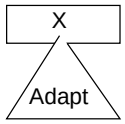
6.3.1 Adapter

The Adapter pattern is used to allow objects with a type in one type hierarchy to be used in a different type hierarchy. The class-based adapter takes a class and modifies it to present a different interface. The resulting class can typically be used in either hierarchy. In C++, multiple inheritance is used to combine an adapter class with a class from one hierarchy (see figure 6.8). For each class `X`, the programmer must create a new class `AdaptX` that inherits both `X` and `Adapt`. The adapter belongs to the new hierarchy and can translate calls using the new interface into calls using the old interface. The disadvantage of this pattern is that the user has to create a new adapter class for each class in the old class hierarchy.

OBSTACL allows mixin adapters, which are implemented with parameterized inheritance instead of multiple inheritance (see figure 6.8). For each class `X`, the programmer applies the mixin `Adapt` to class `X` to produce a new class `Adapt(X)`. Although a new class is created, there is no class *declaration*, along with constructors and any necessary ambiguity resolution. Thus mixin adapters in OBSTACL do not suffer from the same disadvantage (relative to object-based adapters) as their multiple-inheritance-based counterparts.



Multiple Inheritance



Mixins

```
// Adapter code
class Adapt {
   ...
};


      ...

// User code
class AdaptX extends Adapt, X {
   // Create new constructors
   AdaptX(a, b, c): Adapt(a), X(b, c) {}
   AdaptX(a, d): Adapt(a), X(d) {}
};
new AdaptX(1, 2, 3)
```

```
// Adapter code
mixin Adapt(X) extends X {
    ...
};


      ...

// User code



 Adapt(X)(1, (2, 3))
```

Figure 6.8: Class adapters using multiple inheritance vs. mixins

```
class Adapter
   field target :
      {| get:int->'a, set:'a*int->unit, highestIndex:unit->int |};
   method getElement = target.get;  (* rename *)
   method setElement(a,b) = target.set(b,a);  (* swap arguments *)
   method count() =
      1 + target.highestIndex(); (* derived information *)
   constructor make(dest)
      fields { target = dest }
   instantiator make;    (* standard instantiator *)
end;
```

Figure 6.9: Object adapter

Still, in some situations an object-based adapter may be desired. The object-based version uses an object compatible with one hierarchy to translate and forward messages to a second object compatible with a different hierarchy. Figure 6.9 shows an object adapter: each method translates the message and forwards it to another object. Unlike the C++ class adapter, the object adapter works for all classes in the original class hierarchy, [6] so no additional work is needed if the class hierarchy grows. Although the mixin adapters do not suffer from this problem, the object adapter still offers one advantage: it works on existing objects, while the mixin adapters work only for newly created objects. Since there are many more object users than object creators, the object adapter is useful in more situations than the mixin adapter. However, it is subject to several drawbacks:

- Object adapters introduce additional objects, which pose problems with object identity. Lists of objects may have one object or another, but not both, so the programmer must add code to keep containers consistent.
- Object adapters introduce an extra level of indirection. When multiple adapters are needed, the overhead increases linearly.
- Object adapters cannot override methods in the target. Although the object adapter can provide its own version of a method, it isn't called by other methods in the original object.

Since neither mixin nor object adapter is always better than the other, the programmer must choose which form of adapter is more appropriate in each case.

In OBSTACL, object adapters can be created from mixin adapters. A mixin adapter can be applied to a ***forwarder class*** to produce an object adapter. A forwarder class has one field, which points to another object (see figure 6.10). The methods of the forwarder class pass the message to the second object, without performing any translation. The adapter mixin performs translation and calls the superclass methods. The combination is an object adapter, which performs translation and then passes the message to another object.

In OBSTACL, mixin adapters can be used instead of class adapters. Object adapters are mixin adapters applied to forwarder classes. The programmer can easily choose between mixin and object adapters. In addition, the adapter can be used with other variants of forwarders, such as remote forwarding proxies (see 6.2.5), and the forwarders can be used with other patterns, such as Decorator, to transform a class-based pattern into an object-based one.

6.3.2 Decorator

In traditional class-based programming, capabilities are added to objects by subclassing. However, in most object-oriented languages, subclassing is too static and cumbersome to use

[6] For this discussion we assume the classes implement the same interface. In practice, one adapter is needed per *interface*.

```
class Forwarder
   field target : {| draw:unit->unit, resize:int*int->unit |};
   method draw = target.draw;
   method resize = target.resize;
   constructor make(dest)
      fields { target = dest }
   instantiator make;    (* standard instantiator *)
end;
```

Figure 6.10: Forwarder class

in situations in which many combinations of features are needed. An object-based decorator can add behavior to a set of classes without subclassing each one, can be added or removed at run time, allows the programmer to control the order in which modifications are performed, and allows modifications to be applied any number of times. The implementation of object based decorators is similar to that of object-based adapters: a decorator object points to another object, and methods of the decorator perform additional actions before and after passing the message to the target object. However, where adapters perform the same task with a different interface, decorators perform a different task with the same interface. Decorators suffer the same disadvantages as object adapters: multiple objects are more difficult to work with, forwarding imposes an overhead even for methods that are not modified, and calls from the target object to itself do not go through the decorator. The object-based decorator is a good alternative to class-based extension (subclassing) but neither is always better than the other.

To illustrate the use of decorators, we present an example using scoops of ice cream. Suppose we would like to represent an ice cream cone with any number of scoops of ice cream on top. Using class decorators (subclasses), we might write a hierarchy like that in figure 6.11. The empty ice cream cone is represented by the Cone class, and its subclasses represent an ice cream cone with a scoop of chocolate or vanilla ice cream. To have one scoop of each, we could use multiple inheritance. However, all possible combinations must be created at compile time and most forms of multiple inheritance do not allow repetition of classes (for example, to create an object representing *two* scoops of chocolate ice cream).

With object decorators, we can choose at run time what combination of ice cream scoops to put on a cone. For example, given a list `[Vanilla, Vanilla, Chocolate]`, we can use a loop such as the one shown in figure 6.12 to build an ice cream object with the desired flavors.

With mixin decorators, we can write the same loop, but the result for N flavors will be a single object instead of $N + 1$ objects. Figure 6.13 contains pseudocode for building a class from a list of mixins and then instantiating that class.

OBSTACL's mixins provide a solution between object decorators and static inheritance. Mixin-based decorators add behavior to a set of classes, allow control of the order and number

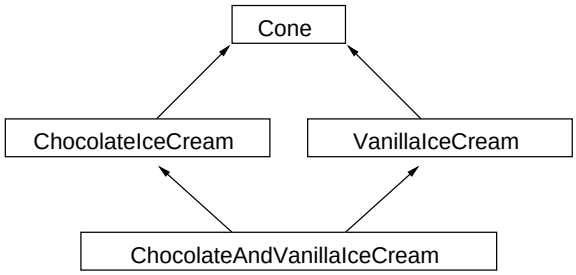


Figure 6.11: Ice Cream represented with multiple inheritance

```
obj := new Cone;
for each flavorClass in list:
    obj := new flavorClass(obj);
```

Figure 6.12: Ice Cream object built from a list of flavor classes

of times decorators are added, result in one efficient object instead of a chain of object pointers, and route calls from the object through the modifications made by the decorators. However, like class decorators, they cannot be added or removed after the object is created. Summarized in table 6.1, mixin decorators share all the advantages of class decorators and most of the advantages of object decorators. As with adapters, using a mixin decorator with a forwarder class produces an object decorator.

6.4 Multiple inheritance

In many languages, classes can be built by inheriting from more than one class. Multiple inheritance introduces many complexities and remains a controversial feature. In this section we look at structures built with multiple inheritance and how they might be represented in OBSTACL.

6.4.1 Possible approaches

First we will look at what other languages provide in the way of multiple inheritance.

```
class := Cone;
for each flavorMixin in list:
    class := flavorMixin(class);
obj := new class;
```

Figure 6.13: Ice Cream class built from a list of flavor mixins

	Object	Class	Mixin
added after object created	Yes	No	No
removed after object created	Yes	No	No
control order of application	Yes	No	Yes
apply more than once	Yes	No	Yes
avoid combinatorial explosion of declarations	Yes	No	Yes
single object (easy to work with)	No	Yes	Yes
self calls routed through decorators	No	Yes	Yes
low overhead for unaffected methods	No	Yes	Yes

Table 6.1: Object, class, and mixin decorators

Rely on the programmer. In general, C++ relies on the programmer to specify the semantics of multiple inheritance. The programmer can refer to any inherited method using base class name as a prefix to bypass normal inheritance rules or to resolve ambiguities [ES90]. C++ takes the view that name conflicts between inherited methods are program design flaws that have to be addressed explicitly by the programmer [Cop92]. Therefore, C++ does not provide a rule for resolving ambiguities. Attempts to call an ambiguous method are signaled as errors at compile time. The standard solution is to define a new method with the same name in the derived class and in the body of this method call the conflicting inherited methods in some order. This puts the burden of specifying the order on the programmer. A similar but more general approach in Eiffel requires renaming in the case of ambiguous methods. Note that in C++, if conflicting methods are inherited from base classes but not called by the derived class's clients, no error is signaled [ES90].[7]

C++ does not provide a good solution to the problem of redundant method calls in diamond inheritance, as shown in figure 6.11. The approach suggested by the designer of the language [Str97] essentially requires the implementor of `ChocolateAndVanillaIceCream.eat` to explicitly call special "helper" methods of all ancestor classes without relying on calls to inherited `eat` methods. This approach in some sense violates the modularity of class extension, since a class implementor has to know the class hierarchy beyond his class's immediate parents and refer directly to methods implemented in ancestors. This kind of design is sensitive to changes in the class hierarchy and makes class libraries fragile. The code that uses the library depends not only on the terminal classes, but also on the entire class hierarchy, which makes it difficult to change the implementation without affecting *some* user of the hierarchy.

[7] In C++, the ambiguity is not in the class definition but in the method call. The caller can resolve the ambiguity by specifying which base class's definition should be used.

Linearize the hierarchy. CLOS [Ste90] and other object-oriented descendants of Lisp solve the ambiguous methods problem by linearization. The listing order of base classes is considered significant, and methods from the classes that are listed earlier take precedence over methods with the same name from the classes that are listed later. This approach is also sensitive to changes in the class hierarchy [Sny86] and sometimes leads to unexpected results. Some of the problems with the CLOS linearization algorithm are fixed in Dylan [BCH+96].

Imposing an arbitrary ordering on base classes (CLOS-style) is not a very good approach because it is sensitive to rearrangements of the class hierarchy [Sny86, VRTB98] and introduces non-local errors—local changes to one part of a class hierarchy can affect distant parts of the hierarchy. Any ordering should be made explicit by the programmer to avoid unexpected behavior. The tradeoff between automatic ordering and explicit ordering is one of initial implementation vs. maintenance: explicit ordering has a cost up front but avoids fragility and subtle errors while maintaining the program.

Avoid multiple inheritance. Early object-oriented languages such as Simula67 [DMN70] and Smalltalk-76 [Ing78] had only single inheritance. Since the experience with multiple inheritance accumulated by C++ users does not lead to any definite conclusions, the designers of Java decided that the complications introduced by multiple inheritance outweigh the benefits and did not include it in the language [AG96].

It is worth noting, however, that Java does provide a mechanism for multiple *interface subtyping*. In this section, we intend to demonstrate that common multiple inheritance designs can be expressed using single inheritance, interface subtyping, and parameterization. Java currently does not provide any form of parametric polymorphism, although this may change in the near future [BLM97, OW97].

6.4.2 Multiple Interfaces

Multiple inheritance is often used when a single class must support more than one interface. Some languages, such as Java and OBSTACL, support this use but not other uses of multiple inheritance. Multiple interfaces in Java allow the programmer to declare that a class satisfies one or more interfaces. Structural subtyping in OBSTACL allows a type to have several potentially unrelated supertypes. For example, a class might implement both the `Editor` and `Stream` interfaces, so that its instances can serve both as an editor window and as a stream of characters (see figure 6.14). To do this, it can can inherit from an `Editor` window class (E) and then add any code required to support the `Stream` interface. This class is declared to implement `Stream`, so objects of `StreamableEditor` can be used in functions that require `Streams`. When those functions call a `Stream` method, it will be redirected to the appropriate `Editor` method by the forwarding functions.

```
class StreamableEditor extends Editor /* SE */
  implements Stream; /* second supertype */
  method read() = getText();
  method open() = lockEditBuffer();
  method close() = unlockEditBuffer();
end;
```

Figure 6.14: Streamable editor class

```
class StreamableHTMLEditor (* SH *)
  extends HTMLEditor;
  implements Stream;
  method read() = ...;
  method open() = ...;
  method close() = ...;
end;
```

```
class StreamableLaTeXEditor (* SL *)
  extends LaTeXEditor;
  implements Stream;
  method read() = ...;
  method open() = ...;
  method close() = ...;
end;
```

Figure 6.15: Specialized streamable editor classes

We may want to do the same for subclasses of `Editor` such as `HTMLEditor` (H) and `LaTeXEditor` (L). Without multiple inheritance, it is not possible to inherit the forwarding functions from `StreamableEditor`, so these methods must be reimplemented (see figure 6.15). If multiple inheritance is available, `StreamableHTMLEditor` can inherit from both `StreamableEditor` and `HTMLEditor`. It inherits the forwarding functions from `StreamableEditor` and the HTML editing extensions from `HTMLEditor`. Similarly, `StreamableLaTeXEditor` can be implemented without reimplementing the forwarding functions. A class that requires a different interface can inherit its implementation from one class and also inherit from an class-based adapter (see section 6.3.1). Alternatively, we could use mixins. The adapter code is put into a mixin adapter and then added whenever needed, as shown in figure 6.17.

Figure 6.16: Multiple inheritance hierarchy compared with mixin hierarchy

```
mixin StreamAdapter extends W with Editor /* S */
  method read() = getText();
  method open() = lockBuffer();
  method close() = unlockBuffer();
end;


class StreamableEditor =
  StreamAdapter(Editor);
class StreamableHTMLEditor =
  StreamAdapter(HTMLEditor);
class StreamableLaTeXEditor =
  StreamAdapter(LaTeXEditor);
```

Figure 6.17: StreamAdapter mixin and its application

The multiple inheritance hierarchy and the mixin hierarchy are shown in figure 6.16. Classes are shown as circles and mixins are shown as triangles. A mixin application is shown as a mixin attached to either a class or to another mixin application.

Even in the case where a class needs only to support multiple interfaces but not to inherit code from more than one parent class, multiple inheritance can be useful to inherit the code needed to forward messages. Using mixins is more convenient than multiple inheritance in this case because a new class does not have to be defined—classes such as `StreamAdapter(HTMLEditor)` and `StreamAdapter(LaTeXEditor)` can be used directly.

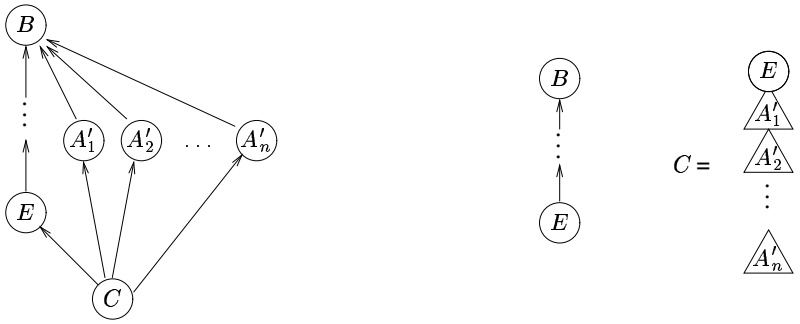


Figure 6.18: A set of adapters using multiple inheritance vs. using mixins

In general, whenever a class C must support multiple interfaces $A_1, A_2, \ldots, A_n$ and inherit code from base class E, multiple inheritance is used with adapter classes A'_i that translate the interface of some class B to the interface A_i (see figure 6.18). Class C inherits from E and $A'_1, A'_2, \ldots, A'_n$. Given this hierarchy, mixins can be used instead to create adapters A'_i to take any class supporting the interface of B and produce a class that also supports interface

```
class Z (* multiple inheritance *)
     extends X,Y
     redefine method m() = X.m();
end;
```

Figure 6.19: A class inheriting multiple unrelated implementations

A_i. Then class C is written as $A'_1(A'_2(\ldots A'_n(E)\ldots))$. Mixins A'_i can also be reused with other classes that support B's interface.

An alternative to adding interfaces to class C is creating a separate object with one or more interfaces. The object-based adapter can forward all requests to the original object, after translating from the A_i interface to the B interface (see section 6.3.1). Whenever a translation object is needed for some class B, a forwarding class F can be written for B's interface. This class has one private field, a pointer to an instance of C. It also defines methods to forward requests to that instance.[8] The mixins A'_i are applied to the forwarding class, producing $F' = A'_1(A'_2(\ldots A'_n(F)\ldots))$. All messages sent to instances of F' are translated from the A_i interface to the B interface, then forwarded to a B-interface satisfying object, which in this case is an instance of class E.

Structural subtyping can be used whenever objects from one hierarchy (A) need to be used as if they belonged to a different hierarchy (B). Any method that has to be supported by objects from B can be added to a class in A with a body that simply redirects the call to another method in A. The new class will satisfy both the interface specified by A and the one specified by B. The type of the new class is a subtype of both A and B, but the class does not inherit from both A and B. Structural subtyping can be viewed as providing multiple interface inheritance without multiple implementation inheritance.

6.4.3 Unrelated Superclasses

The inheritance of multiple unrelated implementations is often used as a convenient form of aggregation instead of for good design purposes [VRTB98]. Nevertheless, there may be situations in which it is needed, so we look for an alternative in OBSTACL. Multiple implementation inheritance can be simulated with composition and exporting of interfaces. Let X and Y be classes from which we would like to inherit. Let x be a method of X that does not occur in Y; similarly, let y be a method of Y that does not occur in X. Furthermore, let m be a method present in both X and Y. Consider a class Z (shown in figure 6.19) that inherits from both X and Y. We can instead use a class Z' (shown in figure 6.20) that contains an X and a Y. The interface of Z' is the same as that of Z, so clients need not know whether multiple inheritance is being used. This technique is similar to multiple interface inheritance.

[8] A forwarding class may perform additional functions, such as sending messages to another address space, providing protection for the original object, or handling reference counting. See the *Proxy* pattern for details [GHJV95].

```
class Z′ (* composition *)
      private field px : X;
      private field py : Y;
      public method x() = px.x();
      public method y() = py.y();
      public method m() = px.x();
end;
```

Figure 6.20: Composition and forwarding simulates multiple inheritance

```
class Y′ (* Y with cross-calls *)
      extends Y
      private field z : Z′ ref;
      redefine method m() = (!z).m();
end;
```

Figure 6.21: Cross calls provide dynamic lookup with composition and forwarding

One problem remains. If a method of Y calls m, it sees Y's m, not X's m, as would be the case with Z. Here we introduce a *cross-call*. The class Y has to make a call *across* its implementation boundary, into X's side. Now Z must contain an X and a Y' (see figure 6.21). If X needs cross calls, an X' would have to be created as well. Cross-calls are inconvenient, but in the rare case that multiple implementation inheritance of this flavor is needed, it can be simulated in this way.

6.4.4 Related Superclasses

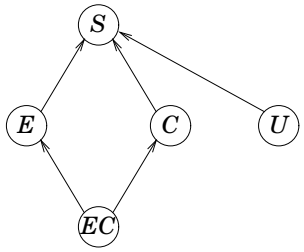
A common use of multiple inheritance is to combine classes related by sharing a common ancestor. Each of the classes extends the ancestor with some feature. The goal is to create a new class with several features. With multiple inheritance, that class extends each of several classes, which each extend a common ancestor. The resulting class graph looks like a diamond, so it is sometimes called "diamond-shaped multiple inheritance".

For example, a stream class may be extended with features such as compression or encryption. Less obvious features include object locking and logging capabilities.[9] The class implementing the feature (a "mixin") and its methods either replace or add functionality to the code in the parent class. For example, the `read` method in the `EncryptedStream` class would read from the stream by calling the parent's `read` method, and then decrypt the data. The `write` method would encrypt the data and then pass it on to the `write` method in the parent stream class.

To use more than one feature, multiple inheritance can be used to inherit from the classes

[9]For example, a mixin could override each method to write information to a log file and then call the original method. This mixin could be added during the debugging process to and removed once the debugging phase is complete.

```
class EncryptedStream extends Stream /* E */
  method read() =
        data = super.read();
        /* decrypt data */
        /* return data */
  method write(data) =
        /* encrypt data */
        super:write(data);
end;
```




```
class CompressedStream /* C */
  extends Stream
  /* Similar to EncryptedStream */
  method read() = ...
  method write(data) = ...
end;
```

```
class UUEncodedStream /* U */
  extends Stream
  /* Similar to EncryptedStream */
  method read() = ...
  method write(data) = ...
end;
```

```
class EncryptedCompressedStream /* EC */
  extends EncryptedStream, CompressedStream
end;
```

Figure 6.22: Encrypted, compressed, and uuencoded streams

that have implemented the feature. The code and class hierarchy in figure 6.22 shows the "diamond" shape that results. The class on top (Stream in this case) is inherited by the class on the bottom (`EncryptedCompressedStream`) through more than one path.

Even if the language provides a suitable "super" call mechanism, the methods may not be called in the desired order. Some languages, such as CLOS and Dylan, provide automatic linearization; in those, the programmer must be aware of the algorithm used for linearization so that he or she can ensure that compression occurs before encryption, and encryption occurs before uuencoding. Other languages, such as C++, do not provide linearization, and instead, each method that requires linearization has to be written again in the class that inherited several mixins. Languages with multiple inheritance offer many different ways of handling the situation with multiple mixins, but they either require extra code to be written for every combination of mixins or are error-prone in the sense that changes to the class hierarchy may lead to unexpected changes in behavior.

With mixins, each extension of the common base class becomes a mixin that can be applied to one of the non-mixin classes. In our example, `Encrypt`, `Compress`, and `UUEncode` would be mixins that could be applied to any `Stream` class. Figure 6.23 shows the code and class hierarchy that results.

Instead of writing code for each combination of extensions, the mixin approach allows us to name each combination without writing any code. It also allows us to use a combination of extensions without naming it, such as for `ueKStream` above. Mixins give us control over the order in which the extensions are assembled. We can make sure that the user uuencodes an encrypted version of a compressed stream, rather than a compressed encrypted uuencoded stream.[10] In addition, mixins are not limited to being applied once. A stream could be encrypted *twice* by using `Encrypt(Encrypt(Stream))` (see figure 6.24). This would not be possible with most multiple inheritance systems because classes (such as `EncryptedStream`) cannot have copies of themselves as siblings. The advantages of mixins over multiple inheritance illustrated in this example are the ability to control the exact order of feature addition, the ability to use a combination of features without writing any additional code, and the ability to apply a feature more than once.

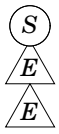


Figure 6.24: Double encryption

In general, whenever features $A_1, A_2, \ldots, A_k$ are needed to add functionality to class that inherits B, they can be implemented as mixins that take as input any class that satisfies the interface of class B. Any class E that had $C, A_{i_1}, A_{i_2}, \ldots, A_{i_n}$ as parents and added new fields and methods is first split into two classes E and E' (see figure 6.25). Class E' inherits from $C, A_{i_1}, A_{i_2}, \ldots, A_{i_n}$ but adds no fields or methods. Class E inherits from E' and adds the fields and methods. With mixins, class E' is replaced by the mixin application $A_{i_1}(A_{i_2}(\ldots A_{i_n}(C)\ldots))$,

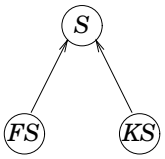
[10] UUencoding is needed when we need to transmit a file in ASCII. Encrypting an ASCII stream would lose its 7-bit property, so it is pointless to encrypt after uuencoding. Similarly, compression can be used well when there are patterns in the file. Encrypting the file destroys any patterns, so compression should be done before encryption. However, encrypting a compressed file may make it easier to decrypt, since many compression programs add a known byte sequence header that can be used in the decryption process.

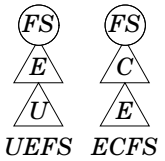
```
type AnyStreamType = classtype
    method read: unit → array of α;
    method write: array of α → unit;
    method eof: unit → boolean
end


mixin Encrypt extends S with AnyStreamType /* E */
  method read() =
        data = super:read();
        /* decrypt data */
        /* return data */
  method write(data) =
        /* encrypt data */
        super:write(data);
end;

mixin Compress extends S with AnyStreamType /* C */
  method read() = ...
  method write(data) = ...
end;

mixin UUEncode extends S with AnyStreamType /* U */
  method read() = ...
  method write(data) = ...
end;
```






```
class FileStream; /* FS */
class KeyboardStream; /* KS */
class UUEncodedEncryptedFStream = UUEncode(Encrypt(FileStream)); /* UEFS */
class EncryptedCompressedFStream = Encrypt(Compress(FileStream)); /* ECFS */
val ueKStream = UUEncode(Encrypt(KeyboardStream)).new()
```

Figure 6.23: Stream class hierarchy expressed with mixins

and class E inherits from it. A simpler case is when all the additions are symmetric, and there is no special C class; in this case, $C = B$.

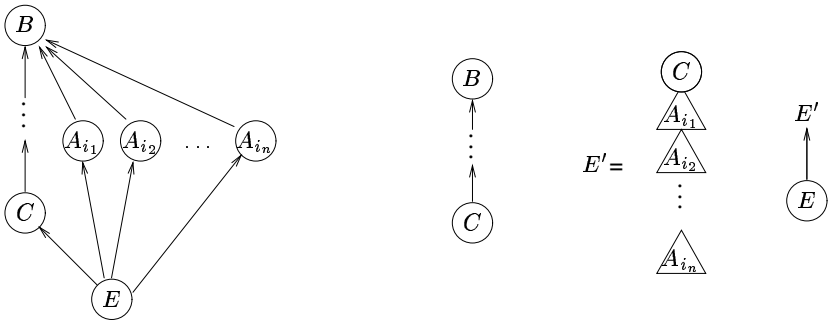


Figure 6.25: A set of features using multiple inheritance vs. using mixins

The *Decorator* design pattern [GHJV95] is similar to mixins, but applies to features that can be added or removed dynamically. Decorator classes can be created with mixins $A_1, A_2, \ldots, A_k$ and the forwarding class F described in section 6.3.2. A decorator class for a single feature i would be $A_i(F)$. With mixins one can also combine multiple features statically to produce a new decorator class by applying a mixin A_j to a decorator class.[11] For example, $A_j(A_i(F))$ forms a decorator from the composition of A_i and A_j. The use of forwarding classes makes it possible to produce both static extensions and dynamic decorators from the same mixins.

6.4.5 Conclusions

Most cases where multiple inheritance would be used by a C++ programmer can instead be expressed using two simpler concepts: *structural subtyping* and *parametric inheritance* (mixins).

The examples above show how common designs employing multiple inheritance can be implemented using only mixins and single inheritance (assuming that interface subtyping is available in the language). In general, the most fruitful approach is to design class hierarchies with mixins in mind rather than trying to convert an existing multiple inheritance hierarchy into the mixin form. The main reason is that the order of mixin application should be determined by design requirements (e.g., compression should be applied before encryption) rather than an algorithm that considers only the listing order of base classes.

However, if the need should arise, an arbitrary class hierarchy using multiple inheritance can be converted into a sequence of mixin applications using a monotonic superclass linearization algorithm such Dylan's proposed C3 algorithm [BCH+96]. For any leaf class A, all of its ancestors $A_1, \ldots, A_n$ are linearized from the most specific to the least specific. Then for all i,

[11] With mixin composition (see 9.1.13), one would be able to produce not only new decorator classes (which produce object decorators) but also new decorator mixins (which produce class decorators).

class A_i is turned into a mixin whose parameter constraint is inferred as specified in section 5.3.2 based on how the inherited methods are used in the class body. Finally, A is defined to be the result of a sequence of mixin application $A_1(A_2(\ldots A_n(E)\ldots))$ where E is the empty class or the root of the class hierarchy.

Many problems that seem to require multiple inheritance can be solved with an appropriate use of object composition, single inheritance, and mixins (consider, for instance, most design patterns in [GHJV95]). A diamond-shaped hierarchy can (usually) be expressed as a single inheritance hierarchy and mixins. A V-shaped hierarchy can be expressed as a single composite object with cross-calls or as an object from one hierarchy with extra interface added to make it appear as if it belonged to another hierarchy. The process of adding the necessary interface can be automated using mixins.

6.5 Design Principles

Having seen examples of design and maintenance problems that can arise in object-oriented programming, we now describe some general guidelines for programming in OBSTACL.

6.5.1 Hide Details

A user of an object should be insulated from the details of how the object is implemented. The first rule is well accepted in the object-oriented programming community [Inc94, Boo94]:

> *Keep the object's "physical" state hidden; provide accessors for the "logical" state.*

For example, an object representing a file may be implemented with a pair ⟨block, offset_in_block⟩ representing the current file position, but exports an accessor that returns an integer offset into the file. The concept and implementation of "block" is hidden.

The same principle applies to *creating* objects:

> *Keep details of object construction hidden in instantiators and constructors.*

For example, an object representing a TCP connection may have to determine an IP address and compute a route to the destination. Using a separate method to compute the IP and route introduces a new state—"route not computed" (*i.e.*, object not fully ready to use)—which adds complexity to the interface. Each regular method (*e.g.*, `send`, `receive`) is valid only in the fully constructed state. Both the user of the object and the author of the class must keep the state in mind before calling a method. When the added state is not useful to the user, move it into the constructor. The new state is then never seen by the user, and the interface is consequently simpler. The partially initialized state is not merely a hypothetical situation; it

is seen in commercially available libraries. The OWL toolkit from Borland [Int94] uses two steps for some types of window components: first, the constructor is called to initialize the object in an unusable state; then, the `SetupWindow` method is called to perform the remainder of initialization. Another example is the Motif widget set [OSF91] for the X Window system, in which widget objects are first constructed, then "realized" in a separate step. The reason these libraries do not provide the simpler interface is that it is rather difficult to automatically provide the second construction step in C++. The designers of the Taligent framework for C++ compiled a list of workarounds [Inc94, chapter 4], including an initialization flag that is tested in every method, multiple levels of constructors, and virtual inheritance tricks. There are two reasons why the constructor cannot handle all of the initialization:

- In C++, a constructor cannot call methods redefined in a subclass. Therefore the subclass cannot customize the construction process.
- The object creator needs to customize the construction process by setting attributes or by creating related objects (such as child windows).

In OBSTACL, constructors *can* call subclass methods after all fields are initialized. In addition, since ML supports first class functions, the user can customize the process by passing arbitrary code for the constructor or instantiator to execute. Therefore it is likely that many multistep construction processes will not require separate steps in OBSTACL (see section 6.2.7).

6.5.2 Separate Functionality

A unit of abstraction such as a class may implement more than one set of operations because those sets are to be used together. For example, a text-editing widget may handle scrolling, text display, text entry, and cut/paste operations. An emailing program may handle encryption, MIME encoding, and PGP signing of messages.

Keep logical concepts in separate abstractions.

For example, break the text-editing widget into classes for display, modification, cut/paste, and scrolling. The disadvantage of this approach is that creating text widgets is inconvenient for users, but we can simply combine these components together in a wrapper class that presents a single interface to object users. The advantages of smaller components are:

- Variation. The components can be replaced to create new variations of the combined object. For example the scrolling aspect of text-editing widgets might be replaced with a slider or panning widget.
- Reuse. The components can be reused outside the object for which they were originally written. For example the scrolling functionality from a text widget is useful for list boxes as well.

Some aggregates, such as a text-editing widget, have a fixed set of dissimilar components. Others, such as the email encoding class, have a variable number of similar components.[12]

Use aggregation and interface types to combine components with different interfaces.
Use mixins to combine components with similar interfaces.

6.5.3 Use Interfaces

Use interface types, not classes

Parts of a program that use existing objects do not need to depend on the object's class at all. Since object types in OBSTACL only list the public methods (except for the type of `self`, which also lists protected methods), there is no dependency on the class of an object. In contrast, C++ and Java use classes as types by default, so object users know some superclass of the object. Since tying the object user to a particular implementation is not always desired, C++ programmers use "abstract classes" to define interfaces that have no implementation; Java supports this style directly. However, neither language uses interfaces by default; it takes extra work for programmers to use them. In OBSTACL, the situation is reversed: by default, objects are accessed through interface types and implementation types require additional work (see section 9.3.3).

Accessing objects through interfaces that are independent of classes maximizes substitutivity, a key goal of OBSTACL's design. As a result of using pure interfaces, objects can not have binary methods (see section 9.2.3), methods that use self types (see section 9.2.5), or a form of equality that compares the internals of an object (see section 4.1.3). Each of these requires that the type of the object is linked to the implementation of the object. Substitutivity facilitates the use of design patterns such as Proxy, Composite, Adapter, and Decorator, since these new kinds of objects can be added to the system without changing existing code.

Use instantiators, not constructors

Without instantiators, the object creator depends on the object creation policy. To break this dependency, programmers use virtual methods, global functions, class methods, or object wrappers. Instantiators in OBSTACL provide a consistent way to encapsulate object-creation policy with the class; without them, object creators are affected when policies are changed (*e.g.*, the default `new C(arg)` must be changed to `C::make(arg)` when the class adds a nontrivial policy). Thus, unless changes in policy were planned from the beginning, the author of a class is prevented from changing the policy without affecting object creators. OBSTACL requires the use of instantiators to allow policies to be changed in the future.

[12] A fixed set of dissimilar components corresponds to a record or tuple. A variable sized set of similar components corresponds to a list. In a typed language, lists and records are the basic data structures needed to create all others. For example, C provides only arrays and structs.

Use mixin constraints, not superclasses

A class inheriting another class makes a specific dependency between classes. Often, the link is too specific. To break the dependency, inherit from a parameter by using a mixin. Determine what is provided by the superclass and write that in the mixin constraint. Any inheritance between classes is still possible by using mixins; the mixins both break a dependency and allow for more reuse of the subclass.

6.5.4 Break Dependencies

As programs evolve, changes must be propagated to all affected parts of the program. The least disruptive change to an existing module is a change in its implementation that does not affect its interactions with the rest of the system. We want to maximize the ability to make such local changes. OBSTACL's abstractions (see section 6.5.3) provide three forms of local changes that can be made without affecting other modules. Ideally the compiler should not need to recompile parts of the program that are not affected logically. However even though the program *source* may not depend on the implementation of some module, the program's *compiled code* may exhibit such a dependency. When this happens, the program may require recompilation even though from the programmer's point of view there is no logical need to recompile. An example of this situation is most C++ compilers—a change to the implementation details of a class requires recompilation of modules that use that class, because the modules are compiled with knowledge of the class implementation and layout.[13] In chapter 8 we will look at how OBSTACL avoids mismatches between logical changes and physical (compiled code) changes.

A compiler adding new dependencies to the compiled code is a nuisance but not harmful. At worst it slows development. A more troublesome problem is implementation changes in one module causing *silent errors* in another module. These are the most dangerous, as they typically lead to neither run-time nor compile-time error—only to changed behavior. An example is automatic linearization of multiple inheritance. A slight reorganization of one part of the hierarchy can change the order in which methods are linearized, which changes the behavior of the program. Without automatic linearization, the programmer resolves conflicts, but the same problem occurs when the hierarchy changes—the resolutions must be reverified. A similar problem occurs any time the programmer explicitly bypasses the normal change of method redefinitions (accessed via `next` in OBSTACL). Consider for example a class calling a method from its ancestor, bypassing the immediate superclass definition. If the ancestor `A` is refactored into two classes `A` and `A'` then the subclass should now call A''s method. Such changes

[13] Actually most C++ compilers, with the notable exception of IBM Visual Age C++, merely look for changes to the header file rather than changes to the class layout. As a result, even changes to the text in comments will lead to a recompile.

would be even more difficult in OBSTACL, where the subclass is built at run time, and both hierarchies could coexist. (In general, the presence of mixins makes it unwise to depend on a fixed class hierarchy.) The real danger here is that the programmer who split A may have correctly notified the immediate descendants to extend A' instead of A, but may not be aware of further derived classes. The change in behavior may not even occur immediately; it may only be triggered by a later (seemingly harmless) change.

Changes to implementations should not break programs at run time.

To avoid errors like the above, we avoided adding features to OBSTACL that bypass the normal chain of events. We hide implementation details (such as private fields and constructor code) whenever possible. We also propose an implementation (in chapter 8) that makes physical dependencies as close as possible to logical dependencies. Local changes are kept local.

Although in an ideal world, all changes would be local, it is too optimistic to believe that only local changes will ever be needed. When non-local changes may be needed, it is preferable to err on the side of caution, so that any potential problem is flagged. Unfortunately a compiler cannot detect all potential problems. To help the compiler, the programmer should be precise about interfaces. Sloppy interfaces (see section 5.1.3) tend to increase reuse in the short term but lead to failures during program maintenance.[14]

Changes to interfaces should break programs at compile time.

6.6 Summary

OBSTACL includes language features not for the sake of expressiveness but to solve design problems such as those expressed as design patterns. Substitutivity of objects allows for objects to be replaced by new objects with similar behavior but different implementation. Private fields and modular object construction allow each module to hide its implementation from other modules contributing to a class hierarchy. Instantiators allow class authors to change the details of how objects are produced without affecting the users of the objects. Mixin provide abstraction for classes, allowing classes to be built by combining parts. Mixins and structural types satisfy the same needs that led to multiple inheritance but in a more structured, less fragile, and more flexible way. At times there are tradeoffs between ease of programming and good design. OBSTACL supports good program design in part by making powerful abstractions accessible and easy to use, and at the same time avoiding or at least discouraging the use of features that lead to subtle errors during program maintenance.

[14] Sloppy interfaces (accidental subtyping) discourage refactoring improvements. For example, if points and vectors initially both use a generic $\{|x, y, z|\}$ interface, and lots of code for points is reused for manipulating vectors, then it is difficult to later change points without breaking the code using vectors. The programmer is left thinking, "I'd like to improve this code but can't because it will break unrelated code."

Chapter 7

Theory

I don't believe in mathematics.

–Albert Einstein

One of our goals with OBSTACL is to have a strong theoretical foundation. Formal semantics provide a precise description of the language, avoiding the ambiguities that are present when describing the system in prose. Since OBSTACL is an extension of ML, our semantics should be an extension of semantics for ML. However, a formal description of ML would overwhelm any extension describing objects. To focus on objects, classes, and mixins, we will extend a simple subset of ML. Most notably, we will leave out a treatment of parametric polymorphism and the ML module system. The "core calculus" presented here[1] includes the key constructs of OBSTACL—objects, method selection, data hiding with private fields, object types, subtyping, classes, run-time inheritance, method redefinition, constructors, public and protected visibility levels, mixins, and mixin constraints. We omit instantiators, named constructors, and named private fields. These features are not any more "interesting" than similar features in the core calculus (functions, unnamed constructors, unnamed private fields) and would only distract from the main features.

We discuss design motivations and tradeoffs and give a brief overview of the core calculus in section 7.1. We then present the syntax of the calculus (section 7.2), its operational semantics (section 7.3), and the type system (section 7.4). Finally, we compare our calculus with other object-oriented calculi and indicate directions for future research.

[1] An earlier version of this work appeared in *A Core Calculus of Objects and Mixins*, published in proceedings of ECOOP '99.

7.1 Design of the Core Calculus

In this section, we present our design motivations, discuss tradeoffs involved in designing calculi for object-oriented languages, give a short overview of our calculus, and present an example illustrating mixin usage.

7.1.1 Design Motivations

Our goal is to design a simple class-based calculus that correctly models the basic features of popular class-based languages and reflects modular programming techniques commonly employed by working programmers. Modular program development in a class-based language involves minimizing code dependencies such as those between a superclass and its subclasses and between a class implementation and object users. Our calculus minimizes dependencies by directly supporting data encapsulation, mixin inheritance, structural subtyping, and modular object creation.

We chose to model many of OBSTACL's features directly in the calculus. Our calculus directly supports classes, data hiding, and modular object construction. Mixin applications are reduced to "generator functions" which call all constructors in the inheritance chain in the correct order, producing a fully initialized object (see section 7.3).

7.1.2 Design Tradeoffs

In this section, we explain the design decisions and tradeoffs chosen in our calculus. Our goal was to sacrifice as little expressive power as possible while keeping the type system simple and free of complicated types such as polymorphic object types and recursive MYTYPE.

Classes. Although OBSTACL is a class based language, we could have chosen to represent classes as objects [AC96]. However, even in purely object-based calculi, the conflict between inheritance and subtyping usually requires that two sorts of objects be distinguished [FM95]. "Prototype objects" do not support full subtyping but can be extended with new methods and fields and/or have their methods redefined. "Proper objects" support both depth and width subtyping but are not extensible. Without this distinction, special types with extra information are required to avoid adding a method to an object in which a method with the same name is hidden as a consequence of subtyping (*e.g.*, labeled types of [BL95]). In our calculus, OBSTACL classes are modeled directly. The class construct in the calculus plays the role of a "prototype" (extensible but not subtypable) in other calculi, while objects—represented by records of methods—are subtypable but not extensible.

Objects. Records are an intuitive way to model objects since both are collections of name-value pairs (see section 5.1). The records-as-objects approach was developed in the pioneering

work on object-oriented calculi [CW85], in which inheritance was modeled by record subtyping. Unlike records, however, object methods should be able to modify fields and invoke sibling methods [Coo89]. To be capable of updating the object's internal state, methods must be functions of the host object (*self*). Therefore, objects must be *recursive* records. Moreover, *self* must be appropriately updated when a method is inherited, since new methods and fields may have been added and/or old ones redefined in the new host object. In our calculus, reduction rules produce class generators that are designed carefully so that methods are given a (recursive) reference to *self* only after inheritance has been resolved and all methods and fields contained in the host object are known.

Object updates. If all object updates are imperative, *self* can be bound to the host object when the object is instantiated from the class. We refer to this approach as *early* SELF binding. The name *self* then always refers to the same record, which is modified imperatively in place by the object's methods. The main advantage of early binding is that the fixed-point operator (which gives the object's methods reference to *self*) has to be applied only once, at the time of object instantiation.

If functional updates must be supported — which is, obviously, the case for purely functional object calculi — early binding does not work (see, for example, [AC96], where early binding is called *recursive semantics*). With functional updates, each change in the object's state creates a new object. If *self* in methods is bound just once, at the time of object instantiation, it will refer to the old, incorrect object and not to the new, updated one. Therefore, *self* has to be bound each time a method is invoked. We refer to this approach as *late self* binding.

Object extension. Object extension in an object-based calculus is typically modeled by an operation that extends objects by adding new methods to them. There are two constraints on such an operation: (i) the type system must prevent addition of a method to an object which already contains a method with the same name (unless we resolve these methods with scoped inheritance, as described in section 4.2.3), and (ii) since an object may be extended again after method addition, the actual host object may be larger than the object to which the method was added originally. The method body must behave correctly in any extension of the original host object. Therefore, it must have a polymorphic type with respect to *self*. The fulfillment of the two constraints can be achieved, for instance, via polymorphic types built on row schemes [Fis96] that use kinds to keep track of methods' presence.

Even more complicated is the case when object extension must be supported in a functional calculus. In the functional case, all methods modifying an object have the type of *self* as their return type. Whenever an object is extended or has its methods redefined (overridden), the type given to *self* in all inherited methods must be updated to take into account new and/or redefined methods. Therefore, the type system should include the notion of MYTYPE (or "self type") so that the inherited methods can be specialized properly. Support for MYTYPE generally leads to

more complicated type systems, in which forms of recursive types are required. Support can be provided by using row variables combined with recursive types [FHM94, Fis96, FM95], match-bound type variables [BSvG95, BB98], or by means of special forms of second-order quantifiers such as the `Self` quantifier of [AC96].

Tradeoffs. Our goals are to achieve a reasonable tradeoff between expressivity and simplicity, and to model the features of OBSTACL directly when possible. We do not support functional updates because we believe that imperative updates combined with early *self* binding keep the calculus simpler and closer to OBSTACL. Without functional updates, we can use early binding of *self*. Early binding eliminates the main need for recursive object types. There is also no need for polymorphic object types in our calculus since inheritance is modeled entirely at the class level and there are no object extension operations. This choice allows us to have a simple type system and a straightforward form of structural subtyping, in contrast to the calculi that support MYTYPE specialization [FM95, BSvG95].

There are at least two possible drawbacks to our approach. Although methods that *return* a modified *self* can be modeled in our calculus as imperative methods that modify the object and return nothing, methods that accept a MYTYPE *argument* cannot be simulated in our system without support for MYTYPE. We therefore have no support for binary methods of the form described in [BCC$^+$95], either in the calculus or in OBSTACL. Also, the type system of our calculus does not directly support *implementation types* (*i.e.*, types that include information about the class from which the object was instantiated and not just the object's interface). We believe that a form of implementation types can be provided by extending our type system with existential types (see section 9.3.3).

7.1.3 Design of the Core Calculus

The two new run time structures in OBSTACL are *objects* and *classes*. In our calculus, objects are records of methods. Methods are represented as functions with a binding for *self* (the host record) and *field* (the private field). Since records, functions, and λ-binding are standard, we need not introduce new operational semantics or type rules for objects. Instead, we introduce new constructs and rules for mixins and classes only. The new constructs are: *class values* (representing complete classes obtained as a result of mixin application), *mixin expressions* (containing definitions of methods, fields, and constructors), and *instantiation expressions* (representing creation of objects from classes). The calculus does not directly support class declarations. A class declaration in OBSTACL is represented as a mixin expression immediately followed by a mixin application.

A class value is a tuple containing a generator function, the set of all method names, and the set of method names that are protected. The generator produces a function from *self* to a record of methods. When the class is instantiated, the fixed-point operator is applied to the generator's result to bind *self* in the methods' bodies, creating a full-fledged object.

Mixins are represented by mixin expressions. Inheritance is modeled by the evaluation rule that applies a mixin to a class value representing the superclass, producing a new class value. The generator of the new class takes the record of superclass methods built by the superclass generator and modifies it by adding and/or replacing methods as specified by the mixin. Only class values can be instantiated; mixins are used solely for building class hierarchies.

Like OBSTACL, the core calculus supports only private fields and public and protected methods. Private methods can be modeled by private fields with a function type; public or protected fields can be modeled by combining private fields with accessor methods (see section 9.1.2 and section 9.1.3). Instead of putting encapsulation levels into object types, we express them using subtyping and binding. Protected methods are treated in the same way as public methods except that they are excluded from the type of the object returned to the user. Private fields are not listed in the object type at all but are instead bound in each method body. In the core calculus each class has exactly one private field.[2] Each method body takes the class's private field as a parameter.

7.1.4 An Example of Mixin Inheritance

Mixin inheritance can be a powerful tool for constructing class hierarchies. In this section, we give a simple example that demonstrates how a mixin can be implemented in our calculus and explain some of the uses of mixins. For readability, the example uses functions with multiple arguments even though they are not formalized explicitly in the calculus.

Mixin definition. Figure 7.1 shows the definition of an Encrypted mixin that implements encryption functionality on top of any stream class. Note that the class to which the mixin is applied may have more methods than expected by the mixin. For example, Encrypted can be applied to Socket applied to *Object* where *Object* is the root of all class hierarchies, even though Socket applied to *Object* has other methods besides *read* and *write*.

Mixin expressions contain new methods (marked by the method keyword), redefined methods (redefine keyword), and a constructor.[3] The names of protected methods should be listed following the protect keyword. Instead of introducing a special field construct, every mixin contains a single private field which is λ-bound in each method body (λ key. $\cdots$).

Methods can access the host object through the *self* parameter, which is λ-bound in each method body to avoid introducing special keywords. The λ self. is implicit in OBSTACL, but explicit in the calculus to simplify the treatment. Redefined methods can access the old method body inherited from the superclass via the *next* parameter. Constructors are simply functions returning a record of two components. The *fieldinit* value is used to initialize the private field. The *superinit* value is passed as an argument to the superclass constructor.

[2] To model OBSTACL's multiple fields per class in the calculus, we simply set the field to be a record type.

[3] OBSTACL supports multiple named constructors, while the calculus has only one unnamed constructor. Another difference is that OBSTACL supports instantiators, but the calculus allows direct access to new. OBSTACL instantiators can be modeled in the calculus as ordinary functions.

```
let File =
    mixin
      method write = ...
      method read = ...
      ...
    end in

let Socket =
    mixin
      method write = ...
      method read = ...
      method hostname = ...
      method portnumber = ...
      ...
    end in

let Encrypted =
    mixin
      redefine write = λ next. λ key. λ self. λ data. next (encrypt(data, key));
      redefine read =  λ next. λ key. λ self. λ _ .    decrypt(next (), key);
      constructor λ (key, arg). {fieldinit=key, superinit=arg};
      protect [];
    end in ...
```

Figure 7.1: A mixin and two classes in the calculus for OBSTACL

From the definition of Encrypted, the type system infers the constraint that must be satisfied by any class to which Encrypted is applied. The class must contain *write* and *read* methods whose types must be supertypes of those given to *write* and *read*, respectively, in the definition of Encrypted.

7.2 Syntax of the Core Calculus

The syntax of our calculus is fundamentally class-based. There are four expressions involving classes: classval, mixin, $\diamond$ (mixin application[4]), and new. Class-related expressions and values are treated as any other expression or value in the calculus. They can be passed as arguments, put into data structures, etc. However, class values and object values are not intended to be written directly; instead, these expression forms are used only to define the semantics of programs. Class values are created by mixin application,[5] and object values are created by class instantiation.

Let *Var* be an enumerable set of variables (otherwise referred to as *identifiers*), and *Const* be a set of constants. Expressions E and values V (with $V \subset E$) of the core calculus are as in figure 7.2, where $const \in Const$; $x, x_i, m_i, m_j \in Var$; *fix* is the fixed-point operator; ref, !, := are operators;[6] $\{x_i = e_i\}^{i \in I}$ is a record; $e.x$ is the record selection operation; h is a set of

[4]The calculus uses a different syntax than OBSTACL to distinguish function call from mixin application without examining the types of the expressions.

[5]Class definitions in OBSTACL can be represented here as mixins applied to *Object*. Regular class inheritance in OBSTACL is a mixin definition immediately followed by mixin application.

[6]Introducing ref, !, := as operators rather than standard forms such as ref$e$, !e, :=$e_1 e_2$, simplifies the definition of evaluation contexts and proofs of properties. As noted in [WF94], this is just a syntactic convenience, as is the curried

$$
\begin{array}{lll}
\text{Expressions:} & e ::= & \mathit{const} \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid :=\\
& \mid & \{x_i = e_i\}^{i\in I} \mid e.x \mid \mathsf{H}\ h.e \mid \mathsf{new}\ e\\
& \mid & \mathsf{classval}\langle v_g, [m_i]^{i\in Meth}, [p_\ell]^{\ell\in Prot}\rangle\\
& \mid & \quad \mathsf{mixin}\\
& & \quad\ \mathsf{method}\ m_j = v_{m_j};\ ^{(j\in New)}\\
& & \quad\ \mathsf{redefine}\ m_k = v_{m_k};\ ^{(k\in Redef)}\\
& & \quad\ \mathsf{protect}\ [p_\ell];\ ^{(\ell\in Prot)}\\
& & \quad\ \mathsf{constructor}\ v_c;\\
& & \quad \mathsf{end}\\
& \mid & e_1\ \diamond\ e_2
\end{array}
$$

$$
\begin{array}{lll}
\text{Values:} & v ::= & \mathit{const} \mid x \mid \lambda x.e \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid :=\ \mid :=v \mid \{x_i = v_i\}^{i\in I}\\
& \mid & \mathsf{classval}\langle v_g, [m_i]^{i\in Meth}, [p_\ell]^{\ell\in Prot}\rangle\\
& \mid & \quad \mathsf{mixin}\\
& & \quad\ \mathsf{method}\ m_j = v_{m_j};\ ^{(j\in New)}\\
& & \quad\ \mathsf{redefine}\ m_k = v_{m_k};\ ^{(k\in Redef)}\\
& & \quad\ \mathsf{protect}\ [p_\ell];\ ^{(\ell\in Prot)}\\
& & \quad\ \mathsf{constructor}\ v_c;\\
& & \quad \mathsf{end}
\end{array}
$$

Figure 7.2: Syntax of the core calculus

pairs $h ::= \{\langle x, v\rangle^*\}$ where $x \in \mathit{Var}$ and v is a value (first components of the pairs are all distinct); $\mathsf{H}\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$ associates reference variables $x_1, \ldots, x_n$ with values $v_1, \ldots, v_n$; $[m_i], [p_\ell]$ are sets of identifiers; and $I, J, K, L, \mathit{Meth}, \mathit{Prot}, \mathit{New}, \mathit{Redef} \subset \mathbb{N}$. The expression H binds $x_1, \ldots, x_n$ in $v_1, \ldots, v_n$ and in e; The set of pairs h in the expression $\mathsf{H}h.e$ represents the *heap*, where the results of evaluating imperative subexpressions of e are stored.

Our calculus takes a standard calculus of functions, records, and imperative features and adds new constructs to support classes and mixins. We chose to extend *Reference ML* [WF94], in which Wright and Felleisen analyze the operational soundness of a version of ML with imperative features.[7] Our calculus does not include let expressions as primitives since we do not need polymorphism to model our objects. We do rely on the Wright-Felleisen idea of *store*, which we call a *heap*, in order to evaluate imperative side effects.

The intuitive meaning of the class-related expressions is as follows:

- $\mathsf{classval}\langle v_g, [m_i]^{i\in Meth}, [p_\ell]^{\ell\in Prot}\rangle$ is a *class value*, *i.e.*, the result of mixin application. It is a triple, containing one function and two sets of variables. The function v_g is the generator for the class. The $[m_i]$ set contains the names of all methods defined in the class, and the $[p_\ell]$ set contains the names of protected methods.

version of $:=$.

[7] This choice mirrors our choice to extend ML to define OBSTACL. It is not essential to express the constructs we wish to analyze, but it is clean and convenient.

- mixin
  method $m_j = v_{m_j}$; $^{(j \in New)}$
  redefine $m_k = v_{m_k}$; $^{(k \in Redef)}$
  protect $[p_\ell]$; $^{(\ell \in Prot)}$
  constructor v_c;
  end

  is a *mixin*, in which $m_j = v_{m_j}$ are definitions of new methods, and $m_k = v_{m_k}$ are method redefinitions that will replace methods with the same name in any class to which the mixin is applied. Each method body $v_{m_{j,k}}$ is a function of *self*, which will be bound to the newly created object at instantiation time, and of the private *field*. In method redefinitions, v_{m_k} is also a function of *next*, which will be bound to the old, redefined method from the superclass. The v_c value in the constructor clause is a function that returns a record of two components. When evaluating a mixin application, v_c is used to build the generator as described in section 7.3.

- $e_1 \diamond e_2$ is an application of mixin e_1 to class value e_2. It produces a new class value. Mixin application is the basic inheritance mechanism in our calculus.

- new e uses generator v_g of the class value to which e evaluates to create a function that returns a new object, as described in section 7.3.

Programs and *answers* are defined as follows:

$$\begin{aligned} p &::= \quad e \qquad \text{where } e \text{ is a closed expression} \\ a &::= \quad v \mid \mathsf{H}\ h.v \end{aligned}$$

Finally, we define the root of the class hierarchy, class *Object*, as a predefined class value:

$$\textit{Object} \triangleq \mathsf{classval}\langle\ \lambda\_.\lambda\_.\{\},\ [\ ],\ [\ ]\ \rangle$$

The root class is necessary so that all other classes can be treated uniformly. Intuitively, *Object* is the class whose object instances are empty objects. It is the only class value that is not obtained as a result of mixin application. The calculus can then be simplified by assuming that any user-defined class that does not need a superclass is obtained by applying a mixin containing all of the class's method definitions to *Object*.

Throughout this chapter, we will use let $x = e_1$ in e_2 in terms and examples as a more readable equivalent of $(\lambda x.e_2)e_1$. Also, we use unit as an abbreviation for the empty record or type $\{\}$, instead of having a new *unit* value and type[8] We will use the word "object" when the record in question represents an object. To avoid name capture, we apply α-conversion to binders λ and H.

[8]The unit type is used in a similar way to C's `void` construct. A function `int f(void)` in C is written `f:unit->int` in ML. A function `void f(int)` in C is written `f:int->unit` in ML. Although unit can occur in more contexts than `void`, it plays the same role: a type name indicating the lack of a value.

$$
\begin{array}{rcll}
\mathit{const}\ v & \rightarrow & \delta(\mathit{const}, v) \ \text{ if } \delta(\mathit{const}, v) \text{ is defined} & (\delta) \\
(\lambda x.e)\ v & \rightarrow & [v/x]\ e & (\beta_v) \\
\mathit{fix}\ (\lambda x.e) & \rightarrow & [\mathit{fix}(\lambda x.e)/x]e & \mathit{(fix)} \\
\{\ldots, x = v, \ldots\}.x & \rightarrow & v & \text{(select)} \\
\mathsf{ref}\, v & \rightarrow & \mathsf{H}\langle x, v\rangle.x & \mathit{(ref)} \\
\mathsf{H}\langle x, v\rangle h.R[!x] & \rightarrow & \mathsf{H}\langle x, v\rangle h.R[v] & \mathit{(deref)} \\
\mathsf{H}\langle x, v\rangle h.R[:=x v'] & \rightarrow & \mathsf{H}\langle x, v'\rangle h.R[v'] & \mathit{(assign)} \\
R[\mathsf{H}\ h.e] & \rightarrow & \mathsf{H}\ h.R[e], \quad R \neq [\,] & \mathit{(lift)} \\
\mathsf{H}\ h.\mathsf{H}\ h'.e & \rightarrow & \mathsf{H}\ h\ h'.e & \mathit{(merge)} \\
\mathsf{new\ classval}\langle g, \mathcal{M}, \mathcal{P}\rangle & \rightarrow & \lambda v.\mathit{Sub}_{\mathcal{M}\rightarrow\mathcal{M}\setminus\mathcal{P}}(\mathit{fix}(g\ v)) & \mathit{(new)}
\end{array}
$$

$$
\left(\begin{array}{l}
\mathsf{mixin} \\
\ \mathsf{method}\ m_j = v_{m_j}; \\
\ \mathsf{redefine}\ m_k = v_{m_k}; \\
\ \mathsf{protect}\ [p_\ell]; \\
\ \mathsf{constructor}\ c; \\
\mathsf{end}
\end{array}\right)^{\substack{j \in New, \\ k \in Redef, \\ \ell \in Prot}} \diamond\ \mathsf{classval}\langle g, \mathcal{M}, \mathcal{P}\rangle \rightarrow \mathsf{classval}\langle \mathit{Gen}, [m_j] \cup \mathcal{M}, [p_\ell] \cup \mathcal{P}\rangle \qquad \mathit{(mixin)}
$$

if $[m_j] \cap \mathcal{M} = \emptyset$, $[m_k] \subset \mathcal{M}$, and $[p_\ell] \subset [m_j] \cup \mathcal{M}$; Gen is defined below

Figure 7.3: Reduction rules

7.3 Operational Semantics

The operational semantics for our calculus extends that of *Reference ML* [WF94]. Reduction rules are given in figure 7.3, where R are *reduction contexts* [CF91, FH92, MT89]. Expression Gen is defined below. Relation $\twoheadrightarrow$ is the reflexive, transitive, contextual closure of $\rightarrow$, with respect to *contexts* C, as defined (in a standard way) in appendix A.1.

Reduction contexts are necessary to provide a minimal relative linear order among the creation, dereferencing and updating of heap locations, since side effects need to be evaluated in a deterministic order. Reduction contexts R are defined as follows:

$$
\begin{array}{rcl}
R ::= & & [\ ] \mid R\ e \mid v\ R \mid R.x \mid \mathsf{new}\ R \mid R \diamond e \mid v \diamond R \\
& \mid & \{m_1 = v_1, \ldots, m_{i-1} = v_{i-1}, m_i = R, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \leq i \leq n}
\end{array}
$$

To abstract from a precise set of constants, we only assume the existence of a partial function $\delta : \mathit{Const} \times \mathit{ClosedVal} \rightharpoonup \mathit{ClosedVal}$ that interprets the application of functional constants to closed values and yields closed values. See section 7.4 for the δ-typability condition.

Rules (β_v), (*fix*), and (*select*) are standard.

Rules (*ref*), (*deref*), and (*assign*) evaluate imperative expressions following the linear order given by the reduction context R and acting on the heap. They are formulated after [WF94]: (*ref*) generates a new heap location where the value v is stored, (*deref*) retrieves the contents of the location x, (*assign*) changes the value stored in a heap location.

Rules (*lift*) and (*merge*) combine inner local heaps with outer ones whenever a dereference operator or an assignment operator cannot find the needed location in the closest local heap.

Rule (*new*) builds a function that can create a new object. The resulting function can be thought of as the composition of three functions: $Sub \circ \mathit{fix} \circ g$. Given an argument v, it will apply generator g to v, creating a function from *self* to a record of methods. Then the fixed-point operator *fix* (following [Coo89]) is applied to bind *self* in method bodies and create a recursive record. Finally, we apply $Sub_{\mathcal{M}\to\mathcal{M}\setminus\mathcal{P}}$, a coercion function from records to records that hides all components belonging to the protected set $\mathcal{P}$. The resulting record contains only public methods and can be returned to the user as a fully formed object.

Rule (*mixin*) evaluates mixin application expressions. A mixin is applied to a superclass value $\mathsf{classval}\langle g, \mathcal{M}, \mathcal{P}\rangle$. $\mathcal{M}$ is a set of all method names defined in the superclass; $\mathcal{P}$ is an annotation listing the names of protected methods in the superclass. The resulting class value is $\mathsf{classval}\langle Gen, [m_j] \cup \mathcal{M}, [p_\ell] \cup \mathcal{P}\rangle$ where Gen is the generator function defined below, $[m_j] \cup \mathcal{M}$ is the set of all method names, and $[p_\ell] \cup \mathcal{P}$ is an annotation listing protected method names. Using generators delays full inheritance resolution until object instantiation time when *self* becomes available.

Gen is the class generator. It takes a single argument x which is used by the constructor subexpression c to compute the initial value for the private field of the new object and the argument for the superclass generator. Gen returns a function from *self* to a record of methods. When the fixed-point operator is applied to the function returned by the generator, it produces a recursive record of methods representing a new object (see the (*new*) rule).

$$
\begin{array}{l}
Gen \stackrel{\triangle}{=} \lambda x. \\
\qquad \mathsf{let}\ t = c(x) \quad \mathsf{in} \\
\qquad \mathsf{let}\ \textit{supergen} = g(t.\mathrm{superinit}) \quad \mathsf{in} \\
\qquad \lambda \textit{self}. \\
\qquad\qquad \left\{ \begin{array}{llll}
m_j = \lambda y. v_{m_j} & & t.\mathrm{fieldinit}\ \textit{self} & y \\
m_k = \lambda y. v_{m_k} & (\textit{supergen self}).m_k & t.\mathrm{fieldinit}\ \textit{self} & y \\
m_i = \lambda y. & (\textit{supergen self}).m_i & & y
\end{array} \right\}^{m_i \in \mathcal{M} \setminus [m_k]}
\end{array}
$$

In the mixin expression, the constructor subexpression c is a function of one argument which returns a record of two components: one is the initialization expression for the field (*fieldinit*), and the other is the superclass generator's argument (*superinit*). Gen first calls $c(x)$ to compute the initial value of the field and the value to be passed to the superclass generator g. Gen then calls the superclass generator g, passing argument $t.\mathrm{superinit}$, to obtain a function *supergen* from *self* to a record of superclass methods.

Finally, Gen builds a function from *self* that returns a record containing *all* methods — from both the mixin and the superclass. To understand how the record is created, recall that

method bodies take parameters *field*, *self*, and, if a redefinition, *next*. Methods m_j are the *new* mixin methods: they appear for the first time in the current mixin expression. Gen must bind *field* and *self* for them. Methods $m_i \in \mathcal{M} \setminus [m_k]$ are the *inherited* superclass methods: they are taken intact from the superclass's object (*supergen self*). Methods m_k are *redefined* in the mixin. Their bodies can refer to the old methods through the *next* parameter, which is bound to (*supergen self*)$.m_i$ by Gen. They also receive a binding for *field* and *self*. For all three sorts of methods, the method bodies are wrapped inside $\lambda y.\cdots y$ to delay evaluation in our call-by-value calculus.

7.4 Type System

Our types are standard and the typing rules are fairly straightforward. The complexity of typing object-oriented programs in our system is limited exclusively to classes and mixins. Method selection, which is the only operation on objects in our calculus, is typed as ordinary record component selection. Since methods are typed as ordinary functions, method invocation is simply a function application.

Types are as follows:

$$\begin{array}{rcl} \tau ::= & \iota \mid \tau_1 \to \tau_2 \mid \tau \ \mathsf{ref} \mid \{x_i : \tau_i\}^{i \in I} \\ \mid & \mathsf{class}\langle \tau, \{m_i{:}\tau_i\}, [p_\ell] \rangle^{i \in I, \ell \in L} \\ \mid & \mathsf{mixin}\langle \tau_1, \tau_2, \{m_j{:}\tau_j\}, \{m_k{:}\tau_k\}, [p_\ell] \rangle^{j \in J, k \in K, \ell \in L} \end{array}$$

where ι is a constant type; $\to$ is the functional type operator; τ ref is the type of locations containing a value of type τ; $\{x_i{:}\tau_i\}^{i \in I}$ is a record type; and $I, J, K, L \subset \mathbb{N}$. In class types, $\{m_i{:}\tau_i\}$ is a record type and $[p_\ell]$ is a set of names, where $[p_\ell] \subseteq [m_i]$. In mixin types, $\{m_j : \tau_j\}, \{m_k : \tau_k\}$ are record types and $[p_\ell]$ is a set of names, where $[p_\ell] \subseteq ([m_j] \cup [m_k])$. Although record expressions and values are ordered so that we can fix an order of evaluation, record types are unordered. We also assume we have a function *typeof* from constant terms to types that respects the following *typability condition* [WF94]: for $const \in Const$ and value v, if *typeof*$(const) = \tau' \to \tau$ and $\emptyset \vdash v : \tau'$, then $\delta(const, v)$ is defined and $\emptyset \vdash \delta(const, v) : \tau$.

Our type system supports structural subtyping (the $<:$ relation) along with the subsumption rule (*sub*). The subtyping rules were informally introduced in section 5.1.3 and are shown in full in appendix A.2. Since subtyping on references is unsound and we wish to keep subtyping and inheritance completely separate, we have only the basic subtyping rules for function and record types. Subtyping only exists at the object level and is not supported for class or mixin types.

$$\frac{\Gamma \vdash g : \gamma \to \{m_i : \tau_i\}^{i \in All} \to \{m_i : \tau_i\}^{i \in All}}{\Gamma \vdash \mathsf{classval}\langle g, [m_i]^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle} \quad (\textit{class val})$$

$$\frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \to \{m_j{:}\tau_j\}^{j \in All \setminus Prot}} \quad (\textit{instantiate})$$

$$\frac{\begin{array}{rll} \text{(New)} & \textbf{For } j \in New\textbf{:} & \Gamma \vdash v_{m_j} : \quad \eta \to \sigma \to \tau^{\downarrow}_{m_j} \\ \text{(Redef)} & \textbf{For } k \in Redef\textbf{:} & \Gamma \vdash v_{m_k} : \tau^{\uparrow}_{m_k} \to \eta \to \sigma \to \tau^{\downarrow}_{m_k} \\ \text{(Constr)} & & \Gamma \vdash c : \gamma_d \to \{\mathrm{fieldinit} : \eta, \mathrm{superinit} : \gamma_b\} \end{array}}{\Gamma \vdash \left(\begin{array}{l} \mathsf{mixin} \\ \ \mathsf{method}\ m_j = v_{m_j}; \\ \ \mathsf{redefine}\ m_k = v_{m_k}; \\ \ \mathsf{protect}\ [p_\ell]; \\ \ \mathsf{constructor}\ c; \\ \mathsf{end} \end{array}\right) : \begin{array}{l} {\scriptstyle j \in New} \\ {\scriptstyle k \in Redef} \\ {\scriptstyle \ell \in Prot} \end{array} \mathsf{mixin}\langle \gamma_b, \gamma_d, \{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}, \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\}, [p_\ell]\rangle} \quad (\textit{mixin})$$

$$\text{where} \quad \begin{array}{l} \sigma = \{m_i : \tau^{\uparrow}_{m_i}, m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\} \\ [p_\ell] \ \subseteq \ [m_i] \cup [m_j] \cup [m_k] \\ m_i, \tau^{\uparrow}_{m_i}, \tau^{\uparrow}_{m_k} \text{ are inferred from method bodies} \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \sigma_{\mathrm{Old}}, \sigma_{\mathrm{New}}, P_d\rangle \\ \Gamma \vdash e_2 : \mathsf{class}\langle \gamma_c, \sigma_b, P_b\rangle \\ \Gamma \vdash \sigma_d <: \sigma_b <: \sigma_{\mathrm{Old}} \\ \Gamma \vdash \gamma_b <: \gamma_c \end{array}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{class}\langle \gamma_d, \sigma_d, P_b \cup P_d\rangle} \quad (\textit{mixin app})$$

$$\text{where} \quad \begin{array}{lcl} \sigma_b & = & \{m_k : \tau_{m_k}, m_l : \tau_{m_l}, m_i : \tau_{m_i}\} \\ \sigma_{\mathrm{Old}} & = & \{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\} \\ \sigma_{\mathrm{New}} & = & \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\} \\ \sigma_d & = & \{m_i : \tau_{m_i}, m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}, m_l : \tau_{m_l}\} \end{array}$$

Figure 7.4: Typing rules for class-related forms

Typing environments are defined as follows:

$$\Gamma ::= \varepsilon \mid \Gamma, x:\tau \mid \Gamma, \iota_1 <: \iota_2$$

where $x \in Var$, τ is a well-formed type, ι_1, ι_2 are constant types, and $x, \iota_1 \notin dom(\Gamma)$.

Typing judgments are as follows:

$\Gamma \vdash \tau_1 <: \tau_2$	τ_1 is a subtype of τ_2
$\Gamma \vdash e : \tau$	e has type τ

The set of typing rules for class-related forms is shown in figure 7.4. The remaining rules are standard and can be found in appendix A.2.

Rule (*class val*) types class values. A class value is composed of an expression and two sets of method names. The expression g is the generator (see section 7.3) which produces a function that will later, at the time of new application, return a real object. The type of g can be determined by examining the type of the class value, $\mathsf{class}\langle\gamma, \{m_i{:}\tau_i\}, [p_\ell]\rangle$. Generator g takes an argument of type γ and returns a function that will return an object once the fixed-point operator is applied. The return type of g is therefore $\sigma \to \sigma$, where σ represents the type of *self*, $\{m_i : \tau_i\}$. This record type includes *all* methods, not only public methods. When the fixed-point operator is applied, $\mathit{fix}(g\ v)$ will have type σ when v has type γ.

Rule (*mixin*) types mixin declarations. We describe it following the order of its premises. Note that mixin methods make typing assumptions about methods of the superclass to which the mixin will be applied. We refer to these types as *expected* types since the actual superclass methods may have different types. The exact relationship between the types expected by the mixin and the actual types of the superclass methods is formalized in rule (*mixin app*). We mark types that come from the *superclass* with $^\uparrow$ and those that will be changed or added in the *subclass* with $^\downarrow$.

- (New) The bodies of the new methods v_{m_j} are typed with a function type. The argument types are the type of the private field (η) and the type of *self* (σ). We do not lose generality by assuming only one field per class since η can be a tuple or record type. The return type is $\tau^\downarrow_{m_j}$.
- (Redef) The bodies of the redefined methods v_{m_k} are also typed with a function type. The first argument type $\tau^\uparrow_{m_k}$ is that of *next*, *i.e.*, the superclass method with the same name (recall that the new body can refer to the old body via *next*). The meaning of η and σ is the same as for the new methods. It is not known at the time of mixin definition to which class the mixin will be applied, so the actual type of the method replaced by m_k may be different from the expected type $\tau^\uparrow_{m_k}$.

- (Constr) The constructor expression c is a function that takes an argument of type γ_d and returns a record with two components. The component labelled *fieldinit* is the initialization expression for the private field. It has to have the same type η as that assumed for the field when typing method bodies. The component labeled *superinit* is the expression passed as the parameter to the superclass generator. Its type γ_b is inferred from the constructor definition since the superclass is not available at the time of mixin definition.

Both new and redefined methods in the mixin may call superclass methods (*i.e.*, methods that are expected to be supported by any class to which the mixin is applied). We refer to these methods as m_i. Their types $\tau^{\uparrow}_{m_i}$ are inferred from the mixin definition.

The mixin is typed with a $\mathsf{mixin}\langle\ldots\rangle$ type, which encodes the following information about the mixin:

- γ_b is the expected argument type of the superclass generator.
- γ_d is the exact argument type of the mixin generator.
- $\{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}$ are the expected types of the methods that must be supported by any class to which the mixin is applied. Recall that m_i are the methods that are not redefined by the mixin but still expected to be supported by the superclass since they are called by other mixin methods, and $\tau^{\uparrow}_{m_k}$ are the types assumed for the old bodies of the methods redefined in the mixin.
- $\{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\}$ are the exact types of mixin methods (new and redefined, respectively).
- $[p_\ell]$ is an annotation listing the names of all methods to be protected, both new and redefined.

Type information contained in the $\mathsf{mixin}\langle\ldots\rangle$ type is used when typing mixin application in rule (*mixin app*).

Rule (*mixin app*) types mixin-based inheritance. In the rule definition, σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied. In particular, m_k are the superclass methods redefined by the mixin, m_i are the superclass methods called by the mixin methods but not redefined, and m_l are the superclass methods not mentioned in the mixin definition at all. Note that the superclass may have more methods than required by the mixin constraint.

Type σ_d contains the signatures of all methods supported by the subclass created as a result of mixin application. Methods $m_{i,l}$ are inherited directly from the superclass, methods m_k are redefined by the mixin, and methods m_j are the new methods added by the mixin. We are guaranteed that methods m_j are not present in the superclass by the construction of σ_b and σ_d: σ_d is defined so that it contains all the labels of σ_b plus labels m_j. Type σ_{Old} lists the (expected)

types of the superclass methods assumed when typing the mixin definition. Type σ_{New} lists the exact types of the methods newly defined or redefined in the mixin.

The premises of the rule are as follows:

- $\mathsf{mixin}\langle\ldots\rangle$ and $\mathsf{class}\langle\ldots\rangle$ are the types of the mixin and the superclass, respectively.
- The $\sigma_d <: \sigma_b$ constraint requires that the types of the methods redefined by the mixin (m_k) be subtypes of the superclass methods with the same name. This ensures that all calls to the redefined methods in m_i and m_l (methods inherited intact from the superclass) are type-safe.
- The $\sigma_b <: \sigma_{\mathrm{Old}}$ constraint requires that the actual types of the superclass methods m_i and m_k be subtypes of the expected types assumed when typing the mixin definition.
- The $\gamma_b <: \gamma_c$ constraint requires that the actual argument type of the superclass generator be a supertype of the type assumed when typing the mixin definition.

In the type of the class value created as a result of mixin application, σ_b is the argument type of the generator, and σ_d (see above) is the type of objects that will be instantiated from the class (except for the protected methods which are included in σ_d but hidden in the instantiated objects). In the resulting subclass we protect all methods that are protected either in the superclass or in the mixin.

Rule (*mixin app*) also determines how name clashes between the mixin and the superclass are handled. Suppose the superclass and the mixin contain a method with the same name m. If m is a redefined method in the mixin (*i.e.*, $m \in [m_k]$), then it will replace the method from the superclass as long as its type $\tau^{\downarrow}_{m_k}$ is a subtype of the replaced method's type τ_{m_k}. This requirement is checked by the $\sigma_d <: \sigma_b$ premise. If m is a new method (*i.e.*, $m \in [m_j]$), then the rule's premises will fail since a method that is considered new by the mixin appears in the superclass ($m = m_j \in \sigma_b$), and the type system will signal an error.

Rule (*instantiate*) types the creation of a new object. The $\mathsf{new}\ e$ term is typed as a function that takes the generator's argument and returns a fully initialized object. The object's type contains only the public methods; the protected methods are hidden.

The proof of soundness can be found in [BPSM99].

7.5 Related Work

In the literature, there exists an extensive body of work on calculi for object-oriented languages. The calculus for OBSTACL presented in this chapter can be compared directly with the following class-oriented calculi:

- In the simplest of Cook's calculi [Coo89], objects are represented by records of methods and created by taking the fixed-point of the function representing the class (*constructor* in Cook's terminology). Inheritance is modeled by generating the subclass constructor from the superclass constructor, and *self* is bound early. However, classes are not a basic construct. The calculus relies on record concatenation operators, but typing issues associated with them are not addressed.

- The closure semantics version of the "dynamic inheritance" language analyzed by Kamin and Reddy [KR94] is similar to our calculus. The language is class-based, and the semantics of inheritance is similar to our generators. They also compare late and early *self* binding (*fixed-point model* and *self-application model* in their terminology). However, no type system is provided and there is no discussion of object construction or method encapsulation.

- The calculus of Wand [Wan94] is class-based and strongly typed. Classes are modeled as extensible records, inheritance is record concatenation plus *self* update so that inherited methods refer to the correct object. As in our calculus, objects are records, *self* is bound early, and the *new* operation (called *constructor*) is an application of the fixed-point operator. There is also no support for parameterized inheritance with flexible constraints.

OBSTACL includes classes and imperative updates. By modeling these directly in the calculus, we can omit MYTYPE, simplifying the calculus and the type system for the language.

Other approaches to modeling classes can be found in object-based calculi, where classes are not first-class expressions and have to be constructed from more primitive building blocks:

- Abadi and Cardelli use a record of pre-methods plus a constructor function to model classes [AC96]. Pre-methods are functions from *self* to method bodies or functions that are written as methods but not yet installed in any object. The difference between the result of Gen (see section 7.3 above) and a record of pre-methods is that the former is a function from *self* to a record of methods while the latter is a record of functions from *self* to methods. However, their approach provides no language support for classes, instead accepting only some subset of objects (which satisfy a complicated set of constraints) to be used in place of classes [FM98].

- Fisher uses extensible objects to model classes [Fis96]. Extensible objects support method addition but not subtyping, while "proper" object support subtyping but not extension [FM95]. Extensible objects, or "prototypes", roughly correspond to classes in OBSTACL, and the operation to turn prototypes into proper objects roughly corresponds to class instantiation in OBSTACL. The main difference is that methods may be called on prototypes, while methods in classes may not be called until they are placed into a (proper) object.

While in the prototype state, methods must have a partially abstract view of its host object. While the system of [BF98] can model a form of mixins, our calculus is simpler, more intuitive, and has encapsulation and object creation semantics closer to those used in OBSTACL.

To the best of our knowledge, there are not many formal settings in which mixin-based inheritance is analyzed:

- Flatt et al. implement mixins in the MZSCHEME language [FF98] and formalize an extension of a subset of JAVA with mixins in [FKF98]. Their system supports higher-order mixin composition, a hierarchy of named interface types, and resolution of accidental name collisions. The collision resolution system allows old and new method definitions to coexist. The two are distinguished using the "view" of an object, which is carried with the object at run-time and altered at each subsumption step (see section 4.2.3). As a result, method lookup is sensitive to the object's history of subsumptions. Our mixins are created and manipulated as run-time values as opposed to static top-level declarations. Mixin constraints prevent objects from having incompatible methods with the same name, so method lookup is straightforward and does not depend on the object's subsumption history. Proper object initialization is guaranteed.

- BETA [MMPN93] replaces classes, procedures, functions, and types by a single abstraction mechanism called the *pattern*. Objects are created from the patterns, and in addition to traditional objects as found in conventional object-oriented languages, objects in BETA may also represent function activations, exception occurrences, or concurrent processes. Patterns may be used as *superpatterns* to other patterns in a manner similar to conventional inheritance. Since patterns are a general concept, inheritance is available also for procedures, functions, exceptions, coroutines, and processes. *Virtual* patterns are similar to generic templates or parameterized classes with the additional benefit that the parameter may be restricted without actually instantiating the template (this is similar to computing the mixin constraint without actually applying the mixin to a class). Mixin inheritance is a partial case of the very general pattern inheritance mechanism developed in BETA.

- OCAML [LRVD99] supports a very limited form of parameterized inheritance by combining a module abstraction mechanism with classes that can inherit across module boundaries. Because the exact module containing the superclass may not be known when the subclass is defined, the same subclass can be used with multiple superclass definitions. However, methods not mentioned in the superclass type become inaccessible. In the example of section 7.1.4, this would mean that all methods that are present in the Socket $\diamond$ *Object* class besides *read* and *write* are forgotten once Encrypted mixin is applied to it.

Chapter 8

Implementation

A language may look good on paper, but without a reasonably clean and efficient implementation it cannot serve the needs of programmers. At this time there are no complete implementations of OBSTACL, but we have implemented prototypes that indicate that an efficient implementation would be feasible, except for type inference, which is known to be difficult in the presence of subtyping. In this chapter we describe a proposed implementation, parts of which have been implemented in two prototypes.

Since OBSTACL is an extension of ML, we will not discuss here the implementation of ML features used in OBSTACL, such as records, functions, control constructs, and arrays, except where their implementation affects the representation of objects or classes. There are several good implementations of ML [LRVD99, AM91] and there has been work on highly efficient implementations [TMC+96]. Instead we focus on representation (objects, classes, and mixins) and operations (selection, comparison, inheritance, and instantiation) of our extensions to ML. Most ML implementations, like implementations of other high level languages, put all structured data on the heap by default and use the stack only for primitive types, so we put objects, classes, and mixins on the heap and access them through pointers.

8.1 Implementation Strategy

In the proposed implementation described here, separate compilation is a key restriction on the kinds of techniques we allow. An important goal is to require recompiling a module only when that module is changed in some programmer-visible way. Recompilation should not be caused by some artifact of the implementation.[1] In particular, private fields should be able

[1]Recompilation is not an issue that is independent of the language design. Note that excessive recompilation is a common complaint of C++ users, but not a complaint of Smalltalk users. This is because the "header file" paradigm in C++ encourages programmers to include implementation details in files that should contain only interfaces. Most C++ systems trigger recompilation on change to the file. Thus a change to the implementation *or even the comments*

to change without subclasses needing recompilation. Separate compilation rules out whole-program analysis, which enables cross-module optimizations. On the other hand, it allows for dynamically linked (shared) libraries. For example, as long as interfaces are compatible, a superclass in a dynamically linked library can be upgraded (possibly changing the layout of the objects) without breaking subclasses defined in the main application. Separate compilation of mixins is related to dynamically linked libraries in that when the mixin is compiled, the definition of the superclass is not available. The implementation is necessarily prohibited from relying on the layout of private data in the superclass. Such a challenge is not normally an issue in most statically typed object-oriented languages, where subclasses are typically compiled after the superclass definition is available. In dynamically typed languages, knowledge of layout is often deferred until run time, at the cost of performance. The proposal described here preserves the efficiency of static languages while maintaining separate compilation of mixins.

In a class-based language, the representation of objects can differ from the representation of classes. Typically object representations are optimized for space, since in most situations there are many more objects than classes. We expect relatively few classes so we focus on speed instead of space. We first examine object layout in other languages and then describe the format used in OBSTACL, and well as how the various operations are implemented.

8.2 Layout in Other Languages

An object's representation must allow us to find its fields and methods. Much of the variation in object representation is a tradeoff between flexibility of the language and optimization of space or time. The key operations are selection and extension. In statically typed languages we also want subtyping. We consider here three dynamically typed languages (SELF, Python, and Smalltalk) and two statically typed languages (Java and C++).

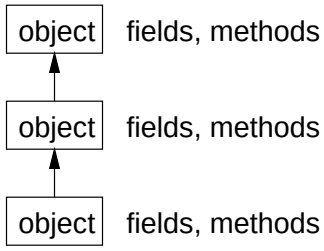


Figure 8.1: SELF: it's all objects

SELF SELF is a classless language, so the objects must support not only selection and comparison but also extension. An object contains both fields and methods, implemented as a

in an interface can lead to recompilation of all code that refers to that interface.

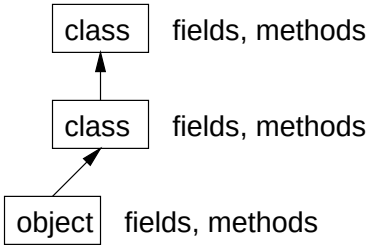


Figure 8.2: Python: like SELF, with an artificial distinction

map from names to either data or code. In addition, each object has a "parent" (see figure 8.1). If a message is not understood by the object, the message is delegated to the parent object. Delegation continues until either some ancestor processes the message, or a "root" object is found. JavaScript [Fla98] and MOO [Cur97] use a similar object model.

Python Unlike SELF, Python is a class-based language, with both classes and objects. However, Python allows both classes and objects to define new fields and methods, and also define or modify classes at run time, so the run-time structures are similar to those used in SELF (see figure 8.2). To find a method, the system first looks in the object; if the method is not defined there, the system proceeds to search in the object's class, then the class's ancestors. Although the language is class-based, the implementation is similar to that of an object-based language. In fact it is possible to build an object-based system in Python.[2]

Smalltalk Like Python, Smalltalk is a class-based language. However, unlike Python each instance object cannot have new fields or methods (see figure 8.3); instead, fields and methods are added as extensions to classes. As a result, the implementation need not perform lookup in the object; it instead starts with the class, then looks in each of the class's ancestors until the method is found. Like Python, Smalltalk allows classes to be defined and modified at run time; however, objects cannot be modified in this way. An object's representation is simpler: since all instances of a class have the same fields, the mapping from names to locations within the object can be stored in the class. An object requires minimal space, containing only its fields and a pointer to the class. A class contains a mapping from field names to offsets within the object, a mapping from method names to method implementations, and a pointer to the parent class.

Java Java's object layout [LY96] (see figure 8.4) is similar to that of Smalltalk, but Java's class layout can take advantage of an additional restriction: once classes are loaded, new methods and fields cannot be added to them. The method lookup table of a superclass can

[2]See, for example, Pythonic MOO [Str], extension classes [Ful96], and Python metaclasses [vR99].

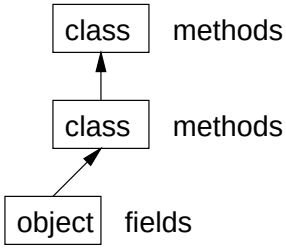


Figure 8.3: Smalltalk: classes have methods, objects have fields

be merged into the table of the subclass at link time. Method lookup is performed in one table only; if the method is not there, it cannot be defined in a superclass. In addition, Java is statically typed, so most method lookup failures are detected at compile time. Modula-3 and Eiffel can be implemented with a similar model.

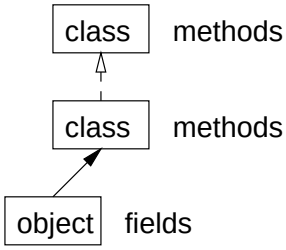


Figure 8.4: Java: similar to Smalltalk, but less dynamic

Java is statically typed, so we need a way to pass objects instantiated from subclasses to functions expecting a superclass instance. Java uses only single inheritance, so it is possible to lay out the fields of a subclass object so that it contains a superclass object layout as a prefix. When the subclass object is passed to a function expecting only a superclass instance, it can ignore the subclass-specific fields at the end. Java also provides interfaces. When a subclass object is passed to a function expecting an object implementing some interface, that function looks up names at run time in a map. One supertype (the type of the superclass) is treated differently from other supertypes (interfaces).

C++ C++ is even more restrictive than Java in that the ancestors of a class must be compiled in order to compile a subclass. The method lookup tables can be merged at compile time, so the location of methods within the table is known statically. Although a class may container a pointer to its parent class at run time (see figure 8.5), such a pointer is not necessary. In addition, the placement of fields is known statically, so methods can be compiled with precise knowledge of where fields are to be found. Like Java, C++ uses the prefix implementation for single inheritance. For multiple inheritance and virtual inheritance, common C++ compilers use a “pointer offset” implementation, where some

offset from the beginning of an object contains a prefix that matches each superclass. For example, if class C extends A and B, then objects of class C might contain a prefix matching A at offset 0 and a prefix matching B at offset 3. When passing objects statically known to match class C to a function expecting an object of class B, the compiler passes the object pointer *minus* 3. With virtual inheritance, the offset is not known statically, and is found inside the object itself. The details are complex; see [Str94] for details. The complexity (relative to Java) is needed to avoid all run-time name lookups and for uniform treatment of supertypes.

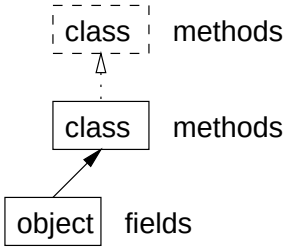


Figure 8.5: C++: class hierarchies aren't used at run-time

8.3 OBSTACL object and class layout

OBSTACL is a class-based language in which methods are defined in classes and fields are per object. A good starting point would be to consider the layout of objects in Smalltalk, Java, or C++. Like Java and C++, OBSTACL classes cannot be modified at run time. Like Java, OBSTACL supports interface types, and implementation details of a class may change without recompiling code that uses the class. (However, the implementation details may not change after linking the program modules together.) We therefore expect an object/class layout most similar to that of Java.

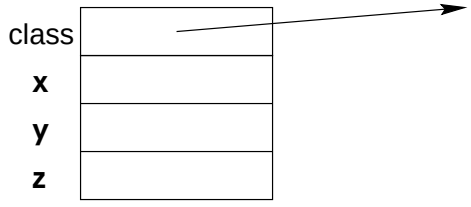


Figure 8.6: An OBSTACL object

Object layout is fairly straightforward in most class-based object-oriented languages. OBSTACL can use this format, shown in figure 8.6. All values in ML[3] use one word of storage to

[3] I write "in ML" even though I mean "in most implementations of ML." The former is shorter.

either hold all of the value or a pointer to additional data [App98].[4] We therefore expect an object with F fields to use $F + 1$ words (with one word pointing to the class structure). Note that neither field names nor methods are stored in the object; since they are the same for all instances of the class, they can be stored in the class instead of once per object.

Run-time class structures are where the interesting information is stored. Since there are far fewer classes than objects, and because languages offer so many different features that require run-time support, there is greater variation in class layout than in object layout. For OBSTACL, the key features we must support are instantiation of objects, with the two-step initialization system described in section 4.3.2, and subclassing *at run time* to support run time inheritance (see section 4.2.5), including mixins (see section 5.3). In addition, since methods are stored in classes instead of in objects, we must be able to find the class of an object in order to select a method. Depending on the separate compilation system used, we may also need a description of the class signature so that we can verify type safety at link time. Of these, supporting run-time inheritance the most challenging.

In most object-oriented languages, a single class declaration becomes a single class at run time, but in OBSTACL, each class declaration can expand into an unlimited number of classes at run time.[5] Figure 8.7 demonstrates the need for run-time class structures. The run-time system must provide a way for *instances* of $f(5)$ to access the *class* returned by $f(5)$, which in turn must have access to the *environment* present at the time the class declaration C was executed, which provides access to the mapping $k \mapsto 5$.

```
fun f(k) =
  class C
    method m() = k;
  end;
```

Figure 8.7: Run-time class generation

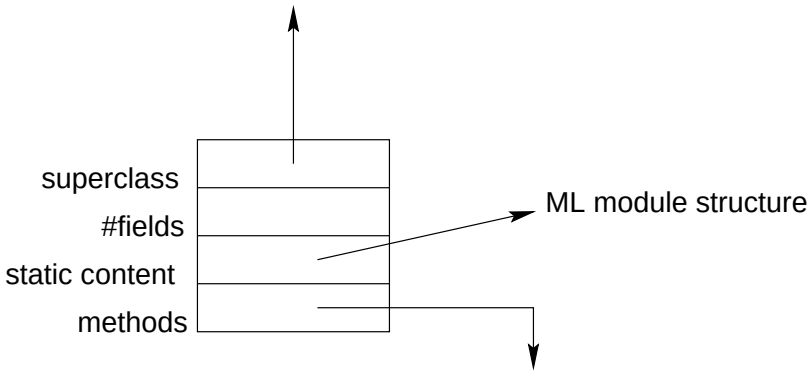
A more demanding example, shown in figure 8.8, defies any attempt to use only one environment pointer in a class. The result of $f(n)$ is a class with n ancestors ($f(n-1)$, $f(n-2)$, ..., $f(0)$) and $n+1$ entries in its method table. The class produces objects with $n+1$ fields. Each of the $n+1$ methods has its own sense of what i and j are, so we must keep in mind the need for multiple environment pointers when designing the run-time class structures. In addition, this example shows that the compiler cannot determine the object size from the class definition; the object size is not available until run time.

[4] The TIL compiler can inline values of known type, such as records, into their containers [TMC+96]. We expect the same optimization to apply when structured values are fields of an object. The object itself, however, cannot be inlined into its container because of substitutivity—there remains the possibility that the instantiator or other function has given us an object with the same type but different internal structure and size.

[5] In C++, a single template class declaration can expand into multiple classes, but this expansion is performed at compile or link time, and in most compilers the expanded classes are individually compiled. In OBSTACL, the class is compiled only once, and expanded at run time.

```
fun f(0) =
        (class C
           field j;
           method m() = j;
         end;
         C)
  | f(k) =
        (class C extends f(k − 1)
           field j;
           method m() = x * next() + k;
         end;
         C)
```

Figure 8.8: Run-time class hierarchy generation



Method	Name	Environment	Code	Field Offset	Super
B.n	"n"	env_B	code	offset_B	-
B.m	-	env_B	code	offset_B	-
D.p	"p"	env_D	code	offset_D	-
D.m	"m"	env_D	code	offset_D	1

Table 8.1: Class and method table layout

Table 8.1 shows a class layout that supports the key features of OBSTACL classes. A class points to its superclass so that we can chain instantiators and constructors and also for run-time tools such as debuggers, but the superclass link is not followed for method lookup. Instantiators and constructors, used to create objects, are parts of classes, not of objects they create. We can represent them in the same way as we represent module functions [App98]. Methods however are conceptually part of an object and require some mechanism for finding object data. It would be straightforward to create a closure per method to include in each object, but that increases the size of the object significantly. Our design of method tables allows them to be stored once, in the class, yet includes just enough information to recover the per-object data.

In section 8.4 we describe how operations are implemented using these object and class layouts.

8.4 Operations

The operations on an OBSTACL object are selection, subsumption, and comparison. Within a method, the operations are field lookup, access to the "self" object, and access to the redefined method. For a class, the operations are instantiation, subclassing, and mixin application.

8.4.1 Operations on objects

Selection

To select a public method on an object involves looking at the method table of the object's class. The method table is a hash table with additional entries for chaining to implement method redefinition. Selection does not involve looking at the chained entries.

```
select(object, method):
    class = object.class
    table = class.methodtable
    entry = hashtable_find(table, hash(method), method)
    entry.code(object, entry)
```

In section 8.4.2, we discuss what the method does with the entry. Note that this operation does not *call* the method. It simply sets up the necessary bindings for the method, such as "self". Also note that this implementation also works for public fields, if we choose to add them (see section 9.1.2).

In C++, methods can be found at a known position in the method dictionary. This optimization relies on subtyping requiring inheritance—the dictionary layout is known for all subtypes because they must also share implementation. With structural subtyping, classes that are not related by inheritance can produce types that *are* related through subtyping. We cannot guarantee that subtypes share a dictionary layout, so the optimization cannot be made. For this reason, the name of the method is used for method lookup instead of an index into the dictionary, and we expect OBSTACL method lookup to be slower than C++'s. However, we can use two optimizations to make lookup faster than simple hash table lookups of method names.

The only operation needed on the method names is equality. The first optimization is to replace each string with a unique value, different for each method name.[6] During module initialization, each name is looked up in a global mapping and its number is found. Method tables are built with the numbers instead of the strings, and method calls perform lookup using the number. Searching the dictionary is still needed, but cheap integer comparisons are used instead of relatively expensive string comparisons.

The other optimization is to cache the result of the last lookup at each call site. At each call site, we store an integer *hint* to the method lookup code. After each successful method

[6] In Lisp, these are called *interned symbols*.

lookup, we store the index of the found method entry. At the next method lookup, we use this hint to check that entry in the method table first, and only if the needed value is not there do we perform the hash table lookup. When successive method calls (from the same call site) are to objects of the same class or to classes that have the same method table layout, the hint will give fast access to the method without going through the cost of hash table lookup. The optimization is particularly useful when the classes being used share implementation through inheritance, and the method tables are constructed appropriately (see section 8.4.3). We pay the full cost of the hash table only when the flexibility of structural subtyping is required; if only traditional inheritance-based subtyping is needed, method lookup is nearly as fast as in C++.

We do not expect as much benefit from whole program analysis [CDG97, DDG+96] as in pure object-oriented languages because these optimizations work better for objects that do not get used in "polymorphic" ways (multiple implementations of a method); in OBSTACL many of these classes are written as regular ML data structures, and are optimized already.

Subsumption

Our object layout does not depend directly on the static type of the object. Since fields are private, access to data must go through methods, which are looked up in a table. Therefore, a pointer to an object viewed as a subtype can be used when viewed as a supertype, and subsumption is free. In Thorup's encoding of objects in ML, subsumption involves a record copy (and change in format). On the other hand, copying the record to a new format for each supertype allows the compiler to know the exact layout of fields and methods, so access is faster (through indirection instead of lookup). C++ takes a middle ground by restricting the subtyping relationship. With only a few supertypes, the compiler can build all the records at compile time. C++ subsumption is free with single inheritance, because fields and methods are kept in a particular order, but with multiple inheritance, subsumption involves adjusting offsets to pointers.

Equality

Two objects are equal if they are produced by the same constructor call (see section 4.1.3). To compare two objects for equality is trivial: we simply compare their pointers. We are free to do this because an object has only one address. If subsumption were implemented as a record copy (as in Thorup's encoding of objects in ML), an object may have multiple addresses. In figure 8.9, `o2` and `o3` may have different addresses and thus are not equal using pointer equality. Under our proposed implementation, subsumption does not allocate a new record, and the pointers for `o1`, `o2`, and `o3` all point to the same place. Therefore, equality checking is trivial.

```
let val o1:Cat = make_object()
    val o2:Animal = o1
    val o3:Animal = o1
```

Figure 8.9: Subsumption may lead to multiple copies at the same type

8.4.2 Operations in methods

Once a method is called, it needs access to free variables, the "self" object, the object's fields, and the previously defined method (if there is one). We will consider each of these.

Free variables

Free variables of a method are found in the environment of the class definition. Following the standard implementation techniques for closures [App98], we can store in the class a pointer to the environment record for the class. We can find this pointer by using the "env" column of the method lookup table (see section 8.3). Since ML variables are immutable, one optimization we can make is to copy the values of the free variables[7] to the class structure at class definition time; if we were to add this object system to a different language, we could copy pointers to mutable values instead of the values themselves. A further optimization used for closures is also applicable here: the compiler can copy only those variables that actually occur free in the class definition. Although we are compiling a class with multiple methods instead of a single function, we can use the same techniques for storing and looking up free variables.

Self object reference

Since methods are compiled once per class definition, [8] the code does not have a pointer to the object. Each method $m : \alpha \to \beta$ can be compiled as a curried function ("pre-method"[9]) with extra arguments: $m^{\text{pre}} : (\sigma \times \gamma) \to \alpha \to \beta$ where σ is the type of the self object and γ is the type of a pointer or index to a row in the method table.[10] At selection time, the self object is known and the method table row is discovered, so we apply these to the pre-method, resulting in a function of type $\alpha \to \beta$. In the general case we build a closure here, but ML compilers can optimize applications of curried functions, and can avoid building the closure when the function is called immediately. The code `obj.m` results in a closure, but the more common `obj.m()` does not. Thus we get the flexibility of separating selection from method call without paying a performance penalty in the common case.

[7] Note that we only copy a word per variable, since most values are represented as a pointer to data on the heap.

[8] Given run-time inheritance, there are actually two "class definition" times we can speak of, instead of one. Free variables are copied at run time, each time the class definition is executed. Method bodies are compiled at compile time, and are shared among all classes using the class definition.

[9] A pre-method is a function that takes `self` and `private` fields as parameters. Identifiers that are bound in the method body are passed as parameters to a pre-method. The pre-method takes an object and object state and then returns a function which can be thought of as a method belonging to that object.

[10] There need not be a real ML type for method table rows. The "type" of a method table row is for explanatory purposes.

Field lookup

Given an object, a method must be able to find the fields that method can access. In some languages, field locations within an object are known at compile time (in C++) or link time (in Java). The compiled method can find the field at a fixed offset from the pointer to self. In more dynamic systems, such as Smalltalk and Python, fields are found through a lookup in a table. OBSTACL falls somewhere in between. Modularity constraints (see section 3.1), run-time inheritance (see section 4.2.5), and mixins (see section 5.3) make it impossible to know field offsets at compile time. Yet, because all fields are private (see section 5.2.1), a full field lookup is unnecessary.

In C++, a field's location in an object is known at compile time.[11] To ensure that this is possible with separate compilation, a class header file must list *private* fields, and a class's field offsets must remain the same in subclasses. Having field offsets at compile time makes field access very fast. However, the disadvantages are that private fields are exposed to users, and subclasses must be recompiled when the superclass's implementation details change.[12]

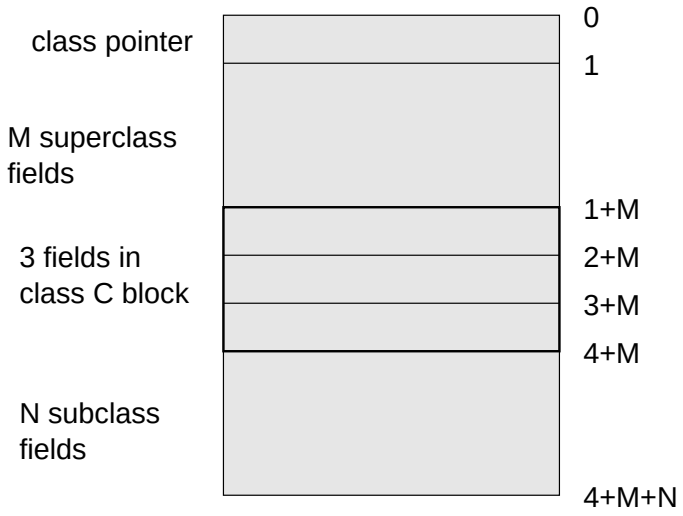


Figure 8.10: Layout of an OBSTACL object

In OBSTACL, private fields are not visible to subclasses. In addition, one of our goals is the ability to change implementation details without recompiling subclasses. With run-time inheritance and mixins, it is possible to compile subclasses before the superclass has been written. For these reasons field offsets cannot be known at compile (or even link) time. Consider for example an instance of a class `N` which inherits class `C`, which in turn inherits class `M`. Since

[11] Except in the case of virtual inheritance, which we will not describe here. For implementation details, see [Str94, section 15.2.4]

[12] Both of these are common complaints about C++. Had C++ been designed differently, the same users would complain about performance of field lookup. There are no easy answers.

```
let val private = Unsafe.cast(self + entry.offset)
    val {x,y,z} = private in
 (* method body, which can use x, y, z *)
end
```

Figure 8.11: Pattern matching to access private fields

all fields are private, a method defined in class C can access only fields also defined in class C and not those defined in class M or class N. The fields of class C can be stored contiguously in the object, as depicted in figure 8.10. Although the offset from the beginning of the object is unknown, the offset from class C's block of fields is known statically. Thus we need only a way to find the beginning of class C's block. In the method table, the "offset" column stores the offset of class C's block from the beginning of the object. Since the method receives `self` and the method table row `entry`, its private field block `private` begins at `self + entry.offset`. Fields x, y, z are at `private`, `private+1`, `private+2`.[13] Another way to think of field access is to use pattern matching, as shown in figure 8.11. (We used this approach in one of our prototype implementations of OBSTACL.) Field access in OBSTACL therefore involves a small computation of `private` once for each invocation of a method that accesses fields. After this `private` is computed, field access in the method is as fast as field access in C++. We therefore maintain our modularity goals as well as the flexibility of run-time and mixin inheritance without a significant performance hit.

Access to the redefined method

If a method body is a redefinition of a method in the superclass (*i.e.*, it has the same name), the redefinition in the subclass may need to call the redefined method in the superclass. In Smalltalk and Python, the redefined method is looked up at run time, at each invocation. In C++, the redefined method is known at compile time, so no lookup is needed. Given run-time inheritance in OBSTACL (including mixins), a reference to the redefined method is not available at compile time. However, OBSTACL's `next` construct gives access to the one method being redefined, not to arbitrary definitions in any ancestor class, and a single method can be found without lookup.

Recall from figure 8.1 the method table includes entries for all redefined methods. These entries are not found during lookup because their "name" column is blank. However, other entries can refer to them through the index in the "super" column. Consider as an example class D inherits from C inherits from B inherits from A. Classes A, B, and D define method `m`. Method `D.m` redefined `B.m`, which redefined `A.m`. All three are in the method table, shown in figure 8.12, but only `D.m` has "m" in the name column, so only it is found during method

[13] For clarity, these numbers refer to words, not bytes.

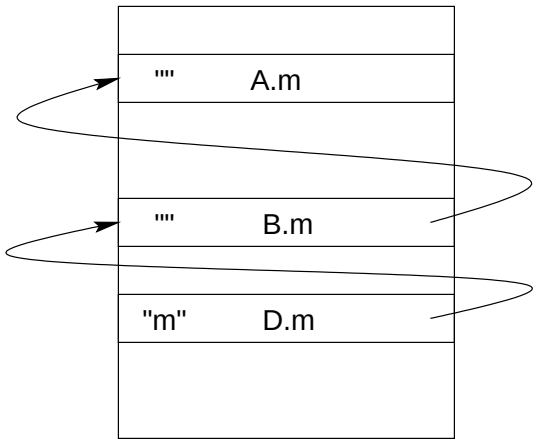


Figure 8.12: Method table chaining

lookup. Hoewver, `D.m` may call `B.m` through `next`. If that happens, the compiled code finds `B.m`'s method table entry by following the pointer in the "super" column of `D.m`'s entry. With the self object and the method table entry, we can call a method (see section 8.4.1). If `B.m` needs access to `A.m`, it can follow the same procedure, using the "super" column in its method table entry. With this scheme, access to redefined methods is fast (requiring no lookup), yet flexible enough to allow for run-time inheritance and mixins.

8.4.3 Operations on classes

The main operations on classes are instantiation and extension. For instantiation, we want a distributed initialization (see section 4.3.2). For extension, we must support run-time inheritance (see section 4.2.5) and mixins.

Instantiation

Instantiation would be a straightforward process but for modular initialization. Each class initializes its own fields and sets up any necessary invariants, and it should also allow its ancestors to do the same (see section 5.2.4). The sequence of operations is shown in figure 8.13. When the instantiator invokes `new`, the run-time system allocates a new object and sets its class pointer. We know the size of the object, since it is stored with the class structure (see section 8.3). We then call a constructor of `C`, which initializes the fields defined in `C`. The constructor has a pointer to the object, and a pointer to its class—a constructor $\texttt{C} : \alpha$ is compiled as a function $\texttt{C} : \sigma \times \gamma \times \alpha \times \mathsf{unit}$, which takes as an extra parameter a pointer to the new object and a pointer to the class structure. Using the object size from the class

structure, the constructor can determine where fields are placed.[14] After initializing fields, the constructor calls a superclass constructor with a pointer to the object and a pointer to the superclass structure. When the superclass constructor returns, the constructor sets up any invariants that are not expressed in field values alone. Finally, the constructor function returns.

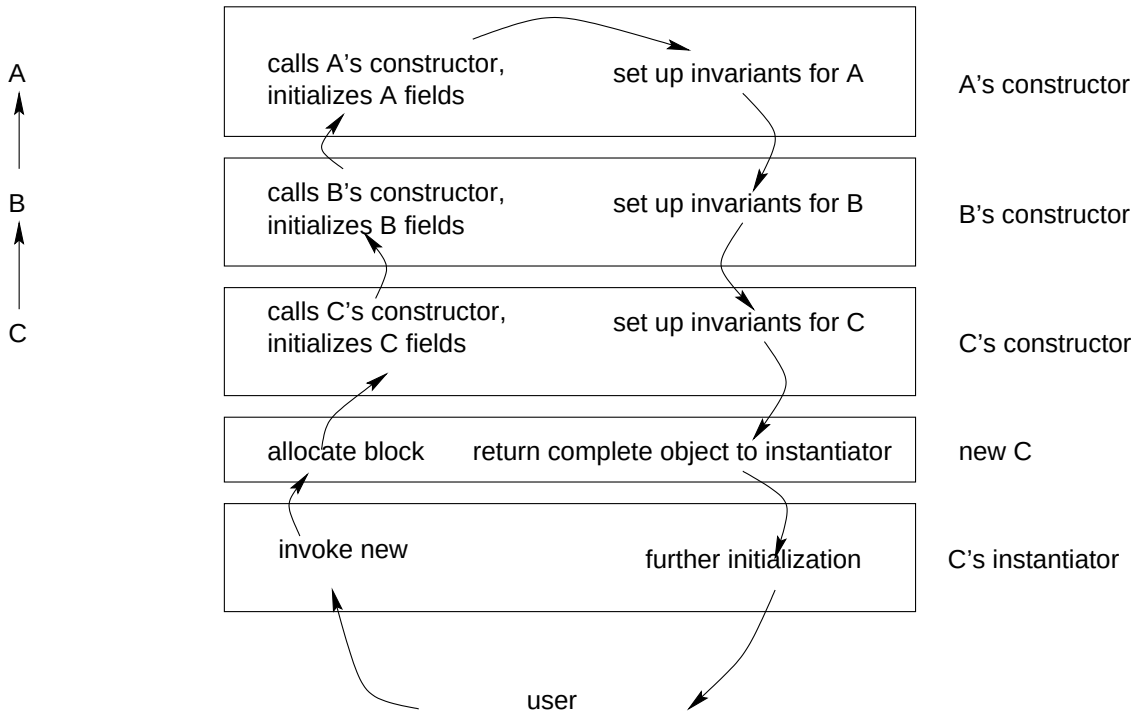


Figure 8.13: Steps in constructing an object

To illustrate this process, we show an example in figure 8.14 of a three class hierarchy as well as the expected resulting object structure. The constructor for C initializes fields at offset `C.objsize - C.numfields` = $7 - 3$ = 4. It then calls B's constructor, with a pointer to the object and a pointer to class B.[15] The constructor for B initializes its field at $4 - 1 = 3$, and calls A's constructor. The constructor for A initializes fields at $3 - 2 = 1$.

Extension

In OBSTACL, we must build class structures at run time. At compile time, the compiler knows the static types of the superclass and subclass, but not until run time is the exact superclass known. To build the subclass, the run-time library examines the superclass, using the

[14] Fields are placed at an offset `class.objsize - class.numfields` from the beginning of the object. This is the same offset put into the "offset" column of the method table.

[15] Recall that each class stores a pointer to its superclass.

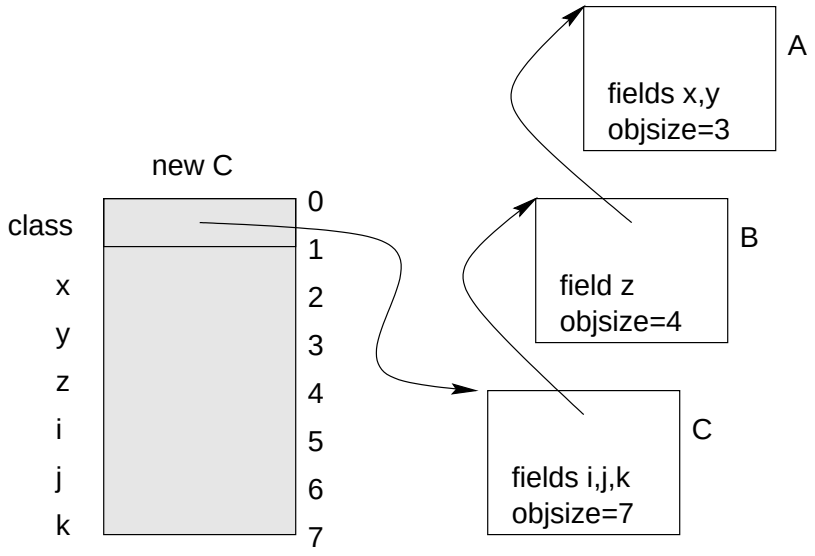


Figure 8.14: Class layout example

object size, method table, and constructor list. The new subclass contains an object size equal to the superclass object size plus the size of the new subclass fields. Recall from section 5.2.5 that constructors invoke superclass constructors but instantiators do not invoke superclass instantiators. Consequently, the subclass contains an instantiator list that does not depend on the superclass instantiator list, and a constructor list that *does* depend on the superclass constructor list, but only those that are invoked by subclass constructors. The constructor list can be compiled in the same way as ML modules. What requires more care is building the method table—the heart of the run-time class structure.

A method table in OBSTACL contains both methods found during selection and redefined methods invoked through the `next` construct (see section 8.3). Creating a subclass method table from a superclass method table involves copying entries for methods not redefined in the subclass, altering entries for methods redefined in the subclass, and creating entries for each new method definition in the subclass.

Suppose a superclass method table contains M entries and the deriving class adds N new entries and redefines R entries. The new method table has $M + N + R$ entries. Given an old method table super (array of $\langle \mathrm{name}, \mathrm{env}, \mathrm{code}, \mathrm{offset}, \mathrm{super{:}int} \rangle$), a list of new methods new (array of $\langle \mathrm{name}, \mathrm{code} \rangle$), and a list of redefinitions $\mathrm{redefined}$ (array of $\langle \mathrm{name}, \mathrm{code} \rangle$), we can construct a new method table for class C using the algorithm shown in figure 8.15. This algorithm preserves array positions of each name to enhance the effectiveness of caching (see section 8.4.1). For example, if "m" occurs in position 3 in the superclass table, and there is a redefinition of "m" in the subclass, then the subclass table will contain "m" at position 3 pointing to the new definition, with a link to the old definition, moved to the end of the table.

(* Create a new method table *)
sub: = allocate $(M + N + R)$ entries
(* Initially set it to the same as the old table *)
for i in $0..M - 1$:
  set sub_i := super_i
(* Add new methods *)
for i in $0..N - 1$:
  set sub_{M+i} := $\langle \text{new}_i.\text{name}, \text{env}_C, \text{new}_i.\text{code}, \text{offset}_C, -1 \rangle$
(* Deal with redefined methods *)
for i in $0..R - 1$:
  look up j such that $\text{redefined}_i.\text{name} = \text{sub}_j.\text{name}$
  (* Move the method being redefined down to the end *)
  set sub_{M+N+i} := sub_j
  set sub_{M+N+i}.name := " "
  look up k such that $\text{sub}_k.\text{super} = j$
  if such a k exists:
    set sub_k.super := M+N+i
  (* Create a new entry in its place that points to the old one *)
  set sub_j := $\langle \text{redefined}_i.\text{name}, \text{env}_C, \text{redefined}_i.\text{code}, \text{offset}_C, M + N + i \rangle$

Figure 8.15: Method table construction algorithm

Run-time inheritance In many class-based languages, classes are produced at compile time through declarations. At the time a class declaration is compiled, the compiler can determine the superclass, its layout, and its methods. In OBSTACL, the superclass methods are known at compile time, but the actual pointers to these methods, and the object layout, are not known until run time. Method pointers are stored in the "code" column of the method table; information about the object layout are stored in the "offset" column. In addition, since a single class declaration may spawn multiple classes, methods cannot be compiled to use a single environment to find values of identifiers. A pointer to the environment is kept in the "environment" column of the method table.

Mixins With mixins, the set of methods is unknown at compile time. To construct method tables of mixins, our algorithm must not rely on a fixed set of methods and method types; it should copy unknown methods from the superclass to the subclass.

8.5 Time and Space Requirements

Table 8.2 compares the space and speed for object representations in different systems. "Objs in ML" refers to an encoding of objects in Standard ML, suggested by Lars Thorup [TT94]. We use this encoding for comparison to illustrate that directly implementing object-oriented programming in the system has advantages over an encoding, which could be provided by a "pre-processor", such as the one used for early C++. In the table, the variables are: v for

	Space	Time		
	object size (words)	subsumption	method lookup	field lookup
Smalltalk	$v + f + 1$	0	1 $\langle \varepsilon m \rangle^4$	1 $\langle \varepsilon f \rangle^4$
C++	$v + f + 1 \langle \ldots + p \rangle^1$	0 $\langle 1 \rangle^2$	1	1 $\langle p \rangle^1$
Objs in ML	$v + f + m$ $\langle \infty \rangle^3$	0 $\langle m + f \rangle^3$	1	1
OBSTACL	$v + f + 1$	0	1 $\langle \varepsilon m \rangle^4$	1 $\langle \varepsilon f \rangle^4$

Table 8.2: Performance characteristics of object-oriented languages

allocation overhead, f for the number of fields, m for the number of methods, and p for the number of ancestors. The values in angle brackets ($\langle \ldots \rangle$) represent infrequently occurring worst-case conditions:

1. Using virtual base classes in C++ usually forces the compiler to include additional pointers in the object layout (up to one per ancestor). Field lookup in this case is also slower, since these pointers must be followed to find fields defined in virtual base classes.

2. Using multiple inheritance can make subsumption a non-zero-cost operation. Pointers have to be adjusted by an offset when converting from a subclass pointer to a superclass pointer.

3. The Objects in ML format requires a new object to be created when changing from a subclass pointer to a superclass pointer. This makes subsumption a non-zero operation; it also means that there may be an unlimited number of "converted" objects in memory.

4. Method (and field) lookup speed depends on the data structure being used for the method dictionary. Assuming a hash table, most methods will be found in the first step of the hash table search. In the worst case, a hash table conflict will force a longer search time, but this time can be minimized by using multiple sparse hash tables for different sets of method names. If $1/\varepsilon$ tables are used, only $m\varepsilon$ entries are in a table, so the longest search (involving all the method names having the same hash value) will take $O(m\varepsilon)$ time.

Note in particular that both of our object layouts lead to zero overhead for subsumption, an object size that does not depend on m, and fast method/field lookup.

In C++, the efficiency of accessing an item depends on whether it's a field or method. Field access depends on the inheritance hierarchy (virtual base classes in particular), and method access depends on whether it is declared `virtual`. In our language, the efficiency depends on the access specifier, not whether it is a field or method. Private items are fast while public items are slower. Protected items are represented as public items that are hidden to the clients, so they are as efficient as public items.

The syntax of our language reflects this difference. Private items can be accessed by naming the item. In ML, referring to an identifier is a fast operation. Public and protected items are

accessed with the dot syntax. Access using dot is expected to be roughly as fast as record field access for a system which allows structural subtyping on records.

A difference not expressed in the above chart is that mixin compilation is different from template compilation in C++. In nearly all C++ compilers, each instance of a template leads to recompiling that template. In contrast, an OBSTACL mixin is compiled only once. The code for constructors, instantiators, and methods is generated once. The associated data (the method table) is generated once per instance, and provides the link between the instance-independent portion of the resulting mixin and the instance-dependent portion of the mixin.

The design of OBSTACL imposes unusual demands on an implementation. Dynamic languages like Smalltalk and Python allow for run-time inheritance but also defer name and type checking until run time. Static languages like C++ check names and types at compile time but do not offer run-time inheritance or mixins. Java is more dynamic, allowing for run-time loading of classes, but these classes must be compiled individually. OBSTACL is between the static and the dynamic languages, with static name and type checking and creation of new classes at run time. The implementation proposed in this chapter preserves the efficiency of a statically type checked language in the presence of new classes generated at run time with an unknown number of fields and methods. In addition, it allows for separate recompilation that matches the expectations of the programmer.

Chapter 9

Extensions

In chapter 6 we analyzed OBSTACL's language features; in this chapter we look at possible extensions to OBSTACL. Section 9.1 describes extensions that involve syntactic changes and simple changes that do not affect the characteristics of the language. Section 9.2 describes features that could be added but have costs that outweigh the benefits in a language like OBSTACL. Section 9.3 describes features that could either be supported directly or expressed as a combination of features. Section 9.4 includes alternative designs for parts of the language.

9.1 Simple Extensions

The core language described in chapter 5 includes constructs necessary for expressing program components with desired modularity constraints, but not constructs that exist primarily for convenience. In this section we look at language features that may be useful to programmers, and how they map to combinations of features already in OBSTACL.

9.1.1 Type Names

Object types in C++ and Java are simple names. OBSTACL types are structural (see section 4.1.4) so they list the names and types of all methods in the object. Unfortunately writing out structural object types can be cumbersome. We therefore use these forms as abbreviations for the longer structural types:

C `object`	public interface of class C
`Self`	public and protected interface of the current class (used as the type of the "self" object)
$\tau_1 @ \tau_2$	union of two object types
$\tau \backslash \iota$	difference (where ι is a list of names)

In mixins, `Self` refers to the supertype statically known through the mixin constraints.

```
class C
   (* private *) field x:int ref;
   (* public *) method get_x():int = !x;
   (* public *) method set_x(x':int):unit = (x := x');
   ...
end;
```

Figure 9.1: Accessor methods give public access to private fields

```
class C
   (* private *) field x:int ref;
   (* public *) method x():int ref = x;
   ...
end;
```

Figure 9.2: Accessor methods returning an ML ref

9.1.2 Public and Protected Fields

Although public fields are generally discouraged in object-oriented designs, their effect can be simulated by creating a private field along with public "accessor" methods (see figure 9.1). Since ML allows the direct export of the reference value, a single method can return x, allowing the object user to either use `!` to read the value or `:=` to write a value (see figure 9.2). Note that we can use the same name for both the field and the method, because private names (fields) and public names (methods) are in separate namespaces. Field names are visible in method bodies, but method names are seen only in the "self" object. Given the above arrangement, there remains but one small step to make x truly look like a public field (see figure 9.3). If we extend OBSTACL to support methods that are not functions, public fields are easy to add. The syntax "public `field` $x : \tau$" would be converted into (private) "`field` $x : \tau$" followed by (public) "`method` $x : \tau = x$". Protected fields are similarly decomposed into private fields with a protected accessor. Both public and protected fields can be supported with only minor changes to OBSTACL.

```
class C
   (* private *) field x:int ref;
   (* public *) method x:int ref = x;
   ...
end;
```

Figure 9.3: Non-function accessors

```
class C
   field f1;
   field f2;
   ...
   field fn;
   private method p(args) = ...;
   public method m() = p(3,5);
   ...
end;
```

Figure 9.4: Private methods

```
fun p(self, f1, f2, ..., fn)(args) = ...;
class C
   field f1;
   field f2;
   ...
   field fn;
   method m() = p(self, f1, f2, ..., fn)(3,5);
   ...
end;
```

Figure 9.5: Module-level pre-methods simulate private methods

9.1.3 Private Methods

Having seen public and protected methods to simulate public and protected fields, we may ask if private fields can be used to simulate private methods. Since methods are shared among all instances of a class, there is no need to make a private method into a field. We do use the binding approach used for private fields to build a private "pre-method". The code in figure 9.4 can be simulated by writing pre-methods, as shown in figure 9.5. A method has implicit bindings for the host object ("self") and the private fields; in a pre-method we make these bindings explicit. Each call to the method is replaced by a call to the pre-method function with extra arguments. This transformation is possible because the method is private and therefore all calls to it are seen in the class declaration and can be changed, and methods are merely functions with additional bindings so they can be moved outside the class declaration. If methods and classes had been subject to different rules than functions and scopes, implementing private methods in this way may not have been straightforward.

9.1.4 Method Update

In OBSTACL a method body is shared among all instances of a class. In some situations it may be useful for each instance to have a different method. Many dynamic object-oriented

```
class C
   field f1 : 't1;
   field f2 : 't2;
   ...
   field fn : 'tn;
   field m:((C self * 't1 * 't2 * ... * 'tn) -> 'a -> 'b) ref;
   method m(a:'a):'b = (!m)(self, f1, f2, ..., fn)(a);
   ...
end;
```

Figure 9.6: Method update

```
class C extends B
   ...
   redefine protected m = next;
   ...
end;
```

Figure 9.7: Hiding protected methods

languages support ***method update***, in which a method is replaced by another. In a strongly typed language, we may expect the new method to have the same type as (or a subtype of) the original method. Since OBSTACL supports updatable fields and ML supports first class functions, we can build an object with an updatable field that contains a function (pre-method). In the example code shown in figure 9.6, updating `m` involves assigning a suitable pre-method to the field `m`. Calls to `m` go through the `m` method, which dereferences the `m` field and then calls the resulting function. With support for imperative update and first class functions, method update is a simple extension of OBSTACL.

9.1.5 Hiding inherited methods

Although inheritance in OBSTACL naturally leads to subtyping, the two are separate concepts. At times it may be useful to build a subclass without having a subtyping relationship. In OBSTACL, all fields and methods are inherited (see section 4.2.4), so subtyping results by default, but the relationship can be broken by hiding some of the inherited methods. We can hide public methods by redefining them to be protected. The method which is public in the superclass is marked protected in the subclass, so the user will not see it. In the example shown in figure 9.7, `C` objects will not be subtypes of `B` objects. The problems described in sections 4.2.3 and 4.2.4 are avoided because inside the class the full type can be seen to be a subtype. It is only through standard subtyping rules that the subtyping relationship is hidden for the user (see the array example in section 5.1.3).

```
mixin M
   ...
   redefine f = next;
   ...
end;
```

Figure 9.8: Adding mixin constraints

```
f(class
     field f:int;
     method m(x:int) = f+x;
     constructor make(f0)  fields {f=f0};
     instantiator make(f0) = new make(f0);
  end)
```

Figure 9.9: Anonymous classes

9.1.6 Additional Mixin Constraints

The technique of redefining a method to be the same as the inherited method can also be used to create additional constraints for mixins. To state that a mixin requires some method to be present in the superclass, simply redefine the method to match the inherited method. In figure 9.8, the mixin `M` requires its superclass to contain a method `f`. To require the method to have a particular type, a type can be given explicitly in the redefinition.[1]

9.1.7 Anonymous Classes

ML supports both function *declarations* and *anonymous* functions. Anonymous classes (and mixins) could be added to OBSTACL as well, with minor changes to syntax; they are already supported by the calculus. The code in figure 9.9 would create a local class and pass it to a function. The two changes to the syntax are:

- A class name does not follow the "class" keyword.
- A class name does not follow the "new" keyword (in the instantiator).

The second change does not pose a problem except in the case where the instantiator declares a local class, as illustrated in figure 9.10. If the class names are omitted, the inner instantiator would not work as written. However, this kind of program structure does not appear to be common. The calculus for core OBSTACL (see chapter 7) goes one step further, and treats *all* classes and mixins as anonymous.

[1]There are actually two subtyping relationships that can be given, one for the upper bound and one for the lower bound. See chapter 7 for a description of mixin constraint inference.

```
class A
   ...
   instantiator make() =
      let class B
             ...
             instantiator make() =
                new B make(new A make());
          end;
      in
         ...
      end;
end;
```

Figure 9.10: Anonymous classes introduce ambiguity

9.1.8 Objects without Classes

Given OBSTACL's object system, all objects are instances of some class. Occasionally it may be useful to create an object directly. Given anonymous local classes, there is no advantage of direct object construction other than syntax, so we may transform the object definition shown in figure 9.11 into an equivalent definition using anonymous classes, shown in figure 9.12.

```
let f = ref 3 in
   object
      method inc() = (f := !f+1);
      method get() = !f;
   end;
```

Figure 9.11: A classless object

```
let f = ref 3 in
  (class
      method inc() = (f := !f+1);
      method get() = !f;
      constructor make() fields {};
      instantiator make() = new make();
  ).make();
```

Figure 9.12: Anonymous class used once

9.1.9 Standard Instantiators

Many examples we have presented do not use the full power of instantiators. For these classes, the instantiator simply creates an object and returns it. Since this usage (shown in

```
class C
   ...
   constructor make(args) ... ;
   instantiator make(args) = new C make(args);
end;
```

Figure 9.13: Standard instantiator

figure 9.13) is so common, it can be abbreviated as "`instantiator` make;", which generates a standard instantiator, which returns a new object generated by calling the constructor with the same name, passing to it the same arguments passed to the instantiator.

9.1.10 Abstract Classes

Many languages support the notion of an ***abstract class***: a class which is missing some of its implementation and cannot be instantiated. In OBSTACL any class that lacks an instantiator cannot be instantiated. However, classes must have a complete implementation, in the sense that all methods listed in the type of "self" must be defined in the class. To support unimplemented methods, new syntax should be introduced that simply adds method signatures to the self type. A class with unimplemented methods cannot be instantiated with the `new` operator, but *it may still contain an instantiator.* For an example of how this could be useful, see the virtual constructor pattern (section 6.2.2).

9.1.11 Classes as Modules

Object creators do not need to see individual method and constructor declarations in a class. Object creators need only to see the object type and the instantiators. If instead of using C `object` as an abbreviation for the type of objects produced by class C, we use C.`object`, then classes look much like ML modules. One simple extension of OBSTACL would be to allow module signatures to match classes, so that classes can be used as functor arguments. Figure 9.14 shows a class and its corresponding module signature. Class signatures would be used by subclasses, while module signatures would be used by object creators.

The advantage of this approach is that the object creator can create objects without knowing whether they are instances of a class or some combination of objects. For example, a windowing toolkit might provide a tabbed dialog "class" with an "instantiator" that is an ordinary function that composes tab objects with a dialog object. The object creator can consider the resulting object to be an instance of a class without realizing that it is implemented in terms of simpler objects.

Another important use of this extension is in object versioning. If an interface has been changed from version 1 of a library to version 2, then for backwards compatibility both versions

```
class C
   field f:int;
   method m() = f+3;
   constructor mk_one() fields {f=1};
   instantiator mk() = new C mk_one();
end;

sig
   type object = {|m:unit->int|};
   val mk:unit->object;
end;
```

Figure 9.14: Module signature for a class

may need to be supported. To support both versions, the vendor can provide a version 2 object in addition to an *adapter* object (see section 6.3.1) that uses a version 2 object internally but presents a version 1 interface to old code. The version 1 "class" is actually a module with an "instantiator" that creates the version 2 object and an adapter to it, and returns the adapter.

9.1.12 Destructors

OBSTACL does not directly support ***destructors***, functions that are called at the end of an object's lifetime. In C++, most destructors are used to deallocate memory. In a garbage collected language, such a step is unnecessary. However, destructors may also free resources not managed by the garbage collector, such as network connections, dialog boxes, and shared memory regions. To free these sorts of resources, Java provides ***finalizers***, which are functions called before the garbage collector deallocates an object. For OBSTACL, we believe that finalizers are not specific to objects, and instead could be provided for all ML values. If such a feature were available, any objects that require finalization could register a finalizer function during object construction. For convenience, it may be useful to provide syntax for declaring and automatically registering the finalizer.

One problem with finalizers is the "resurrection" of the object which no longer has references to it. The execution of the finalization function can introduce a new reference to it, and the object is now "undead" [Cha99]. In OBSTACL we can instead register a finalizer to run when an object is no longer accessible, but instead of giving the *object* to the finalizer function, we can close the finalizer function over the private fields by defining it as a local function inside the "initialization" section of the constructor. If we use this form of finalizer, we can avoid object resurrection issues.

9.1.13 Mixin Composition

OBSTACL includes mixin definition and application of a mixin to a class, producing a class. A natural extension would be application of a mixin to a mixin, producing a mixin. We can define a mixin compose operator $\circ$, such that $(F \circ G)\langle C \rangle = F\langle G\langle C \rangle\rangle$. A type rule for mixin composition is straightforward, but tedious. Table 9.1 shows the constraints arising from different combinations of method definitions. A new method in C could be redefined in F, G, both, or neither. A new method in F may or may not be redefined in G. A method defined in F or G must not be present in C. A redefinition must have a subtype of the redefined method. From the constraints and types for F and G, we can build a constraint for $F \circ G$. The negative constraint for $F \circ G$ is the union of the negative constraints for F and G. F and G's new methods must not be newly defined in the other mixin. The positive constraints for $F \circ G$ is the union of the positive constraints in F and G, where the constraint types in F take precedence over those in G. F's definitions must satisfy G's positive constraints. The distinction between new methods and redefinitions in OBSTACL's syntax (see section 6.1.2) enables us to generate the constraints for mixin composition.

Class	Definition of method m						
C	-	-	-	$m : \tau$	$m : \tau$	$m : \tau$	$m : \tau$
G	new $m' : \tau'$	new $m' : \tau'$	-	redef $m' : \tau'$	-	redef $m' : \tau'$	-
F	-	redef $m'' : \tau''$	new $m'' : \tau''$	-	redef $m'' : \tau''$	redef $m'' : \tau''$	-
Constraint	no m	no m	no m	$\tau <: m'$.next	$\tau <: m''$.next	$\tau <: m'$.next	(none)

Table 9.1: Mixin composition cases

9.2 Unnecessary Extensions

OBSTACL is a hybrid language supporting functional, imperative (procedural), and object-oriented programming. Many features in pure object-oriented languages are less compelling in a hybrid language because there are equivalent non-object constructs available. In this section we explain why OBSTACL does not contain certain features that may be expected in other languages.

9.2.1 Class Methods and Fields

Some information and functionality should be placed at the module level rather than at the object level. In a language with classes but no modules, they are placed in a class. Smalltalk and Java call them "class methods" and "class fields", while C++ uses the terms "static member functions" and "static member data". OBSTACL includes ML modules, so we can place data and functions in a module instead of a class. However, class methods typically have privileged access to objects in languages with class-based protection. OBSTACL uses object-based protection, in which only an object's methods can access the private fields of the object. A class method would have no special access to objects, so they offer no benefit over module level functions. However, instantiators might be considered a special form of class method, as in Smalltalk. Class-level protection can be simulated using opaque types (see section 9.3.3).

9.2.2 Typecase

With subtyping, an object's static type may be a supertype of its actual type. The ***typecase*** construct allows the programmer to recover the actual type of the object. In Java, the `instanceof` keyword allows the programmer to ask whether an object is an instance of a particular class, and casts allow the programmer to assert the relationship. In C++, the `dynamic_cast` mechanism serves as both a query and an assertion mechanism. Typecase is generally discouraged in object-oriented design because it decreases substitutivity: if a function states that it expects one type of object but then uses typecase to assert that it has a different kind of object, then a different kind of object cannot be used instead because the function makes additional assumptions that are not stated in the interface. Typecase is used in Java most commonly because the type system cannot express parametric polymorphism (see section 2.10). For example, a program must use a list of `Object` instead of a list of `Stream`, so typecase is needed to turn objects back into streams. C++ templates serve as a form of parametric polymorphism, and as a result the use of `dynamic_cast` is rare, except when C++ classes are being used as a substitute for "union" types. Since OBSTACL supports parametric polymorphism and union types directly, we do not expect typecase to be a needed feature.

9.2.3 Binary Methods

A binary method can be chosen as a result of looking at the classes of *two* objects instead of only one. Common examples of binary methods are set union, addition, and equality. Unfortunately, binary methods break substitutivity. Consider a binary method defined on a set class. If we write another set class with the same definition (see the thought experiment on page 21), it should be substitutable for one of the original sets. In the presence of binary methods, `a.union(b)` will do something different if `b` is an instance of the original set class than if it is an instance of the new set class, *even if the set classes are exactly the same*. Binary methods are

thought to be necessary to implement certain kinds of objects. In OBSTACL, these objects would be better represented as non-objects (such as instances of abstract data types), since they do not enjoy substitutivity and therefore do not gain much from object-oriented programming. In Java and C++, binary methods can be simulated using typecase (see section 9.2.2).[2] Binary methods are needed in pure object languages but when non-object alternatives are available, we prefer to use them rather than give up substitutivity.

9.2.4 Functional Update

As described in section 4.1.2, we chose to support imperative rather than functional update in OBSTACL. "Objects" that benefit from functional update tend to also be better represented as values. To support functional update in OBSTACL would require extensive changes, such as a re-examination of object invariants, the type system, the use of references for fields, the binding model for fields and self, and implementation.

9.2.5 Self Types

Some languages support the notion of self types, or MYTYPE, which is a name that can be used in an object type to refer to the type of the containing object. Unlike OBSTACL's recursive type names (see section 5.1.3), MYTYPE is specialized in subclasses to refer to the new object type. MYTYPE has three uses:

1. **As Argument Types for Methods**. For example, a method `C.equals(x:C object)` would continue to take only `C` objects if it were inherited into a subclass `D`. With MYTYPE, `C.equals(x:MyType)` would take `C` objects, but when inherited by `D`, would take only `D` objects. Unfortunately the type of `D` objects are no longer subtypes of the type of `C` objects, so it is not clear that `C.equals`, a method written assuming "self" is a `C` object, works when "self" is a `D` object. To support this use of MYTYPE, OBSTACL would need proper support for binary methods and a different way to type the "self" object.

2. **As Return Types for Methods**. For example, a method `C.clone()` would return a `C` object, but the same method in a subclass `D` should return a `D` object. In OBSTACL, this behavior is allowed if `D` redefines `clone`; the inherited `clone` method however would return a `C` object. How could a method be written to return a value of MyType?

   - If "self" has type MYTYPE, the method could return itself.
   - If an argument to the method has type MYTYPE, the method could return that argument.

[2] In Java, using typecase for binary methods is quite common, especially for the `equals` method. In C++, the use of binary methods is rare because classes can be used to define non-object-oriented data structures.

- If the language has a built-in clone operation, the method could clone an object with type MYTYPE and then return the clone.
- If the language supports functional update, the method could update an object with type MYTYPE and then return the new object.

Since OBSTACL does not include binary methods, built-in cloning, or functional update, the only object that can be returned is "self". Whenever we create objects without full type information, we need MYTYPE.

3. **For Convenience**. A common idiom in Smalltalk programs is for a method to perform a side effect and then return the self object so that further method calls can be chained. In C++, streams return themselves so that output or input operators can be chained together (*e.g.*, `cout << "hello" << name;` instead of `cout << "hello"; cout << name;`). This use of MYTYPE is not necessary for expressiveness. Any chain of calls `x.f().g().h()` can be replaced by `x.f(); x.g(); x.h()`.

OBSTACL does not need MYTYPE. Without binary methods, cloning, or functional update, the only use for MYTYPE is convenience. Given the difficulty of supporting MYTYPE in the type system, we could not justify supporting them.

9.2.6 Final Classes and Methods

Final classes in Java are classes that cannot have subclasses. The primary purpose is for security, to ensure that no subtypes are allowed. In addition, final classes may be more efficient than normal classes, since methods can be found statically. Dylan supports a similar feature, called *sealed classes*, which are used primarily for efficiency rather than to prevent subtyping. In OBSTACL, object types are structural rather than named, so preventing subclassing does not prevent subtyping. To prevent subtyping in OBSTACL would require the use of opaque types (see section 9.3.2). To prevent subclassing, a class can be defined in a module but not exported; only the instantiators need to be exported to allow the object creator to create instances. Without access to the class, subclasses cannot be defined.

Final methods in Java are methods that cannot be redefined in subclasses. Like final classes, the final methods are used both for security and efficiency. For example, in a stream hierarchy, if the `read` method is marked `final` in the File class, then subclasses of File may not redefine `read`. Dylan supports the *sealing* of methods, but the semantics are different from Java's final methods. Dylan supports multimethods (see section 3.2.4) so methods are not encapsulated with classes. When a method is sealed, it affects all definitions of the method, not only the definitions in subclasses. Continuing the stream example, if after the definition of a File class, the `read` method is sealed, then no subclass of `Stream` can redefine `read`. Java's `final` modifier is spatial in the sense that it affects classes based on their location in

the class hierarchy. Dylan's sealing is temporal in the sense that it affects classes based on when they were defined. OBSTACL supports neither form of preventing method redefinition. As with final classes, the security and efficiency advantages of final methods are incompatible with structural object types. Instead, module-level functions or abstract data types can often be used to guarantee that specific code is being executed.

9.2.7 Redefinable Fields

In OBSTACL, methods can be redefined but fields cannot. Although this appears to be an arbitrary rule, the rule actually is that public and protected items can be redefined but private items cannot. Since all methods are public or protected, and fields are private, we end up with redefinable methods and no redefinable fields. Given public fields (see section 9.1.2), fields can be overridden.

9.3 Programming Idioms

Within the language framework described in chapters 5 and 7, many features are possible. In section 9.2 we described extensions that we would not want in OBSTACL; in this section we describe constructs that fit into the system, and can either be extensions provided by the language or idioms used by the programmer.

9.3.1 Const Types

C++ supports the "const" qualifier on a type to restrict operations to those that do not modify the logical state of the object.[3] Const helps document interfaces and prevent certain errors. A type `const X` can be considered to be a supertype of `X`: an `X` can be used where a `const X` is required. There are two problems with the `const` qualifier in C++:

- **Preserving constness:** there are some functions for which we want both $\texttt{X} \to \texttt{X}$ and $\texttt{const X} \to \texttt{const X}$. The "solution" in C++ is to overload the function name for both `X` and `const X` arguments. We would instead write $\forall\tau <: \texttt{const X}.\tau \to \tau$.
- **Shallow vs. Deep constness:** the subobjects accessible through a `const X` are not `const` by default; to make them `const` requires overloading every accessor function.

In OBSTACL we can define a structural type `const_X` as a supertype of `X`. Any `X` is automatically a `const_X`. The programmer can define a spectrum of restricted versions of `X` to provide a more precise level of control over operations allowed by each client.

[3] The logical state is what would be part of the interface; the physical state is what would be part of the implementation. Splay trees demonstrate the difference: tree traversal preserves logical state but not physical state because it involves rotation of nodes.

9.3.2 Encoding behavior in types

ML does not associate behavior with types. There are no promises made in the *language* when you receive a value of some type. However, we can use modules and simple opaque types to create abstract data types. Similarly, we can use modules and opaque types to create promises about behavior for objects. A *contract* is a special marker represented by an opaque type. Programmers can use contracts to help associate behaviors with types:

- **Writing a contact.** A programmer creates a new opaque type, gives it a name, and writes documentation describing the contract. For example, there may be a comment "Declare a public method `promise` of type `IsDistanceMetric` if for any object d of type $\{|m$:Point$\times$Point $\rightarrow$ real, promise:IsDistanceMetric$|\}$, and any object x, y, z of type `Point`, we have $d.m(x,y) + d.m(y,z) >= d.m(x,z)$."
- **Signing a contract.** A programmer creates a distance metric class and includes an empty method `promise` of type `IsDistanceMetric` to declare that the terms of the contract are satisfied.
- **Requiring a contract.** A programmer creates code that only accepts with the contract in the type. In this example, a function takes an argument of type $\{|m : \text{Point} \times \text{Point} \rightarrow \text{real}, \text{promise} : \text{IsDistanceMetric}|\}$ even though it never invokes the `promise` method.

One may ask why this idiom is useful, when the terms of the contract are but a comment and therefore not compiler enforcible. The key benefit is in better static checking. If a programmer requiring a distance metric accepts objects of type $\{|m : \text{Point} \times \text{Point} \rightarrow \text{real}|\}$, which is all that is required by the type system, it is possible to get accidental subtyping (see section 4.1.4). If something goes wrong it is more clear who is at fault.[4] In this case, one programmer requires an object with a particular property. A second programmer has an object with the right type but does not know the implementation details, and therefore does not know whether the object has the needed property. There is a risk of an object without the needed property being passed to the function.

9.3.3 Class-Level Protection

OBSTACL uses object-level protection, in which only an object's methods have access to its private and protected data. Opaque types can be used to simulate class-level protection, in which a class can access data in any of its instances. Data which should be accessible to the class is marked *public* but is given a type known only to the module. Figure 9.15 shows an example of this technique. Since we are simulating accessibility-based protection, we expect to see the same disadvantage (it is not possible to create a properly functioning object with

[4] See also the Design by Contract discussions in [Mey94].

```
struct X : Xsig = structure              sig Xsig = signature
 type hidden = {x:int, y:int}             type hidden;
 class Xclass
   ...
   public field h:hidden;
   method m() = h.x;   (* access *)
   constructor make(x',y') (* init *)
     fields {h={x=x',y=y'}}
   instantiator make;
 end

 (* export these from the class *)
 type obj = Xclass object;                type obj={|...,h:hidden,...|}
 val make = Xclass.make;                  val make: int*int->obj;

 fun add(a:obj,b:obj):obj =               val add:obj*obj->obj;
   make(a.h.x+b.h.x, a.h.y+b.h.y);
   (* access to private data of two objects *)
end
```

Figure 9.15: Class level protection using private types

the same type but unrelated implementation because the object would not have a field of type `X.hidden`) and advantage (it is possible to write functions such as `add` that operate on hidden data of two objects).

However, it is still possible to write an improperly functioning object with the correct interface. Although the definition of `hidden` is not known outside of module `X`, it is possible to declare variables of this type, put values of this type into data structures such as lists, and so on.[5] To create a value of type `hidden` one may first create an `X` object, then extract its `h` field. Accessibility-based protection cannot be simulated exactly in OBSTACL; a determined user can create an object of a class other than `X` and pass it to the `X.add` function. For more restrictive abstractions, abstract data types (see section 3.2.1) are available.

Barring abuse by users, the type with implementation-specific data (an "implementation type") restricts one to using objects of a particular class. In some contexts we would also like to use objects of other classes with a similar interface. With structural subtyping, the type *with* the `X.hidden` field is a subtype of the type *without* the `X.hidden` field. As in [Fis96], implementation types are subtypes of pure interface types.

[5] ML allows one to manipulate values of opaque types, as long as the representation of the value are not accessed.

9.3.4 Changing Classes

Some dynamic languages (like Self and LambdaMOO) allow an object to change its class (or parent object) at any time. In contrast, most object-oriented languages tie an object to a class at instantiation time. Between these two is an approach called predicate classes [Cha93], in which an object cannot change its class arbitrarily but can switch among a fixed set of "states". Changing an object's class can be useful to represent different "phases" an object goes through. For example, a file object may be "open" or "closed". A conventional implementation of such an object would involve a flag indicating whether the file was open or closed, along with methods that check the flag and behave in one of two different ways.

Using class-changing, files could instead be an instance of an `OpenFile` class or a `ClosedFile` class. The `open()` and `close()` methods simply change the class of the object. The advantage of this approach is that methods do not check a flag; instead, `OpenFile.read` can assume the file is open and `ClosedFile.read` can be omitted, leading to a `message not understood` error if the client attempts to read from a closed file.

To work in a statically typed language, the classes representing the phases of the object must have the same type, so that the object's type does not change when it transitions into a different phase. Even with this restriction, it is not clear how the change-class operation should be implemented, especially in the presence of subclassing. A subclass `OpenEncryptedFile` will inherit a `close` method from the superclass `OpenFile`. This method will change the object's class to `ClosedFile`, which has an `open` method that changes the object's class to `OpenFile`. To preserve the encryption, there must be an `ClosedEncryptedFile` class and the `open` and `close` methods must be redefined. On the other hand, perhaps the new file is compressed rather than encrypted, and `OpenCompressedFile` should be used. The language implementation cannot always deal automatically with both switching an object's class and subclassing; the programmer must choose the policy.

If the programmer writes the policy, `ClosedFile`'s `open` method may not be the ideal place for its implementation, since it is difficult to change the policy.[6] A solution that scales better in the presence of subclassing is the Envelope/Letter pattern [Cop92]. A phase shifting object is split into two object: an ***envelope***, which encapsulates the phase transition logic, and a ***letter***, which represents the phase. In this design, the switching policy resides in the envelope, so a different policy can be used by creating a different envelope class—none of the letter classes have to be subclassed. For objects with multiple phases, the Envelope/Letter pattern provides a more flexible solution than changing classes.

[6] Changing the policy would require creating a subclass of `ClosedFile` with a new `open` method. It would also require new versions of `OpenFile`, `OpenEncryptedFile`, `OpenCompressedFile`, etc., with a new `close` method that uses the new subclass of `ClosedFile`. Thus a new policy requires subclassing every class involved, making it an undesirable solution.

The Envelope/Letter pattern uses an envelope to change the letter (phase) and also to forward messages to the letter. The envelope therefore should support the full interface of the object. For example, for files it should support `open`, `close`, `read`, `write`. The letters however need support only those methods that do not change the object's phase: `read` and `write` in this example. The envelope can be implemented as a subclass of a forwarder (see section 6.3.2) to letters. Since OBSTACL uses interface types, object users need not be aware that an object is implemented with envelopes and letters. A file object implemented with this pattern can be used interchangeably with a file object implemented in a different way. The envelope/letter pattern is a reasonable and flexible alternative to changing an object's class, and does not require additional language support from OBSTACL.

9.4 Alternative Designs

There are many decisions we made that could have been made differently without substantially altering the language. In this section we explore some of those decisions.

9.4.1 Mixin Constructors

One oddity in the language is that a mixin must include in its constraint not only methods but the constructors, so that the mixin constructor can call the superclass constructor. An alternative would be for mixins to define a "delta constructor", which is mixed into the superclass constructor. A delta constructor taking type τ and a superclass constructor taking type σ would be merged into a subclass constructor taking the pair $\tau \times \sigma$. Under this scheme, the mixin constraint does not have to list the constructors at all.

9.4.2 Single Constructor

Given that we have to list constructors in the mixin constraint, there may be times that a mixin would applicable if only the constructor had the proper name. Different names may be used for style reasons and not for meaning (*e.g.*, `make_object` vs. `makeObject`). An alternative would be to not name constructors, and have only one. Multiple constructors can be simulated using union types. The user would still be able to use multiple instantiators. A mixin then would contain a constructor that takes a pair (a, b), uses b to initialize its own fields, and passes a to the superclass. The type of a is generic (α) so the mixin can be applied to a superclass without knowing what type the superclass constructor accepts.

9.4.3 Classes are Functions

An alternative to treating classes as extensions of other classes is to treat classes as functions producing objects. Extension is then represented as the class "calling" another class and

```
class (x) = extend B(x) with
   field y=3
   method m() = x+y
end
```

Figure 9.16: Classes as functions

extending the returned object (see figure 9.16). However, the semantics and implementation would continue to be class extension, not object extension. It is not clear how to represent protection levels and instantiators with this syntax in a way that makes sense for the programmer. Sometimes a new syntax is to be preferred over a familiar syntax with slightly different semantics.

9.4.4 User Level Instantiators

Instantiators are rare in object-oriented languages; why do we need them in OBSTACL? Instantiators are the only functions allowed to create instances of the class, but this ability could be exported to the containing module without compromising the ability to encapsulate the construction process. The module can export an ordinary function that serves the same purpose as the instantiator. However, in OBSTACL the instantiator serves a second purpose. Constructors produce objects in which the protected and public methods are visible; instantiators hide the protected methods so that the object user sees only public methods. Demoting instantiators to module level functions is feasible as long as we have an alternative for dealing with protected methods.[7]

9.4.5 Inherited Instantiators

Subclasses do not inherit instantiators from their parents. To inherit instantiators would require a limited form of "Self" types. An instantiator must be written to return subtypes of its own class.

```
[ForAll t <: {|...|}] instantiator mk(x):t
```

Such instantiators are not only more difficult to reason about, but they also restrict some uses of instantiators. For example, we cannot store objects of the self type in an object cache.

If code sharing is needed between instantiators, one can pass the raw instantiation code (a lambda function invoking the `new` operation) to a function that handles instantiation.

[7] Asking the author of the subclass to deal with protected methods of superclasses is not desirable because it introduces a dependency. If the superclass adds a public or protected method that does not affect the subclass, the subclass should not have to change. This is especially true for mixins, which may have superclasses with additional methods not listed in the mixin constraint.

9.4.6 Explicit Interface Hierarchy

It may be easier to deal with object types if each was given a name along with associated behavior. Classes would explicitly state that they implemented some particular object type, as in Java. Recursive object types would be easier with names. Type inference may be easier, and explicit types are less cumbersome. It may be difficult to reconcile named object types with mixins, because mixins create new class types by adding and replacing components in existing class types. From these new class types one gets new unnamed object types.

An explicit type hierarchy may result in slightly more efficient code. It is possible to support efficient method lookup by introducing a separate hierarchy of mixin interfaces similar to the one analyzed by Flatt et al. [FKF98] and requiring that the order of methods in a mixin's dictionary match that given in the interface implemented by the mixin. However, a separate interface hierarchy would make the language and calculus significantly more complicated.

There are still good reasons to use structural types (see section 4.1.4). In addition to the flexibility they offer, in OBSTACL the *protected* visibility specifier is expressed using structural types. Using explicit types instead of structural types will require some reworking of mixins, and the addition of new rules to handle protected visibility.

9.4.7 Explicit Subsumption

Subsumption in OBSTACL is implicit: a function expecting a Stream object can be passed a File Stream object. Implicit subsumption is not a requirement for an object-oriented language. An alternative is to require an explicit conversion from a subtype to a supertype. Evidence from OCAML and also from design patterns suggests that subsumption is not a common operation. Objects tend to be constructed with a specific type, then placed in a data structure requiring a more general type. Further operations are performed with the general type. Since subsumption tends to be uncommon, it may be reasonable to require explicit subsumption by the programmer.

9.4.8 Accessibility Based Protection

With visibility-based protection, an object can pass from a privileged function to an unprivileged function. However, the object cannot pass the other way because in the unprivileged function, the longer type (with privileged access functions to the implementation) is "forgotten". With accessibility-based protection, this isn't a problem. However, widespread use of types describing implementation details (even though these details are marked inaccessible) reduces the opportunities for reuse through substitutivity.

```
struct X
   type memento;
   val make: 'a -> {| ..., destructure:unit->memento,
                      restore:memento->unit, ... |}
end;
```

Figure 9.17: Opaque type used for destructuring

9.4.9 Destructuring

We may want to provide a way to de-structure an object into its state and to create an object from the state. A programmer can write these operations explicitly, as shown in figure 9.17, but the code is often repetitive and error-prone. Destructuring allows for cloning (a de-structure immediately followed by a recreation), network pickling (a de-structure, transmission of the state across a network, and a recreation on the other side), object persistence (a de-structure, saving of the state to disk, and later recreation from state read from the disk), and the Memento Pattern (a de-structure, storage of the state to a token, and a restoration of the object from the token). Destructuring is a more generally useful operation than cloning, and can be used to provide cloning as well.

Chapter 10

Conclusions

> Object-oriented programming is an exceptionally bad idea which could only have originated in California.
>
> *–E. Dijkstra*

Object-oriented programming techniques are not simply hype. Objects offer advantages in program design and maintenance. However objects are not appropriate for every design problem. OBSTACL is a language designed to allow and even encourage multiple kinds of abstractions to be used together. To focus on objects, we adopted ML, a rich language without objects, to provide the basics of a language (variables, expressions, branching, loops, functions, and so on) as well as non-object abstractions (such as integers, strings, lists, tuples, and records). There is one fundamental goal for OBSTACL objects: it should be possible to replace an object by another, with the changes to the program arising only from the difference in behavior of those objects as exposed through the public interface. In particular, if the new object is implemented differently but behaves the same, the program should behave the same. This requirement is not met by languages like C++, Java, and Eiffel. There is one fundamental goal for OBSTACL classes, which are used to define families of objects: a class should be able to define fields and methods which cannot be seen by extensions of the class or users of objects instantiated from the class, so that the class author may change these implementation details while preserving the correctness of existing programs. These two goals restrict the features that can be in the language. For example, binary methods and class-based protection are incompatible with substitutivity for objects, so OBSTACL does not offer these features. Given the constraints imposed by the goals of OBSTACL, some things we thought would be easy turned out to be hard (object construction, for instance), and some things we thought would be hard turned out to be non-problems (deep equality, for instance).

10.1 Objects vs. non-objects

OBSTACL supports but does not mandate the use of objects. OBSTACL's objects are designed to complement, not replace, non-object abstractions. Objects occupy a different place in the tradeoff between guarantees and extensibility of data and functions. Object types support a different sort of polymorphism than non-object types. Objects behave differently than non-objects for equality, assignment, copying, and mutation. We consider these differences important in *simplifying* the language. In each of these cases, two simple forms of abstraction are simpler than one complex form that encompasses properties of both.

- Variance in data. The non-object abstractions with data variants define those variants in a central location, then allow functions to examine the variants. The set of functions is extensible but the set of data variants is not. Objects provide instead the extensibility of the set of data variants while defining the set of functions in a central location. Extensibility in two dimensions at once is provided by objects with multimethods, but the combination of two or more variants is necessarily incomplete (see section 3.2.4).

  OBSTACL provides objects with extensibility of data variants and ML unions with extensibility of functions. During initial program design, the extensibility provided by unions, objects, and multimethods is beneficial. During subsequent maintenance, a programmer needs the ability to add and remove libraries easily. Unions and objects continue to work well, while multimethods typically do not.

- Polymorphic types. Non-object abstractions with polymorphism provided parametric polymorphism, which allows the definition of type structures with placeholders that allow any type (or any type with some restriction). Objects provide subtype polymorphism, which allows the definition of type structures that allow extension. In OBSTACL these forms of polymorphism can be combined cleanly and are orthogonal.

- Values vs. objects. All abstractions can be considered to represent objects. However those objects are stored in a computer in two distinct ways. Objects like files and bank accounts are represented directly in the computer's memory. Objects like the number three or the set of prime numbers less than ten are represented differently—their *names* (or descriptions) are stored in memory. We call these latter form *values*. There may be many memory locations that store the value 3. These all refer to the same abstract number three. Multiple descriptions in memory, such as $\{2, 3, 5, 7\}$ and $\{5, 3, 7, 2\}$, may refer to the same underlying object, and are therefore equal. Objects on the other hand are typically represented in memory just once.

  Values are references to abstract objects that may not themselves be represented on the computer. Several locations in memory may contain the same value. The consequences

of this difference between values and objects involve identity, comparison, copying, and mutation. Values have no identity of their own; only the underlying abstract objects do. Objects have identity—each object created by the `new` operator is brand new and distinct from all other objects. Comparing values for equality is asking whether their underlying objects are the same; thus we need deep equality on the structures describing those underlying objects. Comparing objects in contrast involves shallow equality. Copying a value is a common operation—a new location in memory now contains a value that refers to the same object. Copying an object is not the same kind of operation—a new location in memory does not compare equal to the old.[1] Mutating a value means changing it to refer to a different object. The new value is not equal to what it was before the mutation. Mutating an object preserves its identity but changes its state. The object is equal to the object before mutation.

Values and objects behave differently under the fundamental operations of comparison, copying, and mutation. Some languages (like C++) attempt to unify them into one complicated language construct.[2] Other languages (like ML and Java[3]) provide only one of the two. OBSTACL directly supports both values and objects, keeping the semantics of both simple and consistent.

10.2 Object Definition

Objects are a run-time structure combining data and functions operating on that data. The functions and data definitions can be shared between objects by enclosing them in a class, which serves as a blueprint for creating objects. OBSTACL uses classes over prototype objects as a mechanism for shared definitions because classes and objects are different enough that it is simpler to have two straightforward constructs rather than one that handles everything. Objects support substitutivity, enabling interchanging of objects with the same interface but different implementations. Object types support subtyping, where objects of one type can be used in contexts requiring objects with fewer operations. OBSTACL uses structural subtyping, which is based on the structure of a type instead of the name of the type or the class of the object. Structural subtyping is more flexible than name-based subtyping, and produces a much

[1] Instead, objects may provide cloning, which creates an object with a new identity but the same state as the original object.

[2] Value types in C++ are typically used with pass by value or pass by reference, support `operator =`, support `operator ==`, have a copy constructor, and use deep equality. Object types in C++ are typically used with pass by pointer, inheritance, don't have an `operator =`, don't have an `operator ==`, and use shallow equality (comparison of pointers).

[3] In Java, a few built in types (`boolean`, `char`, `int`, etc.) have value semantics while all other types have object semantics. For `int`, `x=y` is used for copying, `x==y` is used for equality, and `x=3` is used for assignment. For object classes like `Integer`, `x=y.clone()` is used for copying, `x.equals(y)` is used for equality, and `x.set(3)` is used for assignment. (Note however that `Integer` produces immutable objects and thus does not have the `clone` or `set` methods.) The use of `clone` and `equals` is a sign of a value type forced into object semantics (where = and == have different meaning).

richer set of relationships without explicitly being listed by the programmer. Complementing subtyping is class extension, which allows a class to be defined in terms of its differences (new fields, new methods, and redefined methods) from another class. What distinguishes inheritance from sharing between modules is the distributed *recursive* nature of a class. Functions in a base module call other functions in the base module, even when using a derived module. In contrast, methods in a base class can call methods in the derived class when using an object of the derived class. Dynamic lookup of methods allows reuse relationships between classes not possible with modules. In OBSTACL, substitutivity provides external reuse, the reuse of code that uses objects, while inheritance provides internal reuse, the reuse of code defining objects.

Just as common parts of code can be abstracted into functions, common parts of classes can be abstracted into linear mixins. If A and A' are related by inheritance in the same way as B and B', a mixin M can be created such that $A' = M(A)$ and $B' = M(B)$. An alternate form of abstraction over classes is multiple inheritance, with which the shared definitions are placed into class C, and A' extends A and C, and B' extends B and C. OBSTACL provides mixins but not multiple inheritance. Multiple inheritance offers a less structured but more flexible approach to building classes. Uses of multiple inheritance can be replaced by a combination of mixins and structural subtyping. Mixins form an asymmetric (parent/child) relationship with the classes they extend, allowing the programmer to control the order and number of (including repeated) applications. OBSTACL offers classes to build sets of objects and mixins to build sets of classes.

Defining an object's contents is easy compared to the process of building an object. Since the components of a class are defined in multiple modules, the components of an object produced from that class are initialized by the corresponding modules. Modular initialization is more complex than centralized initialization but is desired for maintainability. In the presence of mixins, which do not have access to the implementation details of the parent class, modular initialization becomes necessary. In OBSTACL, each class provides one or more constructors to initialize objects. A constructor initializes fields, calls a constructor of the parent class, and sets up any invariants maintained by the object. Just as methods encapsulate access to fields, in OBSTACL instantiators encapsulate access to constructors. Instantiators choose whether and when to create objects, and invoke constructors to initialize new objects. The instantiators and constructors ensure both type safety and proper initialization of objects as defined by the class.

10.3 Design and Maintenance

Programs are divided into modules to make them easier to work with. The program accesses a module through its interface, which is expected to change infrequently relative to the module's implementation. Determining what is expected to change and what is expected to remain

relatively constant is an important aspect of program design. Also important is determining the granularity at which module boundaries lie. Smaller modules are easier to understand, replace, and reuse. However, fine grained module division generally leads to more modules, with more dependencies between them. Abstractions reduce the number of modules by replacing several specialized program components with a more general one. OBSTACL provides substitutivity, subtyping, mixins, and inheritance to aid in the process of abstraction.

The programmer's goal is to choose abstractions that minimize the need for interface changes and maximize the possibility of implementation changes that do not affect the rest of the program. Dependencies between program components trigger changes to one when another is changed. Therefore minimizing dependencies is a goal of program design. Three basic forms of dependencies exist in object-oriented programming. The user of an object depends on the type of the object. OBSTACL uses interface types that do not mention the class from which the object was instantiated, so that the user of the object does *not* depend on the class. The user creating an object depends on the class and its construction policy. OBSTACL provides instantiators so that the policy is encapsulated within the class and not seen by the user. A subclass depends on the class it extends and what that superclass provides. OBSTACL provides mixins, so that the subclass can depend on what is provided by the superclass but not on the superclass itself.

In addition OBSTACL provides visibility control to specify who can see (and therefore may depend on) each component of an object. Private items are seen only by the class defining them. Protected items are seen by the class and its extensions. Public items are seen by the class, its extensions, and users of the object. Visibility control provides stronger hiding than access control (used in C++ and Java), which allows private names to be seen (and cause name conflicts) even though they cannot be used. OBSTACL constructors allow classes to define initialization and object invariants without causing subclasses to depend on them. OBSTACL instantiators allow classes to define creation policies without causing users to depend on them. Dependencies are also respected at compile time—an unmodified module is recompiled only if it depends on something that needs recompilation. In OBSTACL the class author has the ability to specify interfaces precisely and therefore determine how other parts of the program may depend on that class.

Programmers make decisions based on costs and benefits. Costs can include the time to learn a new language or language feature, development time, compilation time, execution time and space, convenience, and monetary cost. Maintenance costs are often underestimated compared to costs during initial design and implementation. At times a "good" design carries a higher initial cost. In OBSTACL several of these costs are lowered, so that the programmer does not have to trade maintainability for efficiency or ease of initial implementation. Private fields in OBSTACL are faster than public fields, and public accessor functions are just as fast

as public fields. This cost structure encourages the use of private fields and accessor functions (considered good design) instead of public fields (considered bad design). The features of OBSTACL are fairly simple and orthogonal so that the programmer can use some without learning them all. Recompilation directly matches the programmer's expectations—changes to private implementation details do not trigger recompilation, lowering the barrier to improving the implementation of a class. Interface types and instantiators are used by default, and the less flexible alternatives take more work, not less, to use. Virtual functions are fairly efficient, and are used only where substitutivity is offered, so that the cost is not paid when there is no possibility of benefit. In the proposed implementation there are no global optimizations turning objects into non-objects when there is only one data variant. Such optimizations are less applicable in a language where abstractions with only one variant are expressed as ADTs, not objects. These optimizations also create a disincentive to create more data variants, because the second variant imposes a cost disproportionate to the features it uses. The costs of OBSTACL features are locally predictable by the programmer and do not depend on the global state of the system.

At times there is a tradeoff between ease of initial implementation and ease of maintenance. Supporting maintenance is the motivator behind many OBSTACL design decisions: object-based protection, visibility control instead of access control, no binary methods or multimethods, shallow equality, superclass method access restricted to immediate ancestor, no subtyping on classes, no partial inheritance. These decisions may seem restrictive but actually avoid dependencies that lead to maintenance problems. Another set of restrictions (no class fields or methods, no const types, no behavioral or implementation types) are not problematic, yet not supported directly by OBSTACL. Instead these are considered idioms that can be implemented by the programmer using existing features (module variables, structured subtyping, or opaque types, respectively). Python relies on programmer enforcement of access control; C++ relies on the programmer to provide instantiators. Yet OBSTACL directly supports these, along with other features that could be idioms. To balance language simplicity and maintenance needs, OBSTACL directly supports those features that involve cooperation across modules (between programmers) and relies on the programmer when the idiom can be implemented and enforced locally within a module.

10.4 Theory and Practice

The semantics of OBSTACL are explained more precisely by a calculus than by prose. The core calculus for OBSTACL focuses on objects, classes, and mixins, and not on the basic features inherited from ML, which have been explored in other calculi. It uses standard constructs—variables, records, functions, assignable locations, and scoping—to represent objects. Object types are record types with standard subtyping. The new constructs are classes, mixins, class

types, and mixin constraints. The core calculus directly represents these parts of OBSTACL, as well as the modular construction system and visibility control. It omits several straightforward constructs: instantiators, naming of constructors, and multiple private fields per class. A full calculus for OBSTACL would include these plus all of ML's features, such as parametric polymorphism and a module system.

An implementation of OBSTACL objects is straightforward and would be similiar to an implementation of a conventional object-oriented language like Java. Classes and mixins however are trickier to implement efficiently. The main challenge is incomplete information at compile time, both because of run-time class inheritance and the desire to compile mixins only once instead of once per application. The run-time structures proposed for OBSTACL classes preserve both separate compilation and efficiency similar to that of languages with only compile-time inheritance.

10.5 Future Work

In addition to the extensions described in chapter 9, there are several open areas of research regarding OBSTACL. The choice of structural types in preference to named types occasionally leads to accidental subtyping relationships not expected by the programmer. Behavioral types are possible but take extra work by the programmer. A more flexible and convenient, yet error prone, construct is contrary to the general philosophy behind other design choices made for the language. However our formulation of mixins in the core calculus requires structural types. Studying mixins with named types would thus be of interest.

The scope of this work has been to use only generic features of ML. In particular, the object type system does not rely on parametric or bounded polymorphism. If OBSTACL were to be fully implemented as an extension of ML (or another language), a study of the interactions between OBSTACL's constructs and ML's (or another language's) unique features would be needed. For ML, the three primary areas to study are parametric polymorphism (genericity), the module system, and type inference.

10.6 Summary

OBSTACL is a language designed to support good program design and maintenance. To minimize dependencies between modules and also to allow abstraction of common components, we added objects, structural subtyping, classes, run-time inheritance, mixins, and mixin constraints to a foundational language, ML. OBSTACL's features work well together to express common problems in program design and can be analyzed by a simple calculus and implemented efficiently.

Appendix A

Calculus Rules and Definitions

A.1 Definition of Contexts

The definition of contexts is standard but lengthy because of the number of subexpressions in the mixin expression:

$$
\begin{array}{rcl}
C ::= & [\,] \mid C\ e \mid e\ C \mid \lambda x.C \mid C.x \mid C \diamond e \mid e \diamond C \\
\mid & \{m_1 = e_1, \ldots, m_{i-1} = e_{i-1}, m_i = C, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n} \\
\mid & \mathsf{H}\ h.C \mid \mathsf{H}\langle x, C\rangle h.e \mid \mathsf{new}\ C \mid \mathsf{classval}\langle C, \mathcal{M}, \mathcal{P}\rangle
\end{array}
$$

$$
\begin{array}{l|l|l}
\mathsf{mixin}^{\substack{j \in New \\ k \in Redef \\ \ell \in Prot}} & \mathsf{mixin}^{\substack{j \in New \setminus [i] \\ k \in Redef \\ \ell \in Prot}} & \mathsf{mixin}^{\substack{j \in New \\ k \in Redef \setminus [i] \\ \ell \in Prot}} \\
\quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; \\
\quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{method}\ m_i = C; & \quad \mathsf{redefine}\ m_k = v_{m_k}; \\
\quad \mathsf{protect}\ [p_\ell]; & \quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{redefine}\ m_i = C; \\
\quad \mathsf{constructor}\ C; & \quad \mathsf{protect}\ [p_\ell]; & \quad \mathsf{protect}\ [p_\ell]; \\
\quad \mathsf{end} & \quad \mathsf{constructor}\ v_c; & \quad \mathsf{constructor}\ v_c; \\
 & \quad \mathsf{end} & \quad \mathsf{end}
\end{array}
$$

A.2 Type Rules

The type rules for class-related forms were presented in section 7.4. The remaining type rules are presented here.

A.2.1 Subtyping Rules

The subtyping rules are standard. Objects support both depth and width subtyping.

$$\frac{}{\Gamma, \iota_1 <: \iota_2 \vdash \iota_1 <: \iota_2} \ (<: \mathit{proj}) \qquad \frac{}{\Gamma \vdash \tau <: \tau} \ (\mathit{refl})$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \ (\mathit{trans}) \qquad \frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash \tau \to \sigma <: \tau' \to \sigma'} \ (\mathit{arrow})$$

$$\frac{\Gamma \vdash \tau_i <: \sigma_i \ \ i \in I \quad J \subseteq I}{\Gamma \vdash \{m_i : \tau_i\}^{i \in I} <: \{m_j : \sigma_j\}^{j \in J}} \ (<: \mathit{record})$$

A.2.2 Type Rules for Expressions

The type rules for expressions other than class-related forms are simple, except for heaps, which have to be typed globally.

$$\frac{\mathit{typeof}(\mathit{const}) = \tau}{\Gamma \vdash \mathit{const} : \tau} \ (\mathit{const}) \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \ (\mathit{proj}) \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \to \sigma} \ (\lambda)$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \ e_2 : \sigma} \ (\mathit{app}) \qquad \frac{}{\Gamma \vdash \mathit{fix} : (\sigma \to \sigma) \to \sigma} \ (\mathit{fix})$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau <: \sigma}{\Gamma \vdash e : \sigma} \ (\mathit{sub}) \qquad \frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{x_i = e_i\}^{i \in I} : \{x_i : \tau_i\}} \ (\mathit{record})$$

$$\frac{\Gamma \vdash e : \{x : \sigma\}}{\Gamma \vdash e.x : \sigma} \ (\mathit{lookup}) \qquad \frac{}{\Gamma \vdash \mathsf{ref} : \tau \to \tau \ \mathsf{ref}} \ (\mathit{ref}) \qquad \frac{}{\Gamma \vdash \, ! : \tau \ \mathsf{ref} \to \tau} \ (!)$$

$$\frac{}{\Gamma \vdash \, := \ : \tau \ \mathsf{ref} \to \tau \to \tau} \ (:=)$$

$$\frac{\Gamma' = \Gamma, x_1 : \tau_1 \ \mathsf{ref}, \ldots, x_n : \tau_n \ \mathsf{ref} \quad \Gamma' \vdash v_i : \tau_i \quad \Gamma' \vdash e : \tau}{\Gamma \vdash H\langle x_1, v_1 \rangle \ldots \langle x_n, v_n \rangle.e : \tau} \ (\mathit{heap})$$

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[AM91] A. Appel and D. MacQueen. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13. Springer-Verlag, August 1991.

[App98] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[Aug98] L. Augustsson. Cayenne—a language with dependent types. In *Proc. ICFP '98*, pages 239–250, 1998.

[Aus98] M. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[BB98] V. Bono and M. Bugliesi. Matching for the lambda calculus of objects. *Theoretical Computer Science*, 1998.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.

[BCC+95] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BCH+96] K. Barrett, B. Cassels, P. Haahr, et al. A monotonic superclass linearization for Dylan. In *Proc. OOPSLA '96*, 1996.

[BF98] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *Proc. ECOOP '98*, pages 462–497. LNCS 1445, Springer-Verlag, 1998. Preliminary version appeared in FOOL 5 proceedings.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proc. OOPSLA '86*, pages 78–86, 1986.

[BL92] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages (ICCL '92)*, pages 282–290, April 1992.

[BL95] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In *Proc. CSL '94*, pages 16–30. LNCS 933, Springer-Verlag, 1995.

[BLM97] J. Bank, B. Liskov, and A. Myers. Parameterized types and Java. In *Proc. POPL '97*, 1997.

[BLS94] N. Boyen, C. Lucas, and P. Steyaert. Generalized mixin-based inheritance to support multiple inheritance. Technical Report vub-prog-tr-94-12, Vrije Universiteit Brussel, 1994.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications. (second edition)*. Benjamin/Cummings, 1994.

[BPS99] V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, New York, NY, June 1999.

[BPSM99] V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of object, classes, and mixins. Technical Report, The University of Birmingham and Stanford University, 1999. Forthcoming.

[Bra92] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[BRS+98] D. Bäumer, D. Riehle, W. Siberski, C. Lilienthal, D. Megert, K. Sylla, and H. Züllighoven. Values in object systems. Technical Report TR-1998-10-1, Ubilab, 1998.

[Bru94] K. B. Bruce. A paradigmatic object-oriented language: Design, static typing and semantics. *J. Functional Programming*, 4(2):127–206, 1994.

[BSvG95] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952, pages 26–51. Springer, 1995.

[Car93] T. A. Cargill. The case against multiple inheritance in C++. In Jim Waldo, editor, *The Evolution of C++: Language Design in the Marketplace of Ideas*, pages 101–109, Berkeley, CA, USA and Cambridge, MA, USA, 1993. USENIX and MIT Press.

[CDG97] C. Chambers, J. Dean, and D. Grove. Whole-program optimization of object-oriented languages. Technical Report TR-96-06-02, University of Washington, January 28, 1997.

[CF91] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. POPL '91*, pages 233–244, 1991.

[Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *Proc. ECOOP '92*, 1992.

[Cha93] C. Chambers. Predicate classes. In *Proc. ECOOP '93*, 1993.

[Cha99] D. Chase. Garbage collection FAQ, 1999.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. POPL '90*, pages 125–135, 1990.

[Com92] Apple Computer. *The Dylan Reference Manual*. Apple Computer, 1992.

[Coo89] W. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[Coo92] W. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proc. OOPSLA '92*, 1992.

[Cop92] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[Cop99] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, Reading, Mass., 1999.

[Cur97] P. Curtis. LambdaMOO programmer's manual, 1997.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proc. OOPSLA '96.*, pages 83–100. ACM Press, 1996.

[DMN70] O-J. Dahl, B. Myrhaug, and K. Nygaard. SIMULA common base language. Technical report, Norwegian Computing Center S-22, 1970.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[FF98] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ICFP '98*, pages 94–104, 1998.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994. Preliminary version appeared in Proc. *LICS '93*, pp. 26–38.

[Fis96] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford, CA , USA, 1996.

[FKF98] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.

[Fla98] D. Flanagan. *JavaScript*. O'Reilly & Associates, 1998.

[FM95] K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, pages 42–61. LNCS 965, Springer-Verlag, 1995.

[FM98] K. Fisher and J. C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–26, 1998. Preliminary version appeared in Marktoberdorf '97 proceedings.

[Ful96] J. Fulton. Extension classes, Python extension types become classes, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[Gos97] J. Gosling. The evolution of numerical computing in Java, 1997.

[HM95] M. Hoang and J. Mitchell. Lower bounds on type inference with subtypes. In *Proc. POPL '95*, 1995.

[Inc94] Taligent Inc. *Taligent's Guide to Designing Programs: Well Mannered Object-Oriented Design in C++*. Addison-Wesley, 1994.

[Ing78] D. Ingalls. The Smalltalk-76 programming system design and implementation. In *Proc. POPL '78*, 1978.

[Int94] Borland International. A technical comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5. Technical report, Borland International, Inc., 1994.

[Kee89] S. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[KR94] S. Kamin and U. Reddy. Two semantic models of object-oriented languages. In C. Gunther and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[LCI+92] M. Linton, P. Calder, J. Interrante, S. Tang, and J. Vlissides. InterViews reference manual, 1992.

[LM96] M. Van Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[LRVD99] X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. The Objective Caml system, documentation and user's guide, 1999.

[Lut96] M. Lutz. *Programming Python*. O'Reilly and Associates, 1996.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Java series. Sun Microsystems, 1996.

[Mey87] B. Meyer. Eiffel: Programming for reusability and extendability. *SIGPLAN Notices*, 22(2), February 1987.

[Mey92] S. Meyers. *Effective C++: 50 Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.

[Mey94] B. Meyer. *Reusable Software*. Prentice-Hall, 1994.

[MMPN93] O. Lehrmann Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Language*. Addison-Wesley, 1993.

[Moo86] D. Moon. Object-oriented programming with Flavors. In *Proc. OOPSLA '86*, pages 1–8, 1986.

[MT89] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proc. ICALP '89*, pages 574–588. LNCS 372, Springer-Verlag, 1989.

[MTHM90] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[OSF91] OSF, editor. *OSF/Motif Programmers Guide*. Prentice Hall, Englewood Cliffs, 5 edition, 1991.

[Ous94] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. POPL '97*, 1997.

[Pet94] C. Petzold. *OS/2 Presentation Manager Programming*. Ziff-Davis Press, 1994.

[Pit93] K. Pitman. What's in a name? *Lisp Pointers*, VI(1), 1993.

[Pot98] F. Pottier. A framework for type inference with subtyping. In *Proc. ICFP '98*, pages 228–238, 1998.

[SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP '98*, pages 550–570, 1998.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. OOPSLA '86*, 1986.

[Ste90] G. Steele. *Common Lisp: the Language (second edition)*. Digital Press, 1990.

[Str] J. Strout. POO programmer's reference.

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str97] B. Stroustrup. *The C++ Programming Language (3rd ed.)*. Addison-Wesley, 1997.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. PLDI '96*, pages 181–192, 1996.

[TT94] L. Thorup and M. Tofte. Object oriented programming and Standard ML. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 1994.

[US87] D. Ungar and R. Smith. Self: The power of simplicity. In *Proc. OOPSLA '87*, 1987.

[VN96] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. OOPSLA '96*, pages 359–369, 1996.

[vR99] G. van Rossum. Metaclasses in Python 1.5, 1999.

[VRTB98] J. Viega, P. Reynolds, B. Tutt, and R. Behrends. Automated delegation is a viable alternative to multiple inheritance for class based languages. Technical Report Computer Science tech report CS-98-03, University of Virginia, 1998.

[Wad87] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, Munich, Germany, January 1987.

[Wan94] M. Wand. Type inference for objects with instance variables and inheritance. In C. Gunther and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[WF94] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[You89] D. Young. *X Window Systems, Programming and Applications with Xt*. Prentice Hall, 1989.

[Zuk97] J. Zukowski. JAVA AWT reference, 1997.