MULTIPLE INHERITANCE IN THE DISTRIBUTED OBJECT-ORIENTED LANGUAGE DSELF

Rui Zhu

December 2001, Berlin

A thesis submitted to the faculty of Technical University Berlin in partial fulfillment of requirements for the degree of Diplom-Informatiker. Advised by Dr. Robert Tolksdorf.

Diese Diplomarbeit entstand an der Technischen Universität Berlin im Fachbereich Informatik als Teilerfüllung der Anforderungen von dem akademischen Grad Diplom-Informatiker. Betreut wurde diese Arbeit von Herrn Dr. Robert Tolksdorf.

Abstract

The object-oriented programming language dSelf is a derivative of the SELF language with the extension for distributed computing. It is implemented currently in Java. The aim of this work is trying to improve the multiple inheritance in our dSelf implementation. In this thesis, we will first examine generally the inheritance in object-oriented programming languages. Then the special issues about the multiple inheritance will be discussed. Finally we will describe the details about the multiple inheritance in our implementation.

Zusammenfassung

Die objektorientierte Programmiersprache dSelf ist ein Derivat zur Sprache SELF mit einer Erweiterung der Verteiltheit. Sie wird z.Z. in Java implementiert. Unser Ziel ist eine Versuchung, daß die mehrfache Vererbung unserer Realisierung verbessert werden soll. In diese Arbeit wird zuerst die Vererbung in objektorientierten Sprachen allgemein untersucht. Dann wird die spezielle Sache der mehrfachen Vererbung diskutiert. Schließlich erfolgt eine Beschreibung unserer konkreten Implementierung.

Acknowledgements

First and foremost, I thank my advisor, Dr. Robert Tolksdorf. It was he who had inspired the subject in this thesis. He has also given me much valuable advice during the whole process of my working on this subject. This thesis would not have come about without his heartfelt encouragement and kindly support.

I also want to thank Kai Knubben for his work on dSelf, which had built a solid basis for the further development in this thesis. Again from him I got several good tips on my subject.

Special thanks go to my family and friends, whose trust have given me confidence necessary to finish this project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

SELF is an object-oriented programming language that has *no* concept of classes. Objects in SELF are not instantiated from classes, but copied or cloned from other objects, which often called prototypical objects or simply prototypes; that is, in principle, objects in SELF are capable of containing both behaviors and states, whereas in class-based object-oriented programming languages it is the classes that specify behaviors and shared states, and objects instantiated from classes hold the local states. In SELF, where behaviors, shared or local states of objects should go is decided by the user of language; thus, SELF does not provide *policy* here but a simple while at the same time quite general *mechanism*.

Objects in SELF manifest themselves by the ability of receiving messages and react upon them like most of other object-oriented programming languages. One gem of SELF is just this uniformity of message passing paradigm. For example, there is no difference between function (method slot in SELF) calls and variable (data slot in SELF) accesses. When an object is sent a message, it generates a response, might it be value in one of its data slots, a result from invocation one of its method slots, or got from somewhere else about which no one might concern. Some other examples: conditionals can be realized by sending message to the object *true* or *false*, iterations constructs can be implemented by sending message to numbers, collections, etc.. Especially important to note is that such language constructs as conditionals or loops are not primitives in SELF, they are just plain user codes, though implementations are free to provide them in more efficient manner. All those examples show clearly the philosophy of SELF—simplicity.

The characteristic that SELF is based on objects makes it quite attractive in a network environment. Because self-contained objects do not need to carry the extra information of their classes, they can be used in the network environment more conveniently. All communication among objects can be realized by message passing, wherever objects reside. The *dSelf (Distributed SELF)* is born from this idea, i.e., taking those advantages of SELF and making it work in distributed environment. In an earlier attempt [28] the concept of distributed SELF was developed and implemented, and in this work enhancement should be made, especially the multiple inheritance be improved.

dSelf in its current incarnation is built with programming language Java. dSelf can be used as interactive environment, where an user types the source codes into the read-eval-print-loop (REPL), or used as compiler, where dSelf source files are read and translated in batch to Java byte code, which can be later loaded into a dSelf world much faster.

The distributed feature of dSelf is realized by means of Java RMI (Remote Method Invocation). RMI makes it possible that objects on local Java virtual machine communicate with objects on other remote Java virtual machines, in turn allows the dSelf objects in different dSelf worlds to communicate with each other.

In this work we will first look at inheritance in general, then specially focus on multiple inheritance, for example, its problems, resolutions and implementation issues. After that we will describe our concrete implementation for multiple inheritance in dSelf. Finally there are outlook for future development and some other auxiliary information.

Chapter 2

What is Inheritance

> **inheritance** ... **a:** the act of inheriting property **b:** the reception of genetic qualities by transmission from parent to offspring **c:** the acquisition of a possession, condition, or trait from past generations ...
>
> — Merriam-Webster's Collegiate Dictionary, 10th edition [34]

A characteristic feature of object-oriented programming languages is *inheritance*. It is commonly regarded as a feature that distinguishes object-oriented programming paradigm from others. Many people accredit for the most part many of the alleged benefits of object-oriented programming, such as reusability, more robust and faster software development, and improved conceptual modeling, to it. But it is exact this notion of object-oriented programming, that is at the same time most problematic. In this chapter we are going to see some definitions of inheritance.

2.1 Introduction

The concept of inheritance in object-oriented programming comes from the usage of this term in the real world. In general, inheritance means the reception or sharing of properties or characteristics from another, as result of some special relationship between the receiver and giver. In this broad definition, inheritance is quite easy to understand and there is also few disagreement thereat. But an unique and precise definition seems still not be accepted by all.

As early as two thousand years ago classification has been the interest of famous philosophers Plato and his student Aristotle [45]. Object-oriented programming languages today have the same ideas as those classifications of Aristotle. The general rule for classification can be presented as:

$$essence = genus + differentia.$$

Here we could see inheritance (genus) is an importance part of central idea of object-oriented programming languages.

The concept of inheritance in programming languages was at least as old as the programming language Simula [15], in which it is a central feature, though originally defined in terms of textual *concatenation* of program blocks. Some time later the seemingly intuitively more appealing term *inheritance* was established. Till now, there are also some other well-known synonyms for inheritance in object-oriented programming, such as prefixing (in Simula and Beta), subclassing, derivation (in C++).

2.2 One Definition

In [4] inheritance was defined as "a mechanism for *making one* class or type *from another*, in which *self* is late-bound." From this definition two essential elements of inheritance have been clearly pointed out: one is somehow getting new one from old, the other is importance of late-bound self. We will discuss these in detail in Chapter 3 on page 7.

We will also use the a little wider meaning of inheritance, i.e., inheritance occurs not only among classes or types, but also among objects, because although in object-oriented systems inheritance is typically used among classes, there is usually an isomorphic correspondence between classes and their instances, besides there are also object-oriented systems which are not based on classes.

2.3 Inheritance As Incremental Modification

The basic idea of inheritance is not complicated. Inheritance allows new class (or new object in object-based object-oriented languages[1]) definition based on existing ones. Inheritance is essentially a facility for *differential* or *incremental* development. When a new class is to be defined, only those properties that are different from the properties of some existing objects need to be defined explicitly, while the other properties can be automatically *extracted* from the existing ones and *included* in the new object. Here the term extract and include are only in the sense of abstract usage, and should not be thought as a concrete mechanical process.

More formally, in [10] inheritance is identified with the subtype relation on record types: "a record type τ is a subtype (written $\leq$) of a record type τ' if τ has all the fields of τ', and possibly more, and the common fields of τ and τ' are in the $\leq$ relation."

Similarly, inheritance can be specified as follows [45, 44]:

$$Child = Parent + \Delta.$$

It shows remarkable resemblance to Aristotle's classification schema. Here *Child* (*essence*), donates a newly defined class, *Parent* (*genus*) denotes the properties inherited from an existing class, Δ (*differentia*) denotes the incrementally *modified* properties that differentiate *Child* from *Parent*, and $+$ denotes an operation to somehow combine Δ with the properties of *Parent*. So *Child* will contain all the properties of *Parent*, except that the modified part Δ may create properties that overlap with those of *Parent*, causing *redefinition* or *cancellation*. As the result, *Child* might not be fully compatible with *Parent*.

[1] In some object-oriented programming languages classes are also first-class object.

2.4 Inheritance in Operational Semantics

Another widely referred definition of inheritance is given by using of the operational semantics [13]. The canonical semantics is the *property lookup* algorithm of respective programming language, in which late-bound self also plays an important role. For example the lookup algorithm in object-oriented programming language SELF [1] (some details about SELF can be found in Appendix A on page 79):

> The lookup algorithm recursively traverses the inheritance graph, ... No object is searched twice along any single path. The search begins in the object itself and then continues to search every parent. Parent slots are not evaluated during the lookup. That is, if a parent slot contains an object with code, the code will not be executed; the object will merely be searched for matching slots.

2.5 An Example

As an example of inheritance, consider the following inheritance hierarchy (Figure 2.1) and code segment in SELF (Figure 2.2 on the next page). In the example, three objects, `window`, `labeledWindow` and `borderedWindow`, are defined. The object `window` defines four slots. The object `labeledWindow` inherits `window` and adds a new slot, object `borderedWindow` is similar.

(The code may seem not comprehensible at the first look if one has not encountered SELF, but it is actually quite simple, even before we introduce more details of SELF. "`(|`" and "`|)`" are used in Self for grouping slots (or fields, elements, which might be more familiar terms). "`<-`" and "`=`" are used for designating right hand side to left hand side, but there is a small difference, that the former donates changeable slots and later constant ones. And slots which end with a asterisk have special meaning, that it donates a parent slot.)

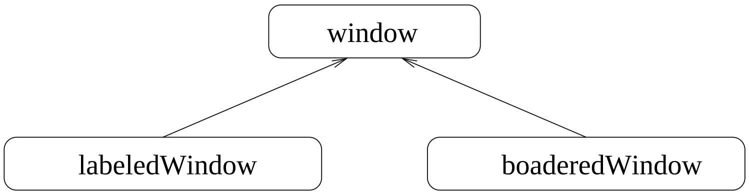

Figure 2.1: A simple example of inheritance

```
window <- (| x <- 0.
             y <- 0.
             width <- 0.
             height <- 0.
          |)

labeledWindow <- (| parent* = window.
                    label <- 'Hello'.
                 |)

borderedWindow <- (| parent* = widnow.
                     borderWidth <- 1.
                  |)
```

Figure 2.2: A code segment in SELF

Chapter 3

Essence of Inheritance

We now take a more detailed look at the two essential elements of inheritance from the definition in Section 2.2 on page 4.

3.1 Incremental Modification

As shown in [47, 20], four increasingly permissive categories of the incremental modification mechanism can be identified.

Behavior-compatible modification
: This imposes the strongest-possible constraints on behavioral relationships. The child preserves the characteristics of parents and may add characteristics but not exclude them. Redefinition is only permitted if the behavior of the child operation is preserved or augmented. But compatibility is usually based on algebraic and axiomatic models that tend to be impractical to implement. Syntax is expressed using signatures.

Signature-compatible modification
: Behavior is approximated by a signature, comprising operation name, plus name and types of formal arguments and return value, therefore full behavioral compatibility can not be ensured. Signature compatibility may be operationally constrained by the use of inclusion relations on domains (contravariance). Type-checking algorithms usually define subtypes in this way because of the difficulty, even impossibility, in operationally determining strict behavioral compatibility.

Name-compatible modification
: Only the names of the set of parent operations need be preserved. This represents the usual modification mechanism found in untyped object-oriented programming languages. The relation enforces no constraint other than name preservation, although any programmer encountering an object-oriented programming language with an inheritance mechanism such as this may apply stronger inheritance relations through self-imposed standards.

Inheritance with cancellation
: While the three previous categories decrease in the severity of constraint thus loosening the set membership of children, inheritance with cancellation also permits the extension of set membership by excluding parent

characteristics. This category is also known as non-monotonic inheritance and is the weakest of the categories.

The first three categories can be seen as *refinement* and refer to the process of behavioral extension. The main requirement of refinement is the preservation of the parent interface. So behavioral and signature compatibility are strong refinement relation, and name compatibility is relatively weaker.

The last category can only be seen as a relation of *likeness*, given its non-monotonic behavior. All that can be said about likeness is that it means some common characteristics.

3.2 The Late-Bound Self

Late binding and self-reference are quite important to realize the incremental modification mechanism. Late binding promotes the physical code sharing, by letting the same operations invoke different properties depending on the context from which these operations are invoked.

We now examine what happens, when a late-bound message is sent to an object. The simplified process is illustrated in Figure 3.1 [13].

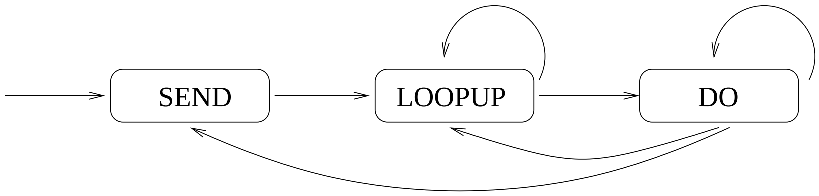

Figure 3.1: The message passing semantics with late-binding

1. The self-reference is set to refer to the receiver of the message. Old value of self-reference is saved so that it can be restored later. This is called SEND in Figure 3.1.

2. The given message selector is then matched against the properties of receiver. The actual traversal depends on the form of inheritance being used. This is called LOOKUP in Figure 3.1.

3. If a matching property is found, that matching property is invoked, with current value of self. This is called DO in Figure 3.1. After execution, the saved previous value of self-reference is restored.

As a result of late binding, the matching operation will be invoked, with the self-reference denoting the receiver of message. Therefore subsequent message sends via self-reference from within that operation can access the other properties of the object. By virtue of late binding, operations inherited from parents can invoke such properties that were not necessarily even implemented at the time when the parent was defined. (But in the actual world, it is not so simple and beautiful for all object-oriented programming languages, especially in static typed languages, such C++ or Java, where programmer must at least specify a virtual method, i.e. a place holder for it.)

The self-reference typically appears to the programmer in the form a special pseudo-variable. By the term pseudo-variable it is meant that system is responsible for maintaining it and programmer can not modify it.

Different object-oriented languages use different names for this self-reference. For example, in Smalltalk and its derivatives it is called *self*, whereas in C++ and Java *this*, and in Eiffel [35] the corresponding language construct is known as *current*. But there are also languages, such Perl and Python, in which the programmer needs to explicit name the self-reference, i.e., there are no such system provided shorthand for self-reference.

Chapter 4

Aspects of Inheritance

Although the basic concept of inheritance seems quite intuitive, as it is a simple linguistic mechanism with relatively straightforward semantics from the theoretical point of view, the fact that there are actually many variations of inheritances makes the analysis of inheritance difficult in practice. Therefore, in this chapter we try to entangle this by approaching it from many different directions. They are not meant to be orthogonal and may overlap themselves. We will examine multiple inheritance more closely in later chapters.

4.1 Message Lookup Process

In Section 3.2 on page 8 we described the general message sending algorithm that can be seen as underlying all inheritance mechanisms. The algorithm consisted of three elements: SEND, LOOKUP, and DO. In particular, the LOOKUP part of algorithm has only stated simply that as part of message matching the inheritance graph (a directed acyclic graph—DAG) will be traversed node by node, but not what the actual traversal order is. Neither did we completely specify when this lookup will eventually end. For example, if the DAG contains several overlapping identifiers, it is unclear which one of these properties should be invoked, or should they all become invoked. In the later case it would be important to specify also the exact order in which all these overlapping properties are to be executed. Moreover, in case of multiple inheritance the overlapping of identifiers is often accidental, and undesired identifier name collisions should somehow be taken into account.

In most contemporary object-oriented systems the message lookup proceeds from the most recently defined (descendant) parts of objects towards the least recently defined parts (parents). This convention is used in various object-oriented programming languages including Smalltalk, Self, C++, Eiffel and CLOS, except that in many systems multiple inheritance makes things somewhat more complicated. Since the lookup is dominated by the most recently defined parts of objects, i.e., those properties that have been defined at the descendant level, people call this scheme descendant-driven (or bottom-up) inheritance.

As usual, there are some interesting variations. Beta [32, 33], a descendant of Simula, uses a totally opposite parent-driven (top-down) lookup approach.

In Beta, property lookup is started from the topmost superpattern (the Beta equivalent of superclass) in the current inheritance DAG. Beta supports single inheritance only, so this topmost superpattern is always unique, and the inheritance DAG can be seen simply as linear path from parents to descendants. When a message is sent to an object (in Beta terminology: a virtual procedure is invoked), lookup proceeds from the virtual procedures defined in the topmost superpattern towards more recently defined superpatterns of the object until a matching procedure is found. This matching procedure will then be invoked in the normal manner. Since superpattern have precedence over their subpatterns, the Beta-style reverse inheritance strategy ensures that descendants can never totally override the behavior inherited from parents, thereby providing additional support for maintaining the behavioral compatibilities between parents and descendants.

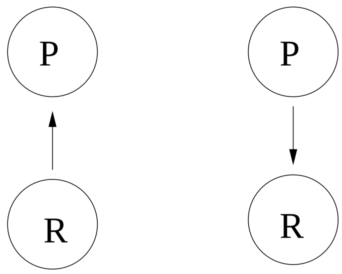

Figure 4.1: Message lookup direction.

The distinction between descendant-driven and parent-driven lookup schemes is illustrated in Figure 4.1. Note that the parent-driven lookup scheme is not really suited to systems with multiple inheritance, because in such systems inheritance graphs (DAG's) do not necessarily have any single root node from which the lookup could be started.

Besides the differences in the direction of message lookup, the inheritance scheme of Beta deviated from mainstream object-oriented languages also in other ways. One important difference pertains to the completion or exhaustion of message lookup. In most object-oriented systems message lookup is terminated as soon as a matching property has been found and executed. If the programmer wants to execute the overridden properties inherited from the parents, it is the programmer's responsibility of explicitly do so by defining the methods to explicitly call the methods of their parents. Many object-oriented languages provide a special super-reference exactly for this purpose, as discussed earlier.

Beta addresses the lookup exhaustion issue differently. In Beta, the overlapping properties in subpatterns will always be invoked automatically without programmer intervention. After finding and execution a matching property, the lookup will be continued implicitly in the parent-driven lookup order until all the corresponding properties in the object have been executed. This is essentially different from mainstream object-oriented systems in which even one matching property will suffice the termination of the lookup. The Smalltalk-style of terminating the lookup immediately after encountering the first matching method is more flexible, but is also a potential source of errors since the programmer may forget to execute the corresponding property in the superclasses.

Note that although the inheritance scheme of Beta is radically different from than in most other object-oriented systems, these systems are typically capable of simulating each other quite easily.

In Smalltalk, for instance, Beta's composing message lookup scheme can be simulated simply by systematically placing an explicit super-reference within every method. Moreover, if these super-references are always positioned in the beginning of each method, the lookup order becomes logically parent-driven, even though the physical search order still remains descendant-driven. The same technique can be used in Beta which provides a special inner construct for controlling the order of subpattern execution [32]. The behavior of inner is analogous to that of super-reference, but its direction is reverse. Rather than invoking properties of superclasses, Beta's inner construct invokes properties in subpatterns. Thus, by placing an inner-reference in the beginning of each virtual function in a Beta program, the lookup order becomes logically descendant-driven. However, there is no easy way to stop calling subpatterns in Beta.

There are also other techniques that can be used to control the direction and exhaustion of message lookup. In particular, mixin inheritance discussed in Section 4.8 on page 20 allows all the inheritance schemes to be simulated flexibly; different schemes are achieved simply by changing the order in which base classes and mixins are combined. For a thorough discussion on this idea, refer to [8]. Among the currently available object-oriented languages, the CLOS system with its special before, after and around methods provides perhaps the most comprehensive facilities for dealing with the lookup direction and exhaustion.

4.2 Class-Based Versus Object-Based

Object-oriented systems are usually built around classes. Classes work like templates that are descriptions of objects, from which instances, the actual objects described by classes, can be created. This creation process is typically known as instantiation. In broad terms, a class represents a generic concept, or a recipe, while an instance represents an individual. A class holds the similarities among a group of objects, dictating the structure and behavior of its instances, whereas instances hold the local data representing the state of the object.

Class-based systems are class centric. To add a new kind of an object to the system, a class describing the properties of that object type must be defined first. Similarly, inheritance can only take place between classes, whereas instances are incapable of serving as parents. Therefore, the model is commonly referred to as class inheritance.

An interesting alternative for traditional class inheritance is prototype inheritance, or object inheritance. As opposed to class-based systems, in prototype-based systems there are no classes. Rather, new object types are formed directly by constructing concrete objects, which are referred to as prototypes. A prototype can be thought of as a standard example instance, which represents the default behavior for some concept. Prototype-based systems provide no classes, and therefore in these systems there is no notion of instantiation either. The effect of instantiation, the ability to create multiple objects of similar kind, is achieved by cloning or copying existing objects instead. In addition, objects in prototype-based systems are usually individually modifiable. By virtue of the individuality of objects, prototype-based systems support incremental modification at the level of individual objects, as opposed to class-based systems in which typically only groupwise or class-level modification is possible.

From the philosophical viewpoint the distinction between class-based and prototype-based systems reflects the philosophical dispute concerning the representation of abstractions. Plato viewed descriptions of things as having an existence more real than instances of those abstractions in the real world. Object-oriented languages like Smalltalk and Simula are Platonic in their explicit use of classes to represent similarity among collections of objects. Prototype-based systems represent another way of viewing the world, in which one chooses not to categorize things, but rather to exploit their alikeness. A typical argument in favor of the prototype-based approach is that people seem to be a lot better at dealing with specific examples first, then generalizing from them, than they are at absorbing general abstract principles first and later applying them in particular cases.

There are many variations of prototype-based systems. Roughly speaking, prototype-based object-oriented languages can be divided into two broad categories according to how inheritance is supported in them: delegation-based and copying-based. In delegation-based languages, there is a special delegation mechanism that provides the inheritance. In delegation-based languages, when an object receives a message it does not understand, the object will delegate the message to those objects that have been designated as its parents. Parent objects will then try to carry out the message on the behalf of delegating object. During the delegation process, the self-reference will remain pointing to the original receiver, allowing late-bound properties in parents to operate in the context of the original receiver. The best example of a delegation-based object-oriented language is SELF.

Others has shown that a prototype-based language does not necessarily have to support delegation in order to be a full-fledged object-oriented language. It is demonstrated that by utilizing cloning (copying) and the possibility to dynamically add new properties to objects in the presence of the regular late-binding message sending mechanism, it is possible to capture the essence of inheritance-incremental modification in a prototype-based language even without delegation. But as a result of extensive use of copying, memory consumption can be high unless there is special handling in implementation. Also, groupwise modification of objects is more tedious than in delegation-based systems due to absence of common parent objects or any explicit inheritance hierarchy. Examples of prototype-based object-oriented languages are Omega [5].

4.3 Delegation Versus Concatenation

Inheritance is acquisition of properties from past generations. More practically, inheritance is a relationship between objects that guarantees that a descendant has at least the properties that its parents have. By incrementally adding, redefining or defeating properties, the descendant can become differentiated from its parents. At the level of object interfaces this implies that the record representing the descendant must contain at least those identifiers that its parents have. In addition, the descendant can introduce new identifiers that are either distinct or overlapping. An incrementally added identifier is said to be distinct if no other identifiers with the same name exist in the same inheritance DAG.

Otherwise, the identifier is overlapping. In general, from the viewpoint of interface management, inheritance is simply a combination operation with possibly overlapping identifiers [47].

Within computer memory there are only two ways to express direct relationships between things: references or contiguity. Based on this observation, two elementary strategies for implementing inheritance can be distinguished: delegation and concatenation. Delegation is a form of inheritance that implements interface combination by sharing the interface of the parent in other words, using references. Concatenation, on the other hand, achieves the same effect by copying the interface of the parent into the interface of the descendant as a result of this, the descendant's interface will be contiguous.

In both cases the basic result is the same: the descendant will be able to respond to messages corresponding to the inherited identifiers in addition to the incrementally added ones. However, the way in which message sending is handled in these approaches is different. In delegation, those messages that are not accepted by the most specialized part of the object must be delegated (forwarded) to parents. In concatenation, the interface of every object is self-contained, and thus no forwarding is needed. The term delegation implies shared responsibility for the completion of a task.

Note that the difference between delegation and concatenation is independent of the question of whether the system is based on classes or prototypes; delegation and concatenation can equally underlie both models. The main difference is that in class-based systems delegation or concatenation takes place between classes, whereas in prototype-based systems the same occurs between individual objects. In fact, although the term concatenation is not widely used in the current literature of object-oriented programming, there are good historical reasons for its use. The concept of prefixing in Simula, which later evolved into the concept of inheritance, was originally defined in terms of textual concatenation of program blocks. Even in the some implementations of Simula and its descendant Beta the interfaces (to be more specific: the virtual operation search tables) of classes and their subclasses are self-contained and independent of each other. On the other hand, the way that inheritance between classes is achieved in Smalltalk is analogous to the delegation approach adopted in prototype-based languages such as SELF. In both approaches unresolved messages will always be forwarded to separate constructs; in Smalltalk this separate construct is a superclass, and in SELF it is a parent object.

In general, delegation and concatenation are distinct strategies for implementing inheritance, both of which can equally underlie class-based and prototype-based systems. Simula and its descendant Beta are examples of languages which implement class inheritance by concatenation. Smalltalk, on the other hand, implements class inheritance by delegation. SELF is the best example of a language that implements prototype inheritance by delegation. The copying-based approach represents prototype inheritance by concatenation.

Inheritance strategy	Delegation	Concatenation
strategy for combination	sharing/references	copying
dependence	dependent	independent
inheritance graph	preserved	flattened

Table 4.1: Delegation versus concatenation

The main differences between delegation and concatenation have been gathered into Table 4.1 on the preceding page. Most of the differences between the approaches pertain to the treatment of inheritance graphs (DAG's). Delegation preserves inheritance DAG's without modifications, but concatenation requires DAG's to be flattened out into contiguous name spaces. This flattening implicitly prevents costly cycles in inheritance DAG's, but at the same time it causes more space needed.

Delegation and concatenation both have certain advantages over each other. For instance, owing to dependent interfaces, delegation enables flexible propagation of change: any change to parents will be automatically reflected to descendants. This can be extremely useful at times, allowing a large group of objects to be modified simply by changing a common parent. On the other hand, thanks to self-contained interfaces, concatenation supports independent modification of object interfaces, making possible renaming and selective inheritance. As for efficiency, delegation is generally more space-efficient due to extensive use of sharing, but concatenation is generally more time-efficient since the absence of sharing makes it easier to optimize the method lookup more aggressively.

4.4 Static Versus Dynamic

In Section 4.3 on page 14 a distinction between two fundamental strategies for implementing inheritance—delegation and concatenation—was made. It was concluded that both strategies have their own benefits that are difficult to capture using the other strategy. In this section this discussion is taken a bit further by discussing dynamic inheritance: the ability to change parents of objects dynamically at runtime. Despite being seemingly a rather dangerous language feature, dynamic inheritance provides some notable benefits, and therefore, although dynamic inheritance has typically been available only in some prototype-based systems, it has nevertheless captured the much interest.

Recall from the earlier discussion that delegation-based systems accomplish incremental modification by sharing with the parent, i.e., using references rather than copying. In many systems, the references between children and parents are usually maintained by the system and are completely inaccessible to the programmer. In contrast, in prototype-based systems these references are typically implemented as parent slots. By giving these slots names, and by making them assignable, we allow the parents of objects to be changed on the fly.

Assuming that the parent slot behaves like an ordinary variable, we can dynamically change the parent slot to refer to some other object. As apparent, dynamic inheritance can be quite dangerous when misused. If the properties of the new parent do not comply with those needed by the descendant, runtime binding errors will result. Dynamic inheritance can also have a deteriorating effect on performance, since the possibility to change parents at runtime prevents certain kinds of message lookup optimizations from being done.

The dynamic inheritance can be quite useful for some examples. In the design and implementation of programs we are often faced with things and structures that can be in different states or conditions. In this context we are not referring to the concrete state of the variables of these objects, such as the value in a variable, but rather to a more conceptual or logical kind of state. For example, container objects such as stacks, lists and queues can be empty, nonempty or full. In graphical user interfaces, windows can usually be open, closed or represented as icons. Similarly, application-specific objects such as flight reservations or bank accounts also exhibit many kinds of logical states. The behavior of the operations of objects typically varies considerably depending on these logical states. For instance, the behavior of the draw operation of a window depends essentially on whether the window is open, closed or in icon state; in the later case draw must not do normal window drawing, for example, frame, title, etc.. Similarly, when a bank account has been frozen by the bank, the teller machine should not dispense any money when a withdrawal is requested but rather swallow the card. In conventional object-object systems, these kinds of logical states or modes, as they are sometimes called, are hard to express. Usually, additional state variables and control structures are needed. For instance, each of the basic window manipulation operations such as open, close, iconify, resize and zoom typically has to contain an explicit if-statement or case-statement to determine whether the operation is appropriate for the window in the current situation. Obviously, the diffusion of mode information and accumulation of control structures both tend to have a negative impact on the readability and maintainability of code.

Dynamic inheritance provides a solution to the problems with logical states. Since the behavior of objects can be changed at runtime, not all the information about an object has to be described in one class. Rather, by implementing a separate set of methods for each logical state, the behavior of each mode can be kept clearly apart from the others. Each of the parent objects implements the same basic set of methods, but the behavior of the methods is specialized to each particular logical state. Using dynamic inheritance, method definitions can be kept concise and clear, because the methods in one mode do not have to know how the other modes behave. The number of control structures in programs is reduced substantially.

Note that dynamic inheritance is not typically possible in class-based systems. This is mainly due to the fact that in class-based systems classes describe not only behavioral but structural aspects as well; if the superclass of a class were suddenly changed at runtime, the structure of the instance would likely be incompatible the structure specified by the new superclass. The fact that some class-based systems do not support late binding of variables makes the situation even more problematic.

4.5 Benefits of Concatenation

Just as delegation enables dynamic inheritance, concatenation gives rise to some interesting variations of inheritance. For instance, the self-contained nature of object interfaces in concatenation has the advantage of making the object interfaces independent of each other. This possibility may be utilized in Eiffel, which allows the inherited properties to be renamed in subclasses.

Besides allowing renaming, independent interfaces have other benefits. In particular, they make some forms of inheritance possible in which not all properties of inherited abstractions have to be passed to descendants. Rather, the descendant can be given the explicit possibility to decide which particular properties of parents to inherit and which not. Such forms of inheritance are said to represent selective inheritance. Selective inheritance has been tried out in

Another benefit of concatenation is also due to its self-contained and independent property—concatenation can be an advantage in network or distributed environment. The communication will be reduced and self-contained objects can more freely migrated from one place to another.

Yet another interesting aspects of selective inheritance is that it shows that the difference between inheriting an abstraction (such as a class or a prototype) and using the abstraction as a variable is actually very subtle. From the interface combination viewpoint the only difference between these is that in the former case the property names of the inherited component are implicitly included into the interface of the descendant, and thus the inherited properties can be referred to directly. In contrast, in the later case the properties are not included and must be accessed indirectly via an intervening variable.

Note that in the presence of selective inheritance the distinction between parents and variables becomes blurred. Selective inheritance allows only a subset of the identifiers of parents to be brought into the interface of the descendant, and hence the degree of "parentness" and "variableness" can be varied. In particular, if a language provides a mechanism that any uniquely named property within an object can be accessed directly without having to specify the full path to it, then from the viewpoint of interface management the distinction between variables and parents will vanish altogether. Of course, in most object-oriented languages encapsulation (information hiding) changes the picture to some extent, since not all variables or methods may be accessible to outside clients due to being declared private.

The use of selective inheritance can also lead to some conceptual and technical problems. In particular, careless omission of inherited properties can produce objects whose interfaces are incomplete, and that do not possess all the properties to which their operations need access. Therefore, additional mechanisms may have to be introduced to guarantee the completeness of objects.

4.6 Single Versus Multiple Dispatching

Most object-oriented systems are based on single dispatching. Messages are sent to distinguished receiver objects, and the runtime type of the receiver determines the method that is invoked by the message. The possible parameters in the message are passed on to the invoked method, but do not participate in the actual lookup. Although this approach is intuitive and works well in most situations, it has some drawbacks. For instance, one implication is that a binary expression such as $1 + 2$ gets the unnatural interpretation:

$$1.plus(2).$$

In other words, 1 is treated as the receiver of the message plus, while 2 is regarded as a parameter that plays no role in actual method lookup. Although such interpretation works fine in most situations, it is conceptually undesirable since in arithmetic expressions right hand sides are usually no less important than left hand sides. It can also have a negative effect on program readability. For instance, in Smalltalk all binary expressions are parsed left to right, and normal arithmetic precedence rules do not apply. This makes mathematical expressions in Smalltalk different from those in most other languages in which multiplication and division take precedence over addition and subtraction.

Besides conceptual problems, single dispatching can also lead to some more technical problems and require duplication of code. In order to make single dispatching handle properly situations in which the same type of object may receive different kinds of parameters, the use of additional control structures is usually needed. For instance, sending a message $1 + 2.2$ requires the integer method to be able to check at runtime whether the parameter is an integer or a floating point number, and handle both integer and floating point addition. Conversely, since it is equivalently legal for the programmer to send a message $2.2 + 1$, the floating point class must include the corresponding code to handle both real numbers and integers. As a result, the same pieces of code may have to be included in two places, or, if the message is $n$ary rather than binary, in n places.

In general, traditional message passing schemes are sometimes too "selfish." There is a single receiver whose type determines which method is executed, while the parameters are ancillary and have no role in method selection. An object-oriented language based on this kind of an approach is termed a single dispatching language. In many situations, however, it would be beneficial to generalize the message binding mechanism to more than just single receiver. To overcome the limitations of single dispatching languages, some object-oriented languages include a more general form of message passing in which multiple arguments to a message can participate in the lookup process. These languages are called multiple dispatching languages. Since methods in multiple dispatching languages may belong to several classes simultaneously, they are called multimethods. Multiple dispatching is common in Lisp-based object-oriented systems, where multimethods are known as generic functions. The best-known multiple dispatching object-oriented language of today may be CLOS. There are also other languages that support multimethods, such as Dylan and Cecil.

Some researchers claim that multiple dispatching provides increased expressive power over single dispatching. With multimethods, the message lookup may involve all arguments, not just the receiver, and thus the above mentioned problems with the asymmetry of single dispatching can be avoided. Multiple dispatching is far from being widely accepted, however. Multiple dispatching languages are generally more complex than single dispatching languages, and their implementations have not traditionally been very efficient (it does not mean that it can not). In practice, the biggest single concern to many programmers is however that languages supporting multiple dispatching do not feel object-oriented. This feeling reflects a basic difference in the programming styles encouraged by single and multiple dispatching systems. When using a single dispatching language, the programmer mentally focuses on defining abstract data types (ADT's), associating operations with the data types for which they are implemented. These operations are contained in the data type, and together the operations and the data form an encapsulated whole. A whole design and implementation methodology has been developed around this ADT-oriented programming style.

Multiple dispatching languages do not provide much linguistic support for the ADT-oriented programming style. Since multimethods are dispatched on multiple arguments, they are not contained in any single class, and the ADT oriented view with strong encapsulation breaks down. In practice, the programming style encouraged by multiple dispatching systems seems to be closer to functional, or traditional procedural programming style than ADT style.

To many advocates of object-oriented programming, multimethods which originated from Lisp-based systems seem like an ad hoc solution that is a consequence of trying to fit round objects to the square world of functions. This feeling is reinforced by the fact that multimethods do not look object-oriented. In multiple dispatching languages the use of the object-oriented *object.message*(*parameters*) notation is discouraged, and the traditional *function*(*parameters*) or Lisp-like (*function parameters*) notation is used instead. Although some researchers have proposed (*receiver1*, *receiver2*, ..., *receiverN*).*message* notation as a compromise, these notations are not as well suited to ADT programming style as the normal dot notation or its equivalents.

4.7 Single Versus Multiple Inheritance

Many modern object-oriented programming languages allow inheritance from more than one parent at the same time. Such inheritance is known as *multiple inheritance*, as opposed to *single inheritance*. While multiple inheritance offers considerably more possibilities for incremental modification than single inheritance, it also introduces many conceptual and technical difficulties. In spite of the intricacies of multiple intricacies, many believe that multiple intricacies should be supported in modern object-oriented programming languages.

4.8 Mixin

Mixin inheritance is a quite particular way of using inheritance that has received also quite a lot of attention in the literature of object-oriented programming. This is not surprising, considering that mixin inheritance has certain important theoretical and pragmatic benefits. On the theoretical side, mixin inheritance has been proven to be capable of capturing the functionality of several other forms of inheritance. Therefore, mixin inheritance is sometimes regarded as a more elementary form of inheritance that can provide help in understanding the other forms. On the pragmatic side, mixin inheritance is beneficial because it can considerably improve the reusability of program components.

The basic idea of mixin inheritance is simple. Unlike in ordinary inheritance, in which the modification (delta) parts in class definitions are embedded directly in the new definitions, in mixin inheritance separate mixin classes are created to hold the modifications. A mixin class is syntactically identical to a normal class, but its intent is different. Such a class is created solely to be used for adding properties to other classes; mixins are not created for instantiate their own objects. New concrete classes are constructed by combining primary parent classes with mixin classes using multiple inheritance. Such different use of ordinary and mixin inheritance is sketched in Figure 4.2 on the facing page.

Since mixin classes mostly do not have superclasses and are not therefore structurally bound to any specific place in the inheritance hierarchy, the same modifications (delta parts) can be repeatedly used (inherited) without having to rewrite or copy their contents manually when similar modifications are needed in other places . This enhances reusability by allowing the same functionality to be flexibly added to various components.

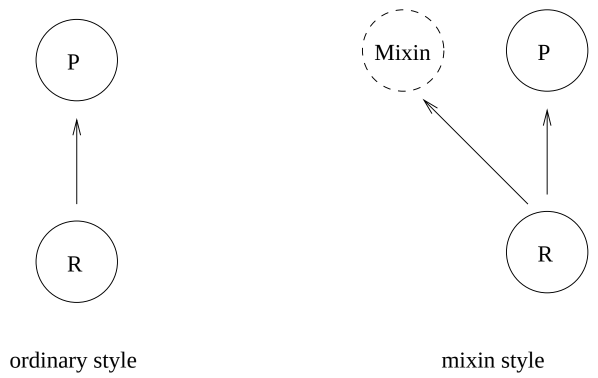

Figure 4.2: Ordinary versus mixin inheritance

Only using single inheritance in a similar situation, considerable code duplication would have been needed because the same pieces of code would have to be repeatedly written in several classes. Ordinary multiple inheritance would usually suffice to solve the problem, too. However, ordinary multiple inheritance is prone to cause difficulties if the same classes become inherited two or more times along different inheritance paths (e.g., class has two superclasses that have a common superclass), and such problem is commonly referred to as repeated inheritance or diamond inheritance.

Note that in mixin inheritance multiple inheritance is only used for combinatory purposes. Provided that the mixin classes are implemented correctly, no additional manual gluing is needed when base classes and mixins are inherited (combined). In practice this requires that the methods of mixin classes are implemented in such a way that they are open to extensions, and remember to invoke the corresponding methods in their surrounding environment.

Mixin inheritance is not without problems. From the conceptual point of view, one of the less beneficial characteristics of mixin inheritance is that it tends to cause confusion in the relationship of object-oriented programming and conceptual modeling by diversifying the role of classes. As apparent, in mixin inheritance classes are used for three distinct purposes. First, there are base classes that define the general characteristics of a hierarchy of concepts. These classes are either concrete instantiable classes, or abstract noninstantiable classes that exist solely to be refined by their subclasses. Second, there are mixin classes that serve as elementary units of reusable behavior. Mixin classes cannot be used in isolation but must first be combined with a concrete class to become functional; therefore, mixin classes are sometimes referred to as abstract subclasses. Third, mixin-based systems also have combination classes that are needed for instantiating concrete instances with the desired functionality.

In summary, mixin inheritance can considerably increase the reusability of class definitions, but at the same time it may decrease the conceptual clarity of the system by requiring the same class mechanism to be used for three different purposes in a rather subtle and unintuitive way. In this sense, mixin inheritance serves as a practical example of the fact that reusability and conceptual modeling are opposite goals: in order to obtain maximum reusability, one usually has to sacrifice modeling, and vice versa.

4.9 Pragmatic Use of Inheritance

Inheritance as a languages mechanism allows new object definitions to be based on existing ones. A child inherits the properties of its parents, and introduce new properties that extend, modify or defeat its inherited properties. In general, most object-oriented programming languages allow the inherited properties to be overridden. Some languages even allow the inherited properties to be renamed, duplicated, removed, have their visibility changed, or other kinds of transformation. As a result, it is quite understandable that inheritance is used for diverse purposes.

4.9.1 Implementation Inheritance

In some situations inheritance is used because the abstractions to be inherited contain properties for the new abstraction that is under construction. In implementation inheritance, theoretical and conceptual issues such as behavior compatibility are ignored, and pragmatic reasons such as potential savings in coding effort, storage space or execution speed are emphasized instead, though doing inheritance for mere implementation reasons is often questionable [38].

Some most often referred reasons for implementation inheritance are: cancellation, combination, inclusion, and specificational structuring tool, in order to achieve optimization, convenience, expressiveness, and structuring of software.

Means

In cancellation, the descendant restricts the inherited behavior by explicitly making certain inherited properties unavailable, or by tightening the parameter type requirements.

In combination, inheritance is used for combining existing abstractions with multiple inheritance. Two subforms of combination exist: combining abstractions of equal importance and mixin-based approach.

In inclusion, inheritance is used for simulating modules or function libraries. And no extra module constructs are needed any more.

In specificational structuring, abstractions with purposely incomplete definitions, whose properties may refer to such components or properties that have not even been implemented yet (late-binding is obviously needed for this). They serve as specifications or contracts upon which the concrete implementations can be based, thus facilitating the convergence of analysis, designing, and implementation. (communication role, interface, abstract class, virtual class, deferred class, etc.)

Purposes

For optimization, the descendant takes advantage of some specific implementation information to improve the code for its operations. For example, descendant may have elements ordered in sorted double linked list, while in ancestor only in single linked list, so descendant can use better algorithm to implement search operation.

For convenience, the new class is made a subclass of existing classes simply because the existing classes seem to provide what is desired, though sometimes there is a need to perform cancellation, overriding or combination.

For expressiveness, inheritance is used as language construct for modules, libraries, etc.

Chapter 5

Multiple Inheritance

In this chapter we take a very general overview of multiple inheritance in object-oriented programming languages.

As most agree, inheritance is a distinguished feature in object-oriented programming languages, but as far as multiple inheritance is concerned, the opinions are still quite differentiated. Some people think multiple inheritance a controversial feature of object-oriented programming languages, as evidenced by its omission from languages such as Java and Ada. However, there are also some other people who will complain if multiple inheritance is absent. For example, Eiffel or CLOS people seem quite fond of multiple inheritance being in their tool box. Multiple inheritance is a mixed blessing.

In the following sections we will look at multiple inheritance from both its advantages and its drawbacks.

5.1 Advantages of Multiple Inheritance

As we have said, multiple inheritance is considered by many people a useful feature. This can shown by its common adoption in languages, such as C++, Eiffel, CLOS and Dylan. It is even sometimes a wished feature by some people in languages that do not provide it, and those people complain that they have to resort to devising their own mechanisms to accomplish what they otherwise would have used[1] multiple inheritance to achieve.

There is no consensus as to when multiple inheritance is desirable. However, many people agree that multiple inheritance does have its advantages when used properly. We will view some of them briefly.

5.1.1 Multiple Specialization

Sometimes one class can be quite naturally the conceptual specialization of more than one other classes. For example, a class `InputOutputStream` can be seen as a specialization of both the class `InputStream` and the class `OutputStream`.

[1] Might it be disciplined use or not.

5.1.2 Polymorphism

One of features of object-oriented programming languages is polymorphism through inheritance, also called inclusion polymorphism. Multiple inheritance enhances this feature, because only having a single inheritance is potentially limiting.

5.1.3 Mixin Style of Programming

As we have discussed in Section 4.8 on page 20, a mixin class can encapsulates a general attribute or functionality that may be of interest to many other classes. Mixin classes are generally small and independent, and a class using mixins may often tend to inherit several of them. The mixins themselves are not normally used stand-alone, and many people like to make mixins uninstantiable if the language underlies them support. Mostly the mixins depend on the presence of a particular agreement from the classes that inherit them. Otherwise, mixins are quite self contained.

Mixins may often be a specialization, so there is some overlap between mixins and multiple specialization. There is no rule to say which approach is generally more appropriate. Certainly, mixins need not be specializations. The advantage of mixin approach is that it encourages modularity of programming. Mixins help programmers to build more modular and reusable parts in applications.

5.1.4 Other Uses of Multiple Inheritance

Multiple inheritance can also be used in other areas. For example, Eiffel uses it as mechanism for module construct, whereas other languages need explicit syntax and semantics for module or package. Sometimes multiple inheritance is chosen for code sharing too, though many people argue it an undisciplined use.

5.2 Drawbacks of Multiple Inheritance

Coming with its power of expressiveness, the difficulties of multiple inheritance can not be neglected either. In fact, despite its advantages, multiple inheritance is not universally appealing, as demonstrated by its exclusion form languages such as Java, Smalltalk and Objective C. Some problems of multiple inheritance are listed in the following sections.

5.2.1 Name Resolution

A common problem that every implementation of multiple inheritance must address it how to handle name collisions which happen, for example, when properties with same name is inherited from multiple parents. We will discuss the issue of name conflict more detailed in Chapter 6 on page 29. Here we only have a brief overview on how such problems can be solved.

Some languages solve name conflicts with an arbitrary rule, such as the pre-order traversal of an inheritance tree, as in Perl or Python. More elaborated arbitration rules are the linearization used in CLOS or Dylan, which is essentially reducing the inheritance graph to a flat list. In other languages the name collisions must resolved explicitly by the programmer, for example, by qualifying the name as in C++ or renaming as in Eiffel. But there is no agreement on which way is most suitable.

5.2.2 Efficiency Issue

Multiple inheritance also cause more headaches for language implementor than single inheritance with respect to the efficiency issue—spatial or temporal. Some techniques used in single inheritance which is relatively space sparing could not used directly by multiple inheritance. Multiple inheritance may need more time in method lookups because of the possibility of name conflict, too.

5.2.3 Misuse of Multiple Inheritance

Another problem with multiple inheritance is that it can be overused. Some programmers may use it in an unclear and undesirable manner. But we should note that the issue of proper use of multiple inheritance itself is much debatable. For example, whether code sharing by multiple inheritance is appropriate. In general, multiple inheritance adds much complicity to an object-oriented system, thus is potentially easy to misuse.

Chapter 6

Name Collision in Multiple Inheritance

Name collision is a quite common and pervasive phenomenon in our real world. It is actually very intuitive to understand, nevertheless the phenomenon has in detail subtleties that we will try to examine more closely.

In our society different people may have the same name, might it be family name or given name, no matter they are relatives or not. Varieties of products may possess the same brand or mark. Different host in the Internet that situates all over the world may have same hostname. In many computer system there may be many files with the same name. In programming languages there are also different things which are denoted with the same name. We will not be able to enumerate all such facts and our aim here just to show that there are a lot of examples of name collision everywhere.

In the realm of inheritance, which is a essential feature of object-oriented programming languages, there is no exception either. Name collisions happen not only in multiple inheritance. It is unavoidable for programmers and designers of object-oriented programming languages.

Generally name collisions may cause problems or may not, which depend on many factors. We will in the following sections examine particularly name collisions in the field of multiple inheritance.

6.1 Classification of Name Collision

A good method of examining something is try to classify it and look at it from different directions, which has rooted far in human history. A famous example is Aristotle's attempt to classify the world in which we are living. Here we will also try to examine the phenomenon of name collision in multiple inheritance by giving it a classification [29].

Before we go into the detail of classification and comparison, we shall clear some terms which will be used later in the discussion.

By the term hierarchy, we mean structures that can be described by acyclic directed graphs (DAG's). The term class hierarchy is used to mean the hierarchy of all the classes in a given system and their superclass-subclass relationships.

The term inheritance hierarchy of a class with respect to a particular attribute of the classes is the part of the class hierarchy that covers the inheritance paths of that attribute.

6.1.1 Horizontal and Vertical Name Collision

One way to classify the name collisions is to look at the different possibilities where name collisions arise. Name collision can arise in two different ways depending on whether the collision arises because of the presence of more than one superclass or whether the collision is the result of ambiguities between the class itself and its superclasses. These two forms of name collisions can be named horizontal and vertical name collision.

Horizontal name collision

Horizontal name collision is used to mean collisions in which a class inherits several attributes with the same name from different superclasses. Figure 6.1 shows this situation. Note that horizontal name collision can not occur in tree structured hierarchies, i.e. in single inheritance hierarchies.

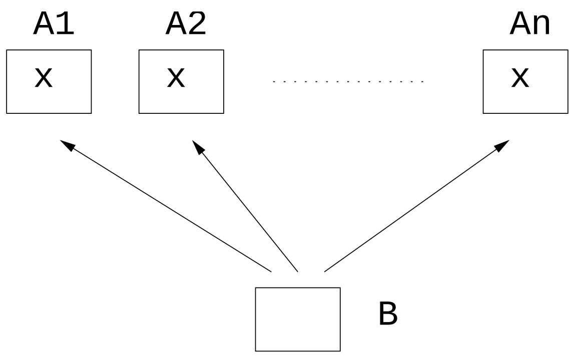

Figure 6.1: Horizontal name collision

Vertical name collision

Vertical name collision means that in a class definition there is at least an attribute with the same name of one or more attributes that is inherited from some of its superclasses. Figure 6.2 on the facing page sketches this kind of name collision.

6.1.2 Consequence of Name Collision

We have stated that name collisions may or may not pose problems, because there are different kinds of name collisions, which have different consequences. Before we try to solve the problems caused by name collisions, the real problematic name collisions should be found out.

There are essentially three different views on the outcomes of a name collision. Some name collisions may be just what is expected and wanted, and the

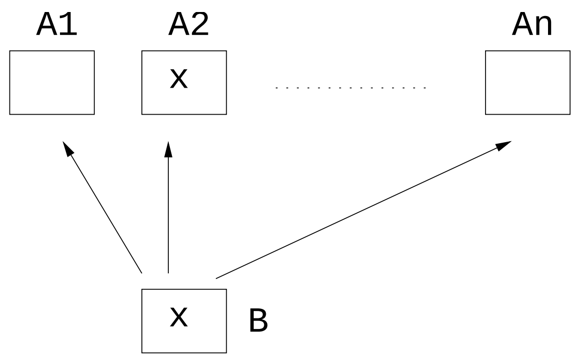

Figure 6.2: Vertical name collision

different attributes which collide described actually the same phenomenon. We say that such name collisions are intended.

On the other hand, there are also name collisions where different attributes with the same name describe in fact different phenomena. Such name collisions are unexpected and sometimes cause quite much surprise.

There are also name collisions which are considered illegal, because the relation between names, attributes and phenomena must be unique. Such collisions are usually not allowed by the definition of programming languages and cause the errors be reported by the compiler without waiting to be detected till runtime.

Intended name collision

When a name collision is regarded as an intended name collision, there is actually only one attribute this is concerned. Such an attribute may have several specifications (one for each inherited attribute, and possibly one in the class itself) which together must constitute the full specification of the unique attribute. If all name collisions are considered as intended in a programming language, the specifications must not be in conflict, in order to be able to ensure the polymorphic property of the class hierarchy.

Most of simple intended name collisions take place in form of vertical name collisions. For the example in Figure 6.2 if we know that the specification of `x` in `B` is a specialization of that `x` in `A2`, the polymorphic property can be ensured. Such situation occurs quite often in practice of object-oriented programming, because it is what programmers intend.

Intended name collisions can happen in the form of horizontal name collisions too. For example, in the class hierarchy in Figure 6.1 on the facing page, the attribute `x` of `A1`, `A2`, ..., and `An` might have a common superclass.

Casual name collision

If several attributes with the same name but the different and not necessarily related specifications, we say a casual name collision is occurred.

In this situation it is important to be able to distinguish between the different attributes by some other means than their names. The most immediate solution is to qualify attribute names with the name of the class from which it inherited, so that the qualified name is unique. For example in the Figure 6.2 on the page before we could denote the attribute `x` of `A2` and `B` as `A2.x` and `B.x`. (Certainly there is no necessity to use dot notation as shown here, the important point is to qualify attributes.)

With fully qualifying scheme as above, it is very possible that the notation will become clumsy quite quickly, so a short-hand may be useful. Horizontal and vertical overwriting serve this purpose. It should be noted that overwriting not be confused with specialization. And horizontal or vertical overwriting does not exclude the possibility of denoting the other attributes by qualification.

Horizontal overwriting means in Figure 6.1 on page 30 that it is possible to state that the `x` attribute of class `B` is `A1.x` which overwrite other `x` attributes in class `A2`, etc.. Vertical overwriting means in Figure 6.2 on the page before it is possible to state the attribute `x` of class `B` overwrite the attribute `x` of class `A2`.

Illegal name collision

A name collision is considered illegal, because the relation between attributes, names and phenomena must be unique. This means that name collisions will always give rise to compile-time errors.

For example, in one class it is not allowed that two attributes (data or methods) have the same name (not to be confused with overloading in some programming languages where overloading actually causes those names to be not same at all). Another intuitive example is that every two children of the same parent should not have the same name, where same name should have meant the same person.

6.1.3 Summary

From above discussion we see the issue of name collision has quite some subtleties which need much care, though in the face name collision seems very intuitive. The classification we have mentioned is summarized in Table 6.1.

	Horizontal	Vertical
Intended	specialization (e.g. common ancestors)	specialization
Casual	qualification, horizontal overwriting	qualification, vertical overwriting
Illegal		

Table 6.1: Classification of name collisions

6.2 Resolution of Name Collision

We have taken a look at the different forms of name collisions. In the following we will examine some ways to deal with name collisions.

The essential point of name collision resolution is that it needs the intervention of programmers. In other words, the programming languages can not always infer the intention of programmer. Certainly the programming languages can help the programmers to express their intention.

As we have discussed above, not all forms name collisions in multiple inheritance are necessarily problematic. Some of them are intended, provided by object-oriented programming languages as valuable mechanism and used by programmers to express themselves—specialization. Only name collisions that need to be concerned here are those that cause problems, i.e. those casual ones.

We must note that, in general, the difference between the intended and casual name collisions are only relative. Saying that some forms of name collisions are intended and other forms casual only makes sense under certain agreed criterion. For our discussion in this chapter, our focus is on the name collisions occurred in multiple inheritance of object-oriented programming languages. Thus, the name collisions caused by semantic specialization are considered intended and others are thought casual. Now we look at some means to solve the problematic name collisions occurred in multiple inheritance.

6.2.1 Renaming

Renaming is one natural way to solve the problems of name collisions. It is provided for example in the object-oriented programming language Eiffel [35]. The idea is simple, since there is a clash between names, then renaming some of them will solve the problem.

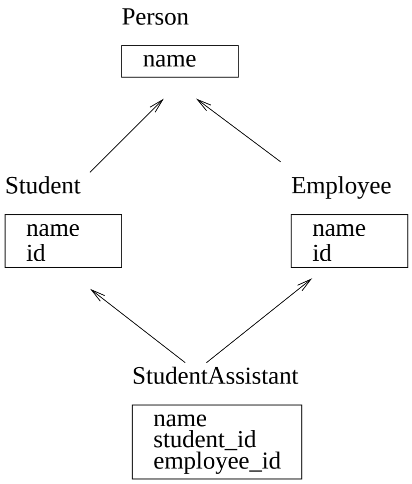

Figure 6.3: Renaming

An example will make its simplicity more clear. The example is shown in Figure 6.3. The class `Person` defines one property `name`, and the class `Student` and the class `Employee` inherit both from the class `Person` and add a new property `id`. Then the class `StudentAssistant` inherits from both the class `Student` and `Employee`. Now the problem arises, that there are name collisions about the properties `name` and `id` in the class `StudentAssistant`. The solution here is the programmer's intervention—renaming. Since the property `name` is thought a singleton, so it is actually intended name collision and needs not more handling. On the other hand, the property `id` from the class `Student` and

`Employee` has different nature. Therefore, in the class `StudentAssistant` they are renamed to `student_id` and `employee_id`.

6.2.2 Qualifying the Name

Since name collisions are caused by the same names, one resolution is to try to make those name themselves unique. We have seen renaming is one way to go, another similar one is trying to make the names a little longer so that they can differentiate themselves, by qualifying the names of concern.

A simple example is that it is quite common that two person have same given name, so how to tell them apart? If they have different family name, the problem is solved. Or even family name is the same, then other means must be exploit, for example, age, gender, home town, etc..

So if there is a name collision where two attributes from different classes which have the same name, in order to disambiguate them, some languages allow those attributes being qualified or explicit designated by their classes. By qualifying the attribute with its class the message lookup process will start to search this attribute from the qualifying class, therefore the conflict is resolved.

The programming language C++ uses such scheme. For example if two classes `A` and `B` both have a attribute called `x` and somewhere they cause name collision, the ambiguity can be resolved by qualifying the attributes with their class name, that is to say `A::x` and `B::x`.

6.2.3 Point of View Notion

Qualifying scheme mentioned above seems quite satisfying to resolve the problem of name collision, but it is not that such a way has no deficiency.

The weakness of qualification or explicit designation is just its explicitness. Explicitness in object-oriented programming can cause problems, because using qualification means constant information about the inheritance hierarchy is built in statically without taking into account of the possibilities that inheritance hierarchy may change later, which may cause unneeded update that must be performed in earlier code. In other words, explicitness makes the incremental development that is an often claimed benefit of object-oriented programming difficult.

In order to eliminate the shortcoming of qualification while at the same time retain its benefit, a new concept called point of view notion (PoV) is proposed [11].

Before we go into details of point of view notion, we will at first look at the problems caused by the explicitness of qualifying names in object-oriented programming languages.

Explicitness against virtue of object-orientness

One import consequence of object-orientness is its ability of specialization, which requires the specialized properties in subclasses to take priority when an object instance of a subclass receive the message corresponding to such a property. With the explicit qualification of message sending such requirement can easily be bypassed because of the possibility that message be qualified with superclass, of any depth. The associated properties will be searched starting from the specified class, despite any specializations in the subclasses.

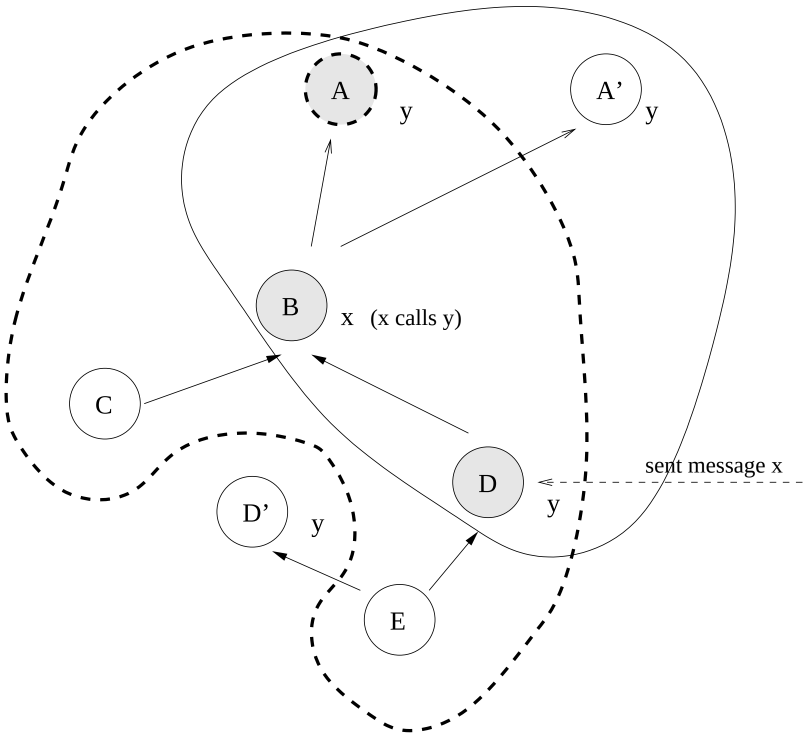

Figure 6.4: Point of view

Consider an example inheritance hierarchy shown in Figure 6.4, class `B` defines a method `x` that uses the method `y`. But there is a name collision because both methods `y` inherited from class `A` and `A'`. Suppose class `B` actually wants to use method `y` inherited from `A`. So we attempt to use the qualified message sending, for example `A.x`, and the name collision is resolved.

But later we add a new class `D` that inherits from `A` and redefined the method `y`. Now if an instance object of class `D` is sent a message `x`, the problem arises, because the newly defined method `y` will not be used. Using point of view notion the above problem can be easily solved.

Definition of the point of view notion

The point of view of a class C on an object o, $PoV(C,o)$, is the intersection of the dependent classes of the class C, $Dep(C)$, and the representation classes of the object o, $Rep(o)$:

$$PoV(C,o) = Dep(C) \cup Rep(o)$$

Where the dependent classes $Dep(C)$ is the set of all related classes, that is the union of all superclasses and subclasses of C, including C itself. And the representation classes $Rep(o)$ is set of all the superclasses of the class of instance object o, including the class of o.

Using the point of view

Now we can apply the point of view to our example in Figure 6.4 on the page before. Instead of using explicit qualification for `y` in the method `x` of the class `B`, we can declare message sending of `y` being from the point of view of the class `A`. Then if an instance object `d` of the class `D` is sent the message `x`, the $PoV(A, d)$ is the set of the classes `A`, `B` and `D`. So the method `y` from the point of view of the class `A` is one in the class `D`.

6.2.4 Sender Path Tiebreaker Rule

Sender path tiebreaker rule [12] is a feature incorporated in some earlier version of SELF implementation (later removed in favor of simplicity [40]) to resolve ambiguities when inheriting equally from two unrelated abstractions.

To resolve such name collisions, SELF's inheritance rules additionally specify that if two slots with the same name are defined in equal-priority parents of the receiver, but only one of the parents is an ancestor or descendant of the object containing the method that is sending the message, then that parent's slot takes precedence over the other parent's slot.

The sender path tiebreaker rule automatically (without the need of the programmer specifying any hints) resolves ambiguities between unrelated abstractions if the sender of the message is part of one abstraction but not the other.

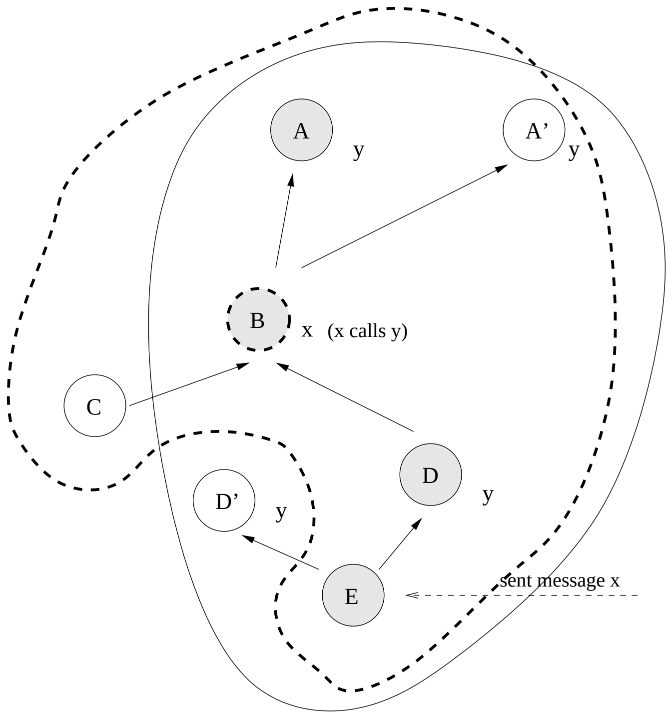

Figure 6.5: Sender path tiebreaker rule

Figure 6.5 on the facing page shows the sender path tiebreaker rule applying on our above example inheritance hierarchy. As we can see, the sender path tiebreaker rule has the same effect here as declaring the message sending of `y` in the method `x` of `B` from the point of view of the class `B` itself. The method `y` called in `x` is the one from `D`.

6.2.5 Linearization

Yet another technique to solve the problem of name collisions is linearization. Linearization techniques are most schematic and automatic among all approaches so far mentioned. If using renaming and name qualification can be thought as driving the car, where the driver decides where to go, choosing of linearization could be seen as taking a train that runs on tracks, where all routes are predefined by the whole railway system.

The linearization techniques are mainly used for inheritance in object-oriented problem languages based upon LISP: Flavors, LOOPS and CLOS and Dylan. They are also used in some other languages.

The main advantage is the fully automatic run-time name conflict solving without much programmer's intervention, which appears to be a quite good mechanism when no precise information on the object and its properties are available. Furthermore, on the basis of linearization more sophisticated techniques such as numerous modes of method combination in CLOS can be built.

On the other hand, the linearization techniques are also criticized for being nonintuitive and their implicit priority rules difficult for the programmer to understand.

In next chapter will look at the linearization in multiple inheritance in more detail.

Chapter 7

Linearization in Multiple Inheritance

In object-oriented systems with multiple inheritance, some mechanism must be chosen for resolving conflicts when inheriting different definitions of the same property from multiple superclasses. Some languages require manual resolution by the programmer, with mechanisms such as feature renaming in Eiffel and explicit designation in C++ .

Yet other languages such as many object-oriented descendants of Lisp, for example, Flavors [9, 25], LOOPS [42, 25], CLOS [37, 7, 25, 24], and Dylan [3, 39], automatically resolves conflicts occurring in method dispatch. This resolution is implemented using a linearization. When a class is created, a linearization of its superclasses, including itself, also known as the class precedence list (CPL) is determined, ordered from most specific to least specific. Then this linearization can be interpreted as a single inheritance hierarchy. When several methods are applicable for a given call, the one defined on the most specific class, according to the linearization, is selected.

Linearization is not only used in Lisp-like languages to resolve name conflict. For example, the object-orientation in Perl use a depth-first and left-to-right search algorithm for its method lookup, and this is actually one form of linearization.

Most object-oriented languages implicitly use a rule similar to linearization for method lookup in single inheritance: a class is more specific than any of its superclasses, so methods defined for subclasses override methods defined for superclasses. The problem with generalizing linearization to multiple inheritance is that the simple rule can not determine which one of two superclasses is more specific which have no subclass-superclass relationship between them.

In this chapter we will at first take a look at some characteristics that a linearization might be desirable to possess. Then we will discuss some known implementations of linearizations for multiple inheritance, apply the various discussed characteristics to those implementations and hope to be able to make the difference among those implementations more clear.

7.1 Characteristics of Linearizations

During the evolution of the linearization algorithm a number of characteristics have been identified as being desirable for linearizations [18, 19, 3]. Some of these characteristics are acceptability, monotonicity and observing local precedence order.

Before we go on to discuss these characteristics, we will at first clarify some terms. Suppose a class C in the class hierarchy H, we denote the inheritance hierarchy of the class C respect to the class hierarchy H as H_C. Then a linearization $\mathcal{L}$ of all superclasses of the class C, including itself, is denoted by $\mathcal{L}(H_C)$ or simply $\mathcal{L}(C)$.

7.1.1 Acceptability

An acceptable linearization is one in which only the shape of the class inheritance graph may be used in determining the linearization. In other words, an acceptable linearization must produce similar results on hierarchies which are similar in structure. All the linearizations discussed in Section 7.2 on the next page, which are all base on only the structure of the inheritance hierarchy, are acceptable.

One example of imaginary linearization which does not fulfill the acceptability is the lexicographical ordering over names of classes, because it is not related to the structure of the a given hierarchy. Then, changing the name of a class in the hierarchy could change the whole ordering and affect the behavior of its subclasses.

Formulated more formally: Let H_{C_1} and H_{C_2} be any hierarchies provided with precedence orders $prec_{H_{C_1}}$ and $prec_{H_{C_2}}$ respectively. Suppose there is such an isomorphism Φ from H_{C_1} to H_{C_2} that for all classes a and b of H_{C_1}, $a \prec_{prec_{H_{C_1}}} b$ iff $\Phi(a) \prec_{prec_{H_{C_2}}} \Phi(b)$. Then, for any linearization $\mathcal{L}$ which observes acceptability, $\mathcal{L}(H_{C_2}) = \Phi(\mathcal{L}(H_{C_1}))$, where $\Phi(\mathcal{L}(H_{C_1}))$ is the list obtained from $\mathcal{L}(H_{C_1})$ by replacing each class with its isomorphic counterpart in H_{C_2} by Φ.

7.1.2 Monotonicity

A monotonic linearization is one in which every property inherited by a class is inherited or defined by one of its direct superclasses. That is to say, an inherited property can not skip over all of the direct superclasses. This reflects the view of incremental specialization in inheritance in that the inherited behavior of a new class could be easily understandable from the behavior of its direct superclasses.

More formally, monotonic can be defined as follows: A linearization $\mathcal{L}$ is monotonic for the class hierarchy H if, for every C' the superclass of the class C, the linearization $\mathcal{L}(C')$ is a sublist of $\mathcal{L}(C)$.

Monotonicity implies that the linearization of a class must be an extension without reordering of the linearizations of all of its superclasses, which is a basic idea in Dylan's linearization algorithm that we will see a little later.

7.1.3 Obeying Local Precedence Order

Some of linearization-based strategies use an order on the direct superclasses of each class, the local precedence order. This order is usually given by the user when defining a class, and corresponds to the implementation inner representation of the hierarchy. When not explicitly mentioned, class A precedes class B in the local precedence order, if class A and B are both in the list of direct superclasses of some class and A is left to B in definition or in the figure.

A linearization that observes local precedence order will not produce a linearization which is inconsistent with the local precedence orders of any of the superclasses. That is, if class A precedes class B in the local precedence order of class C, the linearizations $\mathcal{L}(C)$ and $\mathcal{L}(C')$, all subclass C' of C, should have A before B.

7.2 Variate Approaches to Linearizations

As we have stated above that for linearizations in multiple inheritance there are some choices left to language implementors on how to decide which one of two superclasses is more specific between which there is no subclass-superclass relationship.

We will first examine simpler approaches then more complicated ones. This order is actually also consistent with the chronological order of those implementations realized. An example multiple inheritance hierarchy is used throughout this section and is shown beforehand in Figure 7.1.

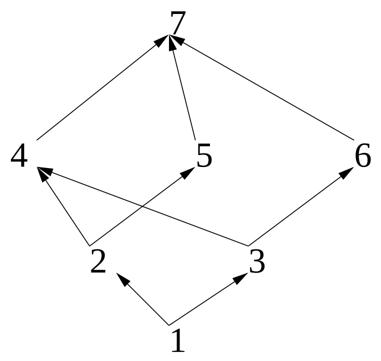

Figure 7.1: A class hierarchy example for linearizations

7.2.1 Depth-First Left-to-Right and Up to Joins

One simplest way to obtain the class precedence list of a class, i.e. the linearization of its all superclasses, including the class itself, is by depth-first and left-to-right (pre-order) traversal of inheritance hierarchy starting from the concerned class and removing the possible duplicates from the result of traversal.

If there are not duplicates in the result of traversal, the linearization will be unique. On the other hand, if there do exist duplicates in the traversal result, which is, for instance, the case when calculating the class precedence list of the class numbered 1 in our example multiple inheritance hierarchy, some criteria must be followed to remove the duplicates. So the rule can be summarized as: “Depth-first, left-to-right, and up to joins.” The result of traversal starting from class numbered 1 is:

$$(1, 2, \underline{4}, \underline{7}, 5, \underline{7}, 3, \underline{4}, \underline{7}, 6, \underline{7}),$$

where duplicates are marked by underlines.

The duplicates, such as class numbered 4 and 7, arise because of they are joins in the graph and have more than one subclasses. When the inheritance hierarchy is traversed using depth-first and left-to-right order, they are visited more than one once.

Linearization in Flavors

The linearization in Flavors can be seen as a depth-first, left-to-right traversal of the class's superclasses graph, removing all but the *first* occurrence of any class that appears more than once. So in our example the linearization of superclasses of the class numbered 1 is:

$$\mathcal{L}_{Flavors}(1) = (1, 2, 4, 7, 5, 3, 6).$$

The problem of the Flavors' approach is that in some hierarchies a superclass may have precedence over its subclass, which is not consist with the concept of specialization and can be quite counterintuitive. In our example, the class numbered 7 precedes the class numbered 5, 3 and 6.

Linearization in LOOPS

The linearization in LOOPS can be seen as a depth-first, left-to-right traversal of the class's superclasses graph, removing all but the *last* occurrence of any class that appears more than once. So in our example the linearization of superclasses of the class numbered 1 is:

$$\mathcal{L}_{LOOPS}(1) = (1, 2, 5, 3, 4, 6, 7).$$

The linearization in LOOPS takes more attention to the right superclass-subclass relationship, but it sometimes does not observe the local precedence order. For example, the class numbered 5 precedes the class numbered 4.

7.2.2 Topological Sort

Another way to get the linearization is to calculate the topological order of all superclasses of the class concerned.

A topological order is a linear ordering of the nodes of a graph such that, for any two nodes, i and j, if i is a predecessor or j in the graph (we call it a constraint and denote it by a pair $i \prec j$), then i precedes j in the result linear ordering.

An algorithm that topologically sorts the all superclasses of the class concerned, including the class itself, is straightforward. We begin by listing out a class in the inheritance hierarchy that has no predecessors. We then delete this class and all links leading out from it. We repeat these two steps until either all the class have been listed, or all remaining classes have one or more predecessors and so we can not remove any of them, which means the inheritance hierarchy is inconsistent and have no linearization possible.

But the result of topological sort might not unique, because there are circumstance where more than one classes without predecessor according the constraints, here lies the freedom for the implementations.

Linearization in LOOPS

Here we reexamine the linearization in LOOPS using the algorithm of topological sort.

The linearization in LOOPS can be performed above by topological sorting on the all superclasses of the classes concerned, including the class itself. All of the superclass-subclass relationships together build up the constraints. They are:

$$1 \prec 2,\ 1 \prec 3,\ 2 \prec 4,\ 2 \prec 5,\ 3 \prec 4,\ 3 \prec 6,\ 4 \prec 7,\ 5 \prec 7,\ 6 \prec 7.$$

When ambiguity arises, i.e. there are several possible choices for the next class of the linearization, the class that has a direct subclass closest to the end of the partially sorted result sequence is selected. If this can not result in a unique candidate, the local precedence order will be consulted to resolve the ambiguity.

For example, after we take out the class numbered 1 and the partially sorted result sequence is (1), both the class numbered 2 and 3 have not subclasses any more. And both classes have a direct subclass 1 which is the last element in the partially sorted result sequence. The class numbered 2 must be chosen because it precedes the class numbered 3 in 1's local precedence list. After we take class numbered 2 and the result computed so far is $(1, 2)$, both class 3 and 5 have not subclass any longer. The class numbered 5 must be chosen because it has a direct subclass 3 which situates most closest towards the end of the partially sorted result sequence. And so forth, we get the same result of linearization as by traversal $\mathcal{L}_{LOOPS}(1) = (1, 2, 5, 3, 4, 6, 7)$.

Here it is more clear why the linearization of LOOPS can sometimes disrespect the local precedence order, because they does not carry the same priority as those superclass-subclass relationships. Local precedence orders are only used as the last resort and can not bypass the depth-first principle.

The linearization of LOOPS can sometimes be non-monotonic too. In our example, $\mathcal{L}_{LOOPS}(2) = (2, 4, 5, 7)$ and $\mathcal{L}_{LOOPS}(3) = (3, 4, 6, 7)$. For the class numbered 1, the class numbered 5 is more specific than the class numbered 4, which contradict with the linearization $\mathcal{L}_{LOOPS}(2)$.

Linearization in CLOS

Linearization in CLOS uses the same topological ordering with the same disambiguating rule as that in LOOPS. The difference to LOOPS is that except of the constraints taking place in LOOPS, CLOS also obeys the local precedence order at the same time and priority.

The local precedence order defines a total ordering on that class and its direct superclasses. For example, the local precedence order of the class numbered 1 defines the total ordering $1 \prec 2,\ 2 \prec 3,\ 3 \prec *$, where $*$ represents some system defined class which is superclass all other classes in system. This rule adds new elements to our constraints as shown for LOOPS[1]. They are

$$2 \prec 3,\ 3 \prec *,\ 4 \prec 5,\ 5 \prec *,\ 4 \prec 6,\ 6 \prec *.$$

Another way to view the constraints is that it is the union of all local precedence orders. In order to calculate the linearization of all superclasses of a class, including the class itself, all local precedence orders of those classes together build up the constraints. Those additional links caused by local precedence order is shown in Figure 7.2 as dashed arrows.

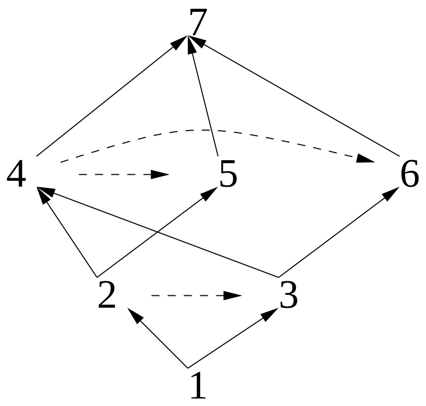

Figure 7.2: The same class hierarchy example with local precedence shown

Now using topological sort and the given constraints we can get the linearization of all superclasses of the class numbered 1, including the 1 itself:

$$\mathcal{L}_{CLOS}(1) = (1, 2, 3, 4, 6, 5, 7).$$

The linearization of CLOS will always be consistent with the local precedence orders. But it is not always monotonic, although that is not the case in the above example.

Let us look at another example shown in Figure 7.3 on the facing page:

$$\mathcal{L}_{CLOS}(B) = (B, D, F, E),$$

$$\mathcal{L}_{CLOS}(C) = (C, F),$$

$$\mathcal{L}_{CLOS}(A) = (A, B, D, E, C, F).$$

Quite clear it is not monotonic.

Linearization in Dylan

Dylan originally specified a linearization equivalent to the one used in CLOS. In light of the research done in [18, 19], and the definition of the monotonicity

[1]Some constraints shown up in LOOPS are here not necessarily any more, because they become redundant and can be inferred by transitive closure on the other constraints

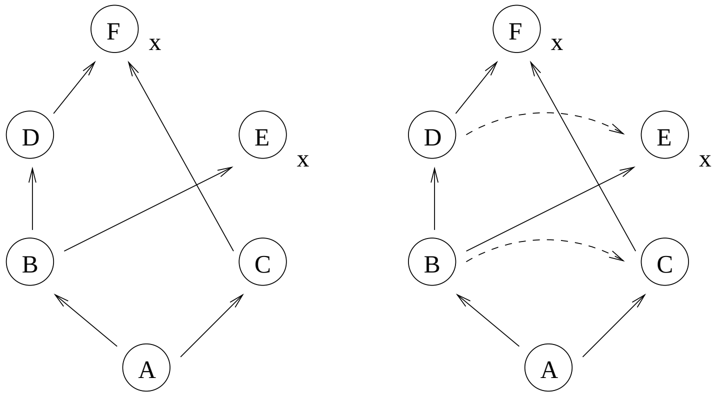

Figure 7.3: Another example inheritance hierarchy that demonstrates monotonic characteristic of linearizations

criterion, it was decided that the CLOS linearization should be replaced by a monotonic linearization.

The linearization used in Dylan merges the local precedence order of the class being defined with the linearizations of its direct superclasses. The CLOS linearization instead merges the local precedence orders of the class and of all its superclasses.

The first reason for the change is that a monotonic linearization seems to match more better the intuitions of programmers about how inheritance and linearizations work. With a monotonic linearization, it is easier to understand the behavior of classes when multiple inheritance is involved, largely because behavior of instances of a class can be explained in terms of the direct superclasses.

A second reason for choosing a monotonic linearization for Dylan is that it enables more compile-time method selection. Dylan has an mechanism, known as sealing, for describing what extensions to a library are possible after it is compiled. A sealed class may not be directly subclassed outside of the library where it is defined, whereas an open class may be. Similarly, methods may be added outside the defining library to an open generic function, but not to a sealed one. A domain of a generic function can be sealed so that it prohibits the definition of new methods or classes which would change the applicability of methods for the types specified by the domain. (For details on sealing, see the Dylan Reference Manual [39].) The restrictions imposed by sealing permit a Dylan compiler to select methods for generic function calls at compile-time without imposing a closed-word assumption on the program.

For our example in Figure 7.2 on the preceding page, Dylan's approach has the same result as CLOS. For the second example shown in Figure 7.3, all constraints are $A \prec B$, $B \prec C$ from local precedence, $B \prec D$, $D \prec F$, $F \prec E$ from linearization of B, and $C \prec F$ from the linearization of C. Then we get the linearization of A:

$$\mathcal{L}_{Dylan}(A) = (A, B, D, C, F, E),$$

which obeys local precedence orders and is monotonic.

Chapter 8

Efficient Message Dispatching in Multiple Inheritance

In this chapter we will address generally the problem of efficient message dispatching mechanisms in object-oriented programming languages with multiple inheritance.

The efficiency of message dispatching involves two important ingredients: the time factor and the space factor, which at the same time conflict with each other, as shown in Figure 8.1. Not unlike other fields of automatic computing, the efficiency is only a compromise between the factors of time and space. There is not such a efficient solution that is both time optimal and space optimal. Both time and space are not infinite for computation. The pragmatic tradeoff must be made. But where to draw the line of compromise is a pragmatic issue of respective concern.

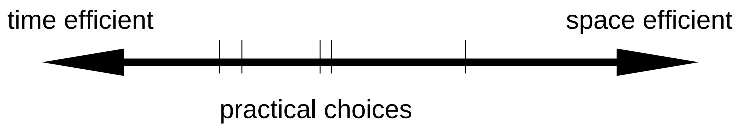

Figure 8.1: Time and space efficiency

In our interest—the efficiency of message dispatching with multiple inheritance, the aim is that the message dispatching, which in the object-oriented languages has long be a central issue and a field of struggle, should be performed as fast as possible while at the same time as spare as possible in space. As the different tradeoff made, there exist understandably diverse approaches [16].

8.1 Dispatch Table Search

Dispatch table search may be the most simple and intuitive approach of message dispatching. What involved for dispatching a message is simply "searching for the selector in the dispatch tables in the linked list of shared structures" as in [14]. In this context, the mentioned linked list of shared structures is just generalized term for the inheritance hierarchy.

Saying it intuitive is because it is just a vivid mimic of what really happens frequently in our daily life. We do some work, but suddenly we need something else in the meanwhile as a requisite, so we stop the work at hand, go to do the needed work, which might require other unexpected work, and in the end we go back to the original work and make it done.

In an object-oriented program, when an object receives a message, it searches its table of methods, whether the table is in the object itself as in object-based programming languages such as SELF, or in its class in class-based languages such as Smalltalk, if there it can not find the corresponding method of this message, it goes on looking further up to its parents' method dispatch tables, until either a suitable method is found or unexpected situation occurs such as multiple choices or no applicable methods at all.

As the efficiency of dispatch table search is concerned, it represents a extremity of the tradeoff made, i.e., it is most space efficient but at the same time it is also the slowest.

There is no space overhead beyond that information held by minimal method dispatch table, which is needed anyway to hold the methods of an object or a class. Although dispatch table search is most economical in space, the time overhead is not. The time overhead can be not only substantial, it is also proportional to the depth of the inheritance. This can sometimes intolerable because of the large depth of the inheritance, especially in large applications.

Nevertheless, dispatch table search is used in many places because of its simplicity, for example, during prototyping or providing basis for comparison. Such situation should not be ignored, particularly when considering the time efficiency from a more wider perspective, that involves not only time measure in real execution but also that of development.

8.2 Selector Table Indexed

In the above section we have gone to one extreme of tradeoff between time efficiency and space efficiency. We now look at the other end of rod.

As noted in [14], the other extremum stresses the time efficiency at the cost of space efficiency.

> It involves using additional space attached to each shared structure. Each table lists the implementation of each of the messages recognized by this kind of object, indexed by the selector code.

We call this extreme the selector table indexed approach. It is most fast because for all possible messages that exist in the system, the corresponding methods are already at hand, and there is no need to lookup further in the inheritance hierarchy.

Consequently the drawback is its huge space requirement, sometimes it is even unrealistic in large applications. Selector table indexed practically prescribes that every object in object-based programming languages or class in class-based programming languages must have a global view of all applicable methods it may receive. To put it simply, the space requirement has the order of multiplication of number of classes and number of selectors. Thus, if 100 classes with total 500 selectors in system and each method (pointer to method) occupies one word in the table, 50 000 words will be need just for the selector tables.

The space inefficiency is not the only problem of selector table indexed. Selector table indexed causes inevitably much duplication as the side effect of its explosive space needs. Then another problem arises.

Duplication makes the consistency difficult. When there are new methods added in or old ones removed from the system, which is in dynamic object-oriented programming languages not at all seldom, selector table must be updated. Selector table updates cost also time which is just contrary to purpose of selector table indexed. Therefore selector table indexed is even more unsuitable for the dynamic object-oriented programming languages than for their static friends.

Most interest in selector table indexed lies in its theoretical significance, for example, as a comparison means to other approaches.

8.3 Techniques in Between

Neither of the both techniques discussed in the above sections is very useful because of the mentioned drawbacks. Most object-oriented systems thus choose some way lying between those two extremes.

In many object-oriented systems, especially for static typed object-oriented programming languages, the more practical techniques are created by means of reducing the space requirement of selector table indexed approach. For example, using the coloring algorithm from graph theory to reduce the number of selector designators in the dispatch table, using better data structures such as sparse array to represent the selector table, trimming out the unused cells in dispatch table, or sharing the entries of dispatch table, etc.

There are also many other object-oriented systems, especially for dynamic typed or weak typed object-oriented programming languages, which choose the some combination of caching (memoizing) and searching (lookup) often called cached inheritance search. Cache techniques are based on the locality principle and mostly implemented using hash.

Essentially all that these techniques have in common is the compromise of time efficiency and space efficiency, they just approach their goals from different directions. As we can see, the selector table with compact representation is not very different from the cache. So we can say all these in-between techniques are just some cache techniques.

8.4 Cache Organization

Caches contain the references to the method. There are several ways of organizing the caches.

8.4.1 Cache per Class

One possible way of organizing caches is that every class (or every object in object-based languages) possesses a caches as shown in Figure 8.2 on the following page. Selectors are used to index into the caches.

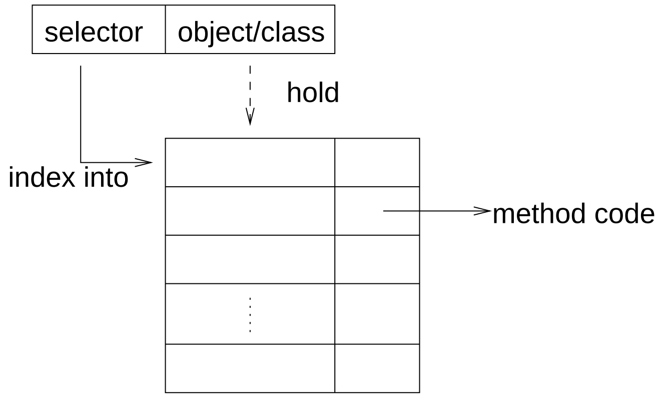

Figure 8.2: Cache per class

8.4.2 Cache per Selector

The orthogonal viewpoint consists of caches owned by selectors as shown in Figure 8.3. Classes are used to index into the caches. This organization is usual in languages which have complex method handling (multi-methods, method combination, etc.) like PCL [26] implementation of CLOS.

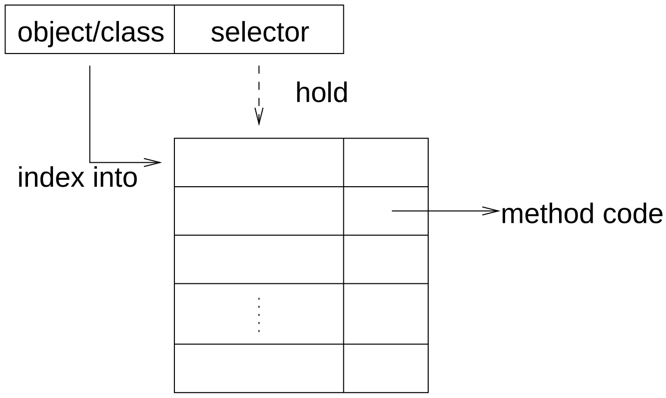

Figure 8.3: Cache per selector

8.4.3 Cache in One Table

The third possibility of cache organization is a large table shown in Figure 8.4 on the facing page, which collates all dispatch information in a single area. This can facilitate the dynamic modifications of the caches.

8.4.4 Inline Cache

Yet another ways to utilize the cache to speedup the dispatching is inline cache, which takes a very different approach as above discussed three ways of cache organization.

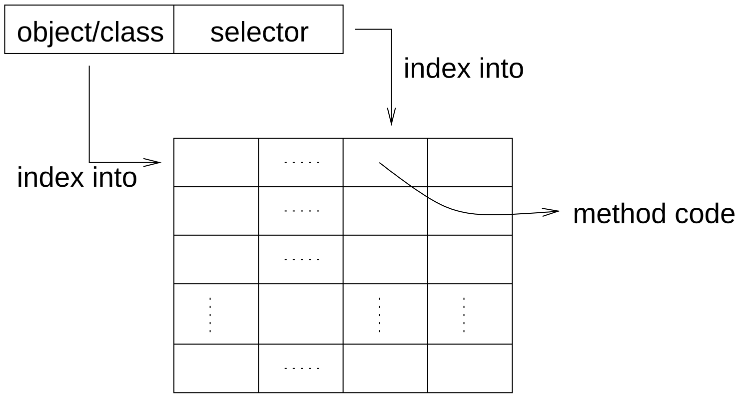

Figure 8.4: Cache in one table

The already mentioned three forms of cache organization can all be seen as lookup caches, in contrary to the inline cache. All those ideas of cache mechanisms come from the fact that sending a message in an object-oriented system takes longer that simply calling a function in static address.

Lookup caches reduce the overhead of message passing. A lookup cache maps the receiver type (class) and message name (selector) pairs to methods and holds the most recently used lookup results. When messages are send, first the cache will be consulted, if that probe fails, the normal (expensive) lookup mechanism will be used and the result be stored in the cache, possibly replacing an older lookup result.

Even with a lookup cache, every message sending still takes considerable amount of time because of the need to probe cache. Eliminating the need of consulting cache every time a message being sent raises the idea of inline cache (on the other hand, it might seem strange to call it inline cache, because its purpose is to avoid cache accesses).

At a given point of message sending in the code, the type of receiver rarely changes, i.e., if a message is sent to an object of some type at particular call, it is vary likely that when the next time the send is executed the receiver has the same type. This locality can be exploited by caching the lookup result—the looked-up method address—at the point of call, e.g. by replacing the call instruction. Subsequent execution of the message sending code can jump directly to the cached method, completely circumventing any lookup. Of cause, the type of the receiver must have be checked against the possible changes because of the polymorphic nature of object-oriented programming. This way of caching is called inline caching since the target method address is stored at the send point in the caller's code. The inline cache is sketched in Figure 8.5 on the next page.

Inline caching has been proven quite effective by many implementations of Smalltalk and SELF systems [22].

8.5 Compacting Selector Table

In this section we will look at roughly some ways of compacting the section tables. First some trivial ones then some more specific techniques. We will

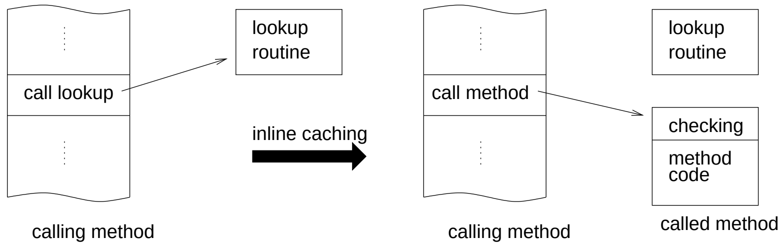

Figure 8.5: Inline cache

also use a simple inheritance hierarchy example to illustrate the process. The example is shown in Figure 8.6.

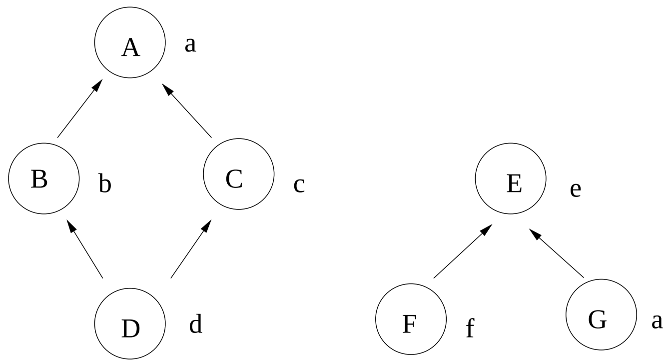

Figure 8.6: A simple inheritance hierarchy

Before any compacting techniques are applied, the selector table for this example is shown in Table 8.1 per definition of the selector table index approach. Where the bullets ('•') mean methods defined in respective classes themselves and circles ('∘') mean methods inherited from superclasses. Each row in the table represents all method applicable to a given class. For our example the selector table requires 42 cells of space and is 33 percent filled.

Class/Selector	a	b	c	d	e	f
A	•					
B	∘	•				
C	∘		•			
D	∘	∘	∘	•		
E					•	
F					∘	•
G	•				∘	

Table 8.1: Selector table without compact

8.5.1 Dispatch Table Trimming

The simplest compact technique is removing trailing empty entries from each row (here) for a given class of the dispatch. This reduces the space requirement of our example to 26 cells, an saving of 38 percent. But in real systems the saving may not be so impressive.

After trimming, the dispatch table can have different length for rows, therefore a simple boundary checking is needed to prevent indexing errors. Actually the checking code is needed only for message sends with an offset larger than the shortest row in dispatch table.

A similar technique is also used, where both trailing and leading empty entries are removed from dispatch table. Then another 8 cells of space can be saved.

8.5.2 Sparse Arrays

Using the trimming techniques or not there could be always many empty cells in the dispatch table. This is reminiscent of sparse matrices, so some researchers [17] have come to the idea of using sparse arrays to represent the dispatch table.

Given a large number of sparsely filled arrays (rows in our example), a mapping of these arrays into a master array can be found, where the indices of non-empty elements in the master array are unique. In Figure 8.7, an example of mapping is shown for our above example.

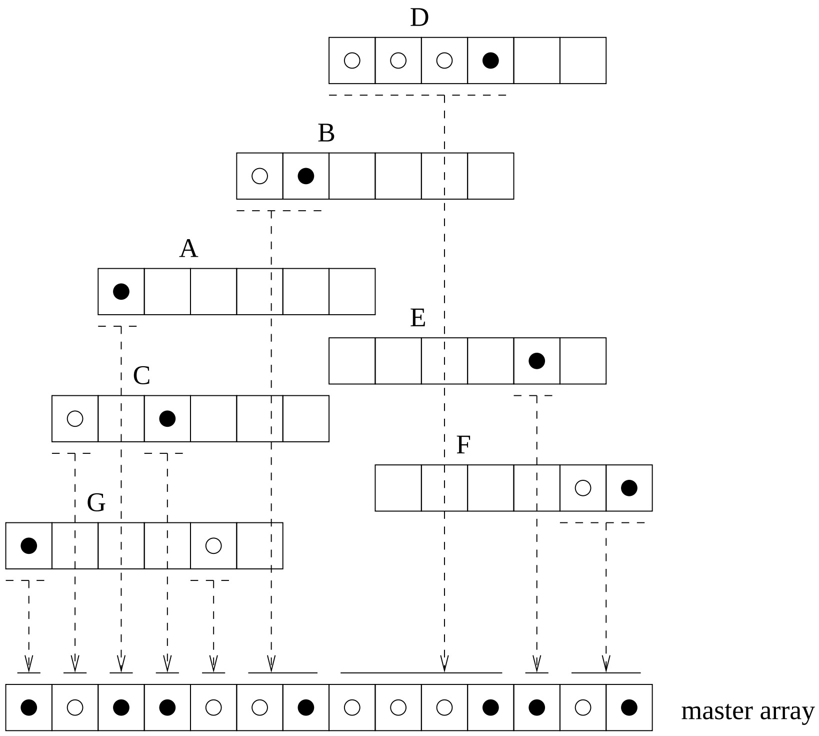

Figure 8.7: Mapping of sparse arrays into a master array

The master array is filled with rows of dispatch table like doing one dimensional jigsaw puzzle. Like a puzzle is here actually a puzzle, because there are two degrees of freedom: the particular coding of the selectors (how the each row in dispatch table looks like), and the order in which master array is filled by those rows. In our example of mapping with order of G C A B D E F, we have got the one of the optimal mapping. In the real systems that may be even impossible.

After master array is built, given an index r in the master array for the beginning of each row and a selector index s in the row, method can be retrieved in constant time with $r + s$.

But there is still one problem that somewhere there must exist information about whether $r + s$ really refers to the method that is interested. When a message is not understood by a class, so it should not have a reference on position $r + s$, but due to the packing together, proper message s' of another class with index r' in master array happens to be at the same position, i.e., $r' + s' = r + s$.

This ambiguity needs to resolved, and we need to be able to verify if the method referred at index $r + s$ is effectively owned by some class. One way of solving this problem is using array with the same length as the a master array, but contains information (for example, row number in the dispatch table) that indicates which class the corresponding reference in master array belongs to.

8.5.3 Selector Coloring

Another way to optimize the selector table size is selector coloring, first proposed in [16], also discussed in [2] and [17].

The main idea is as follows: given a coding of the selectors in the set S and a function $coloring(selector)$ is defined which maps the set S to a new set T, where the number of elements in T is far smaller than the number of elements in S and fulfills that for every two selectors s_1 and s_2 if they are understood by the same class, $coloring(s_1) \neq coloring(s_2)$. In other words, when two messages are understood by the same class, they are colored differently. The color can then replace selectors in selector table. This has an effect that in our example of selector table there will exist fewer columns and the selector table size will be reduced.

In order to accomplish the above requirement, we can at first build a conflict table for the selectors. In fact the construction of a good coloring fulfills translates into a well-know problem in graph theory—graph coloring, hence the name. The conflict table is a representation of the conflict graph whose nodes are selectors and whose edges determined by pairs of selectors (nodes) which are accepted to the same class. Two nodes that are connected can not be assigned the same color.

The conflict table contains one row for each selector used in the system. Each row of table is list whose elements are other selectors for which there is a class which accepts both the one of those selectors and the selector associated with this row. We enter the selectors of superclasses before those of subclasses into the conflict table, the reason will be soon clear. The more selectors are there in the system, the longer the time the building of the conflict table will take. For our simple example, we get the following conflict table shown in Table 8.2 on the next page:

a	b, c, d, e
b	a, c, d
c	a, b, d
d	a, b, c
e	a, f
f	e

Table 8.2: Conflict table

Next we need to assign a color to each selector so that no two selectors which are adjacent in the conflict graph have the same color. These allocated colors will be used later to index into dispatch table, and minimum numbers of colors are desired. The problem of determining this minimum number is computational difficult. Fortunately finding a coloring which is near optimal is not that difficult.

First assign a color to a node, then pick next node and assign it the minimum color that does not conflict with any adjacent node already colored. This simple scheme will actually find a minimal coloring if the nodes are chosen in the right order. One plausible order suggested is to order the nodes in descending order on the number of nodes it conflict or its degree.

Accidently the order in our conflict table shown in Table 8.2 fulfills this requirement already well. That is because classes of a object-oriented system are usually in such a form, that selectors in superclasses are more likely to cause conflicts. The result of coloring our example is shown in Table 8.3:

selector	a	b	c	d	e	f
color	0	1	2	3	1	2

Table 8.3: Coloring result

Now we only need four colors instead of six selectors, and the colored selector table shown in Table 8.4 is two columns smaller than original one shown in Table 8.1 on page 52.

Class/Color	0	1	2	3
A	•			
B	○	•		
C	○		•	
D	○	○	○	•
E		•		
F		○	•	
G	•	○		

Table 8.4: Colored selector table

Chapter 9

Polymorphic Inline Cache

As shortly discussed in Section 8.4.4 on page 50, an inline cache records the class of the last receiver object observed at the call site, and jumps directly to the implementation of the method for that class as shown in Figure 9.1. A method prologue validates that the dynamic type of the receiver matches the expected type. If this test fails, a slower method lookup forwards the call and rebinds the call site cache to the class of the current receiver.

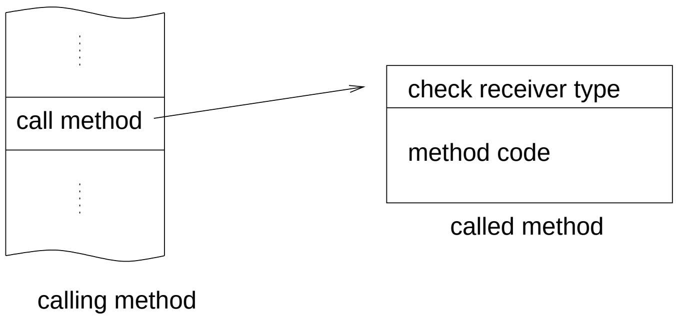

Figure 9.1: Inline cache after one send

9.1 Problem of Inline Caches

Inline cache is quite useful but itself also poses new problem that does not exist in normal lookup caches.

Using of inline caches is extremely effective only if the call site is monomorphic, or at least exhibits good temporal locality, but performs poorly at most polymorphic call sites. In other words, inline caches are effective only if the receiver type changes relatively infrequently at a call site. But when at some call site the receiver types switch back and forth among different calls with high frequency, significant time can be spent on the re-caching procedure itself. Worse, inline caching might even slow down message sends because of the overhead caused by cache misses.

This phenomenon is quite similar to the threshing problem that occurs sometimes in the virtual memory management of operating system. There are two factors that together cause the threshing phenomenon: space shortage and frequent changes of content in the space.

9.2 Enhancing Inline Caches

We have seen there are two factors together to cause the worsening behavior of inline caches, if we can prevent one of it occurring, the problem of inline caches will be solved.

Whether the receiver of a polymorphic call changes its type a lot or not can not be decided by the implementation of object-oriented system, it lies in the realm of application. So the only choice left for us is lessening the constraint of space for inline caches. We can cache not one reference to the real method code but all references to the possible real methods. This is the basic idea of polymorphic inline caches (PIC's), which were developed to overcome the weakness of inline caches.

If the requirement of the extra space for polymorphic inline caches is large, the benefit will be questionable. Fortunately in most applications the degree of polymorphism of sends seems to be either monomorphic (receiver type does not change), polymorphic (a few receiver types), or megamorphic (a lot of receiver types), so polymorphic inline caches can actually be quite practical in many applications.

9.3 Polymorphic Inline Caches

Polymorphic inline cache extend dynamic inline caching to handle polymorphic call sites, by caching for a given polymorphic call site all lookup results, instead of only the most recent one.

In a polymorphic inline cache, the call site invokes a dynamically generated PIC stub that executes a sequence of tests to see if the receiver object matches previously seen cases. If a match is found, then the correct target method is invoked; if a match is not found, a call to the system method lookup routine is invoked, which may in turn choose to extend PIC stub for the call site, in order to handle the new receiver type. Eventually, the stub will contain all possible references to code of methods, and there will be no more cache misses or lookups. As a result, a polymorphic inline cache is not a fixed-size cache, rather is an extensible cache where no cache item is ever removed or replaced by a newer item.

In polymorphic inline cache the stub can branch directly to the method's real code, i.e. skip the type checking in the method prologue, because the receiver type has already been verify in the stub. Methods still keep a prologue for type checking because they can also be used as in ordinary inline cache when the call is monomorphic. How a polymorphic inline cache would look like is shown in Figure 9.2 on the facing page.

A feature of any caching scheme is that it relies on temporal locality and thus cannot guarantee efficient dispatching for all cases. Polymorphic inline caches are less vulnerable than simple inline caches, but they still can perform poorly at megamorphic call sites. Sometimes cache mispredictions would be an issue even for a polymorphic inline cache on some programs.

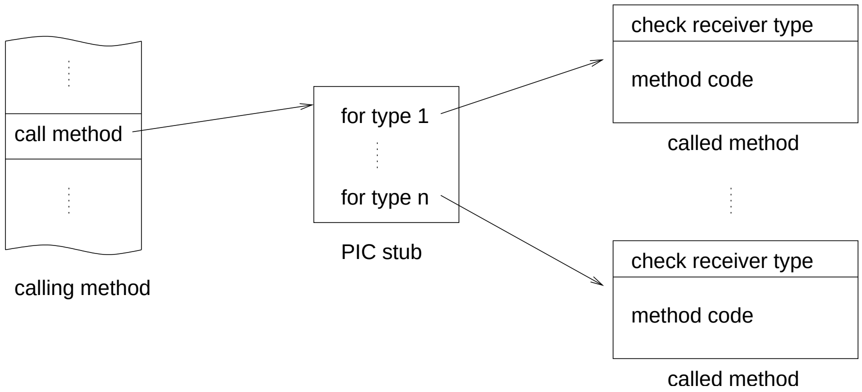

Figure 9.2: Polymorphic inline cache

9.4 Improving Polymorphic Inline Caches

We have seen the essential idea of polymorphic inline caches. In realization of a polymorphic inline cache, there can still be some practical decisions to be made.

9.4.1 Time Efficiency

As above described in the stub code the type check is done sequential, if the dynamic usage frequency of each type were available, stub code could be reordered periodically in order to move the most frequently occurring types to the beginning, reducing the average number of type tests executed.

If linear search is not efficient enough, more sophisticated algorithms and data structures can be used, for example binary search or some form of hashing. However, the number of types is likely small on average, so such optimization might not be worth the trouble.

As often been done by compiler, not only for object-oriented programming languages but also other traditional programming languages, inlining short function calls can save the time needed for the jumping to code. For polymorphic inlining cache some kinds of short method such as accessors can be directly inlined in the stub.

9.4.2 Space Efficiency

Polymorphic inline caches are larger than simple inline caches because of the stub associated with every polymorphic call site. If space is tight, a common polymorphic inline cache may be shared by several call sites which have identical message names. In this way, polymorphic inline cache would act as a lookup cache per selector, which is discussed in Section 8.4.2 on page 50. As the result of sharing it is possible the time efficiency would decrease because the number of types per PIC will increase due to the loss of locality.

We have also stated above that polymorphic inline caches never remove the items in its stub whereas simple inline caches always replace the old entry with the new one if different. But this scheme raises problems for megamorphic sends, where the polymorphic inline cache stub can grow unwieldy, which wastes space (might waste also time). Thus, there could be a limit for polymorphic inline cache where stub can not extend beyond a certain number of type cases.

Chapter 10

Design of Multiple Inheritance in dSelf

In the previous work [28] the object-oriented programming language dSelf was designed and implemented. dSelf is a derivative of the object-oriented programming language SELF and with extensions dealing with the distributed object-oriented programming.

Unlike the original SELF implementation, which is already a quite complete and complex environment, industry-strength, down-to-machine-code and possessing full-blown graphic user interface, our dSelf implementation is implemented in the programming language Java and still in its infantry, though already quite playable. Our goal is to improve our implementation step by step in order to make it more usable for us and other people, and learning experience through the work.

10.1 Background Information

Before we go into details of our design for improving the dSelf, we will first have a short overlook of the original design of dSelf implementation.

10.1.1 General Structure

dSelf implementation chooses a classic approach of interactive programming language environment, quite like those in many implementations of programming language Lisp, ML, Smalltalk, etc. That such a decision was drawn is a quite natural consequence of dynamic nature of dSelf language.

Such an interactive developing environment has the benefit of convenient use, more friendly to programmers and shorter developing and testing cycles, in contrast to the batch compiling.

An interactive developing environment does not necessarily mean a pure interpreter. In fact, under the cover of the dSelf read-eval-print-loop (REPL) is the dSelf compiler. This compiler can either be used implicitly by REPL or run by programmers directly from the outside of the interactive environment.

As we have stated, our dSelf system is implemented using Java. On the contrary to many programming system which use native machine code as the lowest representation of code, the lowest representation in our system is the Java class byte code. Using Java byte code means our implementation may not run as fast as those implementations which compile code to native machine code. On the other hand, it will run more easily on more architectures. Since the hardware will run more faster and the Java virtual machine will grow more mature too, our choices of using Java for our relatively simple implementation seems not unreasonable.

When dSelf source code is typed directly in the interactive environment, the code will be forwarded to the compiler, which parses the code text, generate Java class byte code and augment the environment, then the translated Java class is evaluated to get the desired result.

The dSelf source code can be gathered together into files and later loaded into the dSelf interactive environment by `_RunScript:`. In order to speed up the loading of source codes, those dSelf source files can be compiled from the outside of interactive environment into Java class files (with the help of a Java compiler), which can be loaded later into the dSelf interactive environment much faster.

10.1.2 The Compiler

The compiler of dSelf is created in the Lex/Yacc way with the help of JFlex [27] and CUP [23].

JFlex is a scanner generator quite similar to Lex but emit Java source code. CUP is a parser generator similar to Yacc and works with Java. Like Yacc, CUP also uses LALR(1) grammar to generate parser automatically.

When the compiler is called from the interactive environment, the expression typed in is first parsed, if no syntax error, the corresponding Java byte code in form of Java object is generated directly and linked in the interactive environment at once[1].

When the compiler is run as a stand-alone in order to compile some dSelf source files, a intermediate Java source file is used, which contains in textual form the same information as otherwise the byte code generated when compiler called from the interactive environment. In this way, we do not need to care the details of class file format and some other minor issues. Then this intermediate Java source file can be compiled using any Java compiler to generated class files, which can be later loaded into the interactive environment much faster than loading the dSelf source directly.

Here we give an simple example. Suppose a dSelf source file `Test.dsf` contains dSelf expression:

```
1 + 2
```

When compiled, an intermediate Java source file `Test.java` will contains (slightly edited to suit the text width):

```
import dSelf.*;
import parser.*;

public class Test extends dSelfProgram {
```

[1]Not forget our interactive environment is a normal Java application

```
  public CodeExpr getExpression(){

    return (CodeExpr)(new OrdinaryMsgExpr((CodeExpr)(new
        IntegerSO(1)), (String)("+"), (CodeExpr)(new IntegerSO(2))));
  }
}
```

10.1.3 User Interface

The dSelf interactive environment provides two forms of user interfaces (front-ends) to facilitate the communication between the programmer and system.

One form is a plain command line oriented interface where programmer interacts with the system, which is relative primitive, no history mechanism for example, but light-weighted.

The other front-end is graphic user interface implemented on the Java swing class library. This front-end is a little more friendly than above mentioned line oriented approach. It provides some other helpful facilities to improve the programming practice, for example, a history mechanism for recalling the earlier typed expressions and an object tree browser to explore the system.

10.2 Handling of Name Collisions

Because dSelf provides the possibility that an object can have more than one parent slots, many issues of multiple inheritance must be handled properly. One of them is the problem of name collision. As we have discussed in Chapter 6 on page 29, there are quite some ways to handle the issue of name collision.

10.2.1 Lessons from SELF

The earlier version of SELF implementation had the notion of prioritized parents[2] and exploited the sender path tiebreaker rule. After application of both rules if there are still irresolvable name collisions, such ambiguities cause error be reported.

Prioritized multiple inheritance

Prioritized multiple inheritance was introduced into SELF because of its use in the mixin style of programming. Prioritized parent slots make it possible to express the ordering information to the system, not unlike the local precedence order in system where using linearization, but it does not provide a total ordering like linearization. It is always possible to have parents with the same priority. So prioritized multiple inheritance in SELF provides only some degree of automation.

Prioritized parent slots also have other drawbacks. For example, it is possible to mask ambiguities between otherwise unordered parents, it can also cause

[2]In SELF parent slots are slots whose names end with asterisk. Asterisks are not part of the slot name and not used when matching a slot. Priority is indicated by the number of asterisks. The more asterisks a parent slot ends with, the lower the priority the slot hat.

the inheritance hierarchy inconsistent with respect to the ordering provided by priority.

Prioritized multiple inheritance, mixins, together with SELF's resends, dynamic inheritance may cause complex interactions which can raise surprising effects [12].

For example, if a mixin has ancestors, the slots of the mixin's ancestors, no matter how general, would override any slots in lower priority parents, no matter how specific, because mixin is normally inherited using higher priority parent slots. Another difficult situation arises because dynamic inheritance may cause parent slots to change between resends with a chain of resends, which would not be a problem in a system with single inheritance or unordered multiple inheritance, since the message lookup could always begin with the resending method holder's parents and proceed upwards.

Sender path tiebreaker rule

The sender path tiebreaker rule was included in SELF implementation, because it allows two disjoint objects (unrelated abstraction) to be used as parents with the same priority, without the intervention from programmer to resolve some possible casual name conflicts. But it may sometimes unintentionally mask the ambiguities that should be resolved otherwise.

10.2.2 Conclusion drawn

These two special rules were later abandoned along with other features in favor of the simplicity and minimalism [40]. Although these features had make it possible to directly handle some scenarios better, for example, mixin style of programming, they impose too much burden in all reasoning about programs and implementation of system. Adding features seems good, but every new concept burdens every programmer who comes into contact with the language.

As far as renaming is concerned, we think it is not that necessary in dSelf, because the dynamic characteristics of the language. Changing slot name is relative easy in an interactive environment, which might provides facilities to propagate changes. And consequently the dSelf language itself will be simpler.

Linearization does not bring dSelf more benefits either, because we have not decide to utilize the full automation advantages coming with the linearization, such as method combination. On the other hand, linearization will make the language more complicated and the linearization can sometimes quite nonintuitive.

As the result, from the beginning of dSelf's design, we have chosen the unordered multiple inheritance and without including the sender path tiebreaker rule. Certainly, the explicit resolution by programmer is needed if name conflicts occur.

10.3 Handling of Efficiency Issues

The price that the object-oriented programming language implementations must pay, in contrast to the implementations of the conventional high level procedural programming languages, is more or less the losses of the execution speed, because of more advanced and sophisticated features such as inheritance.

If only single inheritance is concerned and language is static typed, the solution will be relative easy and the speed penalty can be reduced quite significantly. One example is virtual table technique implemented in many C++ compilers, which handles single inheritance quite efficiently[43].

When facing multiple inheritance, the efficiency issue will get much complicated. In dSelf, which supports multiple inheritance, the situation is even complicated, because of its dynamic nature, pervasive use of polymorphic message passing and the ability of distributed computing.

In Chapter 8 and Chapter 9 we have discussed briefly some ways to improve efficiency in multiple inheritance. We now try to make our decision on how to improve the multiple inheritance in our implementation of dSelf.

10.3.1 Evaluation of Possible Solution

Primarily, there are essentially two categories of solutions to the efficiency issues in multiple inheritance. One is to build a method selector table as compact as possible, and build it at the compile-time then use it at run-time efficiently. The other is to build the cache at the run-time by and by, trying to capture and utilize the locality behavior hidden in the running programs.

Static selector table

A static selector table, mostly a global one, is more suitable for static and strong typed languages, where compiler can gather more information during the compilation. The compiler can then offload the cost of many decision making (method dispatching is a kind of decision making) from run-time to compile-time, thus improve the run-time efficiency. With the help of several compact techniques the selector table size can be reduced too.

But for our dSelf implementation, where the language is weak typed and extremely dynamic, and implementation goes the interactive environment approach instead of a batch compiler, static selector table seems not quite suitable for us.

Dynamic cache

On the contrary, dynamic cache approach seems more appealing. Dynamic cache parallels the dynamic nature of language more closely. So it is not surprising that many dynamic languages exploit some forms cache mechanism, for example, many implementations of Smalltalk, CLOS, etc.

Because dynamic cache is built during the running application, it does require certain run-time resource. But in order to improve efficiency in whole, such a price can not be avoided. Comparing to static selector scheme built at the compile-time, the dynamic cache approach is also not without its advantages, because it has more thorough knowledges about the run-time behavior of application, for example, some methods might not be used at all, or used only at the beginning, which are not easily inferred at compile-time.

10.3.2 Choice for dSelf

After comparing variate possibilities we decide the cache mechanism is used in our dSelf implementation. But there are still a decision between a centralized or distributed cache to be made.

Considering the distributed characteristics of dSelf language, there is no absolute suitable centralization, a distributed cache approach seems more proper.

In a class-based object-oriented programming language, a cache mechanism per class is thinkable. But dSelf is object-based, if we build per object cache, there would be too much waste, because many object in system are of transient nature, that is, they come and go unlike the classes which are more stable. That Every object carries a cache seems not acceptable.

The inline cache seems more appealing, actually the polymorphic inline cache is a more suitable approach to take, as already shown by the research done on SELF implementations.

Inline cache techniques are normally applied at the instruction level, for example, machine code. If we generate Java byte code directly, it will just like the normal case of inline cache. But our code generation in dSelf is at the Java object level and we do not mangle the byte code instructions directly, so we can not directly apply the inline cache techniques as in implementation of SELF, which generates machine codes.

So we choose a form of cache which draws idea from the polymorphic inline cache and is adjusted to suit our needs. Because code generation in our implementation creates Java objects that represent the expression of the computation done in compiled dSelf program, we decide that such expression objects can be extended to carry cached information.

In the end, our cache mechanism seems a mixture of lookup cache and inline cache. Like lookup cache is because we do not patch instructions directly and dispatching logic is not downloaded into the code as stubs. On the other hand, those expression objects generated is actually our lowest representation of computation in dSelf, so it is also quite like inline cache. Details of cache implementation will be discussed in next chapter.

10.4 Handling for Distributed Environment

A distinguished feature of dSelf is that it provides a distributed object-oriented computing environment. Once some different dSelf worlds, which may situated anywhere in the network, are connected together, the objects in the whole connected system work like in a single dSelf runtime from the application's point of view.

The facilities that enable the distributed computing in dSelf is built on the Java Remote Method Invocation (RMI). Every dSelf world functions like a RMI server at the implementation level. At the application level the distributed programming appears fully transparent.

The price paid for the distributed transparency is that when referenced objects residing in other dSelf worlds, the performance is inevitably affected, but it may not appears that obvious if viewed from application level, because transparency's sole purpose is concealment. This is another example of tradeoff.

But we try to minimize the negative effects of transparency. We think a cache mechanism could be build for that. Because the intercommunications happening among different dSelf worlds are actually message passing, which is in fact the sole means in dSelf to do computation, the cache mechanism discuss in previous section can be extended to cope with distributed situation.

Because we use the form of cache mechanism which is similar to the polymorphic inline cache, if generated codes (expression objects), for example, in a method activation, needed to be transferred, care must be taken for the cache carried by the codes. That means cache may also consume network bind width, so more sophisticated mechanism might needed here.

Chapter 11

Implementation of Multiple Inheritance in dSelf

After we have viewed briefly our design decisions in last chapter, we show now some details in the concrete implementation. Some class diagrams are shown too, which are presented using UML notation.

11.1 Necessary Changes

After we examined and valuated the original code base of dSelf implementation, we find that it has provided a quite good structure, but there are also some deficiencies which must be fixed first.

11.1.1 Handling of Non-Local Return

Non-local Returns are implemented in dSelf using Java exception. The implementation Java class `NonLocalReturnException` is used to represent the non-local returns in dSelf, which is shown in Figure 11.1. When a non-local return expression represented by `NonLocalReturnExpr` is evaluated, `NonLocalReturnException` should be thrown out. The key is when is appropriate to catch such an exception.

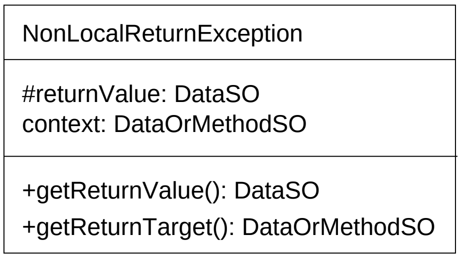

Figure 11.1: Class `NonLocalReturnException`

The essence of non-local returns is that it requires to return through a number of activations, the methods' activations or blocks' activations, and all activations in between. The original implementation in dSelf has overlooked the situations where a chain of activations in between may exist, for example, nested blocks, because it catches `NonLocalReturnException` unconditionally.

Our solution is making `NonLocalReturnException` to contain information about where it should be caught. The fix of non-local returns is quite important because it is used to implement control structures in dSelf, such as various types of loops.

11.1.2 Message Lookup for `value` et al.

In dSelf the blocks' code is evaluated by sending them `value` et al. messages. Because such messages can have variable arguments and the error handling for mismatch between the number of required arguments and that of actual arguments is different from that for normal message, they must be handled in the implementation like primitives.

The original implementation has it handled by specializing the method `dispatchOrdinaryMsg` in `BlockSO`. But its side effect is that `value` et al. must be sent directly to a block objects and normal message lookup process is bypassed, which is mostly not a problem because blocks are mostly used as literals, but shows its awkwardness when `self` is a block and `self` must be used explicitly. For example in the implementation of a loop construct `whileTrue:`

```
[] parent _AddSlots: (|
  whileTrue: = (|:b| [ value ifFalse: [ ^ nil ].  b value. ] loop).
|).
```

11.1.3 Slots in Method Objects

Originally the slots in method objects were not handled correctly. For example, the following method

```
  foo = (| a<-1 | a: a + 1)
```

when called repeatedly, the slot `a` will be incremented by one every time. The problem is solved by freshly initializing the slots of method objects every time when the methods are called.

11.1.4 Changes in Grammar

The grammar for dSelf is written in CUP, which understands LALR(1) grammars like Yacc or Bison but for Java.

The grammar itself for dSelf is changed a little in order to fix a precedence problem. An example illustrates the problem clearly:

```
        (1<2) ifTrue: ['do something...']
```

The parentheses around arithmetic is actually redundant according to the precedence rule of dSelf—Binary messages have higher precedence than keyword messages.

The problem lies in the precedence resolution for productions in CUP: "The precedence of a production is equal to the precedence of the last terminal in that production. If the production has no terminals, then it has lowest precedence." Thus, sometimes explicit precedence information needs to be specified manually.

Other changes are done to correct an error in code generation where wrong code is generated when the compiler is used as a stand-alone program to translate some dSelf sources. Elsewhere in code generation some unneeded codes are removed in order to make the generated code a little more efficient.

11.1.5 The `_Loop` Primitive

In the original implementation of dSelf the only way to do repeated work was recursion, which is quite costly. So we add a new primitive `_Loop`, which do unconditionally looping. With such a primitive, other iteration facilities can be built a lot efficiently.

11.1.6 dSelf World

The usability of the implementation lies not only in the implementation for language itself, but the extra functionalities implemented in dSelf world (like libraries in other languages) are also very important. Therefore, we tried to add more functionalities to the dSelf world when we need, for example, various loop constructs. Besides, some old errors in the dSelf world are fixed.

11.1.7 Miscellaneous Changes

There are also some other changes made, in order to make the implementation more robust and useful. For example, improved command line handling, more useful debug information, more graceful error handling, etc.

11.2 Enhancement for Multiple Inheritance

As we have discussed shortly in last chapter, that we choose a cache mechanism to improve multiple inheritance, which has ideas similar to the polymorphic inline cache.

The cache is carried by the dSelf expression objects of class `OrdinaryMsgExpr` which represents messages sent. The class diagram for `OrdinaryMsgExpr` is shown in Figure 11.2.

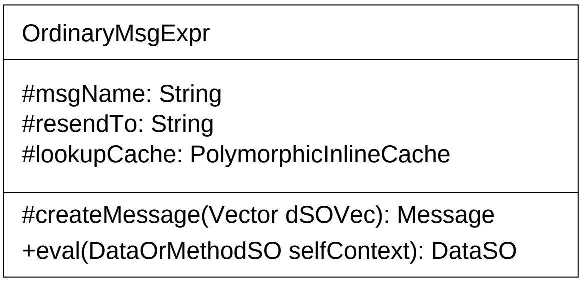

Figure 11.2: Class `OrdinaryMsgExpr`

The cache logic is performed in the implementation Java method `eval`, which consults the cache when executed. When an expression is already evaluated once, the chance of finding corresponding method in cache, thus avoiding more expensive lookup process, is quite high.

At an earlier try we had only saved the method corresponding to the last message sent. Though it took less space but the result is not good for many scenarios, because the receiver can actually change, sometimes to and from frequently. For example, the message `ifTrue:` and its friends are sent to a boolean object, which can be `true` or `false`, which respond to the message `ifTrue:` el al. differently.

In order to cope with such scenarios we decide to cache a certain number of methods most recently executed for an expression, i.e. our caches become polymorphic. The cache itself is represented by the class `PolymorphicInline-Cache` shown in Figure 11.3.

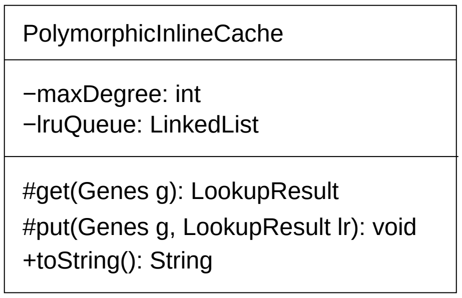

Figure 11.3: Class `PolymorphicInlineCache`

The problem we must face is that such a polymorphic cache need type information of dSelf objects. But in dSelf there are only objects and no classes that are normally quite natural representation of type information. We decide that every object must carry some extra information for its type.

Actually we do not need a rigid type system for the current implementation. We do not care that much about how the type itself of an dSelf object, for example the object `true`, really looks like. The most important is the classification; that is, all objects of one kind are classified together.

For this purpose, every dSelf object carries a simple classification information besides its other slots in its `SlotVector`. Such classification information is represented by the class `Genes`.

Genes is actually just a set of unique gene ID's which depends on the slots. The genes of an object change if the object or any of its ancestors has new slots added or old slots removed; thus, the changes of genes in an object must accordingly propagated properly. Update can be propagated because we decide every children must register itself to its parent.

Although the class `Genes` itself is quite simple and currently just a wrapper of a Java `TreeSet` class, we think adding such a class make our intension more clear and later the change of `Genes`'s representation easier.

Now which information should be saved in the caches? Our first thought is methods themselves; that is to say, the methods which are already executed once and their corresponding genes of receiver are kept together in polymorphic cache. It works well, but sadly only for method calls. Another large portion of expressions are data slot accesses, which have not been taken into account with regard to our cache mechanism.

Therefore, we need to cache data objects too. But data objects are not like methods, in that methods themselves seldom change and their result or side effects are what are interested, whereas if data objects do not change, how would they be useful. Hence, data objects themselves can not be cached because they will soon become obsolete; we need to cache the objects that hold the data slots. As the result, we decide that we package the result of lookup process, may it be a data object or method, and its holder object together, which is represented by class `LookupResult` shown in Figure 11.4.

LookupResult
−holder: DataOrMethodSO −value: dSelfObject
#getHolder(): DataOrMethodSO #getValue(): dSelfObject #getCategory(): int +toString(): String

Figure 11.4: Class `LookupResult`

With `Genes` and `LookupResult` the cache is actually implemented using a LRU (Least Recently Used) queue, where the key is `Genes` and the value corresponding to the key is `LookupResult`. The queue is organized per least recently used principle; that is, most recently used entries will situate most towards the beginning of the queue.

Chapter 12

Finale

In this thesis we have discussed our approach to improve multiple inheritance in our dSelf implementation. Because the time allowed for this work is confined to six months, our improvement is only limited and there are certainly more possibilities to meliorate the whole dSelf system. In the following sections we list some areas where the further work can be made to better our dSelf system.

12.1 Concurrency and Parallelism

In dSelf system, unlike the original SELF implementation, there are no concepts of processes or threads. Including such facilities in the dSelf implementation seems quite appealing.

Although by providing the distributed computing environment we provide one mechanism to enable the concurrency and parallelism in dSelf, such means may not be suitable for some circumstances. For example, the finer control over the concurrency may be desired, whereas distributed facilities in dSelf stress the transparency and the control is relatively coarse.

When concurrency and parallelism is exploited more deeply, there are surely more special cases to be concerned. For example, the issue of synchronization will be important and should be examined more thoroughly, which in turn could influence the distributed environment.

12.2 Extending the dSelf World

A bare dSelf system provides only a minimal initial environment, where only primitive objects and methods exist. All other important constructs and facilities need to be loaded sometime later. At present there is a script named `smallWorld.dsf` that plays this role. This script is already gradually extended. In order to made our dSelf implementation more usable, we need a even more elaborate and complicated dSelf world, where more facilities can be provided to the user to help them concentrate on their own task and not reinvent the wheel once and again.

12.3 Reflection

Reflection in programming languages is a complex subject in its own. There are much research done in this field, for instance, for languages Lisp and Java. We will not go in detail here, but we mention some ideas that may be interesting for dSelf.

12.3.1 Mirror Objects

The idea of mirror objects is not new. It is included in the original SELF implementation, but still not implemented in dSelf due to other more important work elsewhere and time limit. The requirement of distributed computing in dSelf complicates the matter further.

Mirror objects can be used to manipulate the objects themselves, which may be very important for a system with complex developing environment. For example, the method objects in dSelf will evaluate its code and return results when evaluated. If it is the method objects themselves not their impacts that need to be manipulated, mirror objects is one way to go in dSelf.

12.3.2 Open Implementation

Another idea is that the implementation of dSelf itself could be more open, so that the users could adjust some aspects of language system to their needs. For example, users may want to modify the method lookup process for some special purposes, or they may want to add their own primitives for reason of performance. The possibilities are numerous if the system is open. But it is not without its price, the system will sure more complex, which may not be desired.

12.4 Sealing

Sealing, an idea used for example in Dylan [39], is a mechanism to improve system efficiency by user intervention. Java has a `final` keyword that serves some similar purposes.

Sealing in Dylan declares that classes or functions will never be used in particular ways, or will never be extended in particular ways. These sealing directives enable a range of compiler optimizations, and also clarify the programmer's intent.

The idea of sealing may be also useful for dSelf, for example, to express the intension from the users about inheritance, about distributed behavior, etc.

12.5 Modules

Modules can help a lot for structuring large applications otherwise could be tangled masses. When programs grow large, adding modules to dSelf could bring benefits. How modules could be added to dSelf without mangling the elegance of original ideas underlying SELF would need much thorough thought.

12.6 Interfacing with Other Languages

Although dSelf is implemented in Java, it is still impossible to access Java objects now. An interfacing facility which connects dSelf world with remains of world is undoubtedly desirable. We will not raise concrete proposal here, but a well-thought interface with other languages which exists harmonically with dSelf world is worth its own examination.

12.7 Optimizing the System

Last but not least, it is never a bad idea to optimize our dSelf system. The hardware will be faster, the Java virtual machine will be more efficient, but optimizing of our own system should not be neglected. We have tried to improve the multiple inheritance in dSelf, but itself may still need to be more eyed. No doubt, other areas could be optimized, too.

Appendix A

SELF and dSelf—The Language

In this chapter we will take a short tour through the object-oriented programming language SELF, and also give a concise introduction for our baby dSelf.

A.1 A Tour Through SELF

SELF is an pure object-oriented programming language, that was designed by David Ungar and Randall Smith in 1986. It is quite close to Smalltalk in look and feel, for it has borrowed syntax heavily from Smalltalk. Everything in computation is an object, and slots in an object are of unrestricted type. It uses blocks, objects behave like closures, to implement control structures, such as conditionals and loops. Programs in SELF are constructed incrementally within the programming environment, which encourages an exploratory style of programming.

On the other hand, SELF differs from Smalltalk in several important respects, because the underlying principal of SELF is the desire for simplicity and concreteness. As the consequence, SELF is considerably simpler than Smalltalk in syntax and semantics. SELF has no notion of classes, instead an object-based inheritance mechanism is provided, which facilitates the direct manipulation of objects by users.

A.1.1 Objects

Objects are the fundamental entities in SELF, even control are handled by objects—*blocks*, which are SELF closures used to implement user-defined control structures.

Objects contain *slots* that are name-value pairs. Slots contain reference to other SELF objects. When a slot is chosen as result of *message send* and *message lookup*, the object in this slot is evaluated. How one object is evaluated depends on its type, because objects in SELF serve more than one roles. Variant types of objects have different purposes in SELF, and have different semantics, even different syntax when used as literals. There are mainly three kinds of objects, namely data objects without code, method objects and block objects (methods and blocks in short) both with code.

Object literals are represented by delimited parentheses for data objects and method objects, or by delimited square brackets for blocks. Within the parentheses or brackets, an object contains a period or dot ('.') separated list of slots delimited by two vertical bar ('|'), and optionally followed by the code to be executed when the object is evaluated if it is a method or block. The code for an object is a list of *expressions* separated also by period. A period at the end of list is optional. A last expression in the code for an object may be preceded by a circumflex ('^'). For example:

```
(| slot1. slot2 | 'some code' print. ^ true)
```

is a method object, and

```
[| slot1. slot2 | 'some code' print. 'more code' print)
```

is a block object. An empty object can be denoted by either '(| |)' or '()'.

Slots

An object can have any number of slots. When a slot contains a *data object*, it is called *data slot*. Otherwise, a slot contains a method object and it is called *method slot*. Some data slots have special roles: *argument slots* are filled in with the actual arguments during a message send, and *parent slots* specify inheritance relationships.

Argument slots are like normal data slots but with a colon prepended. The prefix colon is not part of the slot name and is ignored when matching the name against a message.

Parent slots are indicated by a trailing asterisk ('*'), which is also not part of the slot name and is ignored when matching the name against a message. Except for their special meaning during the message lookup process, parent slots are exactly like normal data slots, and they may be even assignable, allowing dynamic inheritance. Argument slots can not be parent slots.

Data objects

Data objects are objects without code. A data object returns itself when evaluated. The slots in an object may be read-only or read-write, which have different syntax: an immutable slot is denoted by an equal sign ('=') and an mutable slot is denoted by a left arrow ('<-') in the object. For example, in the object

```
(| x = 0. y <- 1 |)
```

the slot x is read-only, whereas the slot y is read-write. This object has in fact three slots: x, y and y:. The slot y: contains a method object taking one argument, which is in this case the new value for y. This special method object referenced by y: is a primitive called assignment primitive.

A slot containing the assignment primitive is called an *assignment slot*. When an assignment slot is evaluated, the argument to the message is stored in the corresponding *data slot* (without the trailing colon) of the same object and *receiver* is returned as the result.

In SELF there is no separate assignment operation. Instead, assignments to data slots are message sends that invoke the assignment primitive. As for above example, that the data slot `y` is assignable is because there is a slots in the same object with the same name appended with a colon (here `y:`), containing the assignment primitive. Hence, assigning `-1` to `y` needs to send the message `y: -1`. In this way, it is indistinguishable from a message send that invokes a method, clients do not need to know if `y` or `y:` cause data slot access or method invocations.

Method objects

Where distinguishes the method object from a data object is that it has code, while on the contrary a data object does not. Evaluating a method object causes its code be executed and the resulting value returned, not itself be returned as with a simple data object.

A method object can have *argument slots* and (or) local data slots. A method object also always has an implicit parent argument slot `self`, which plays a important role in the semantics of method execution.

More precisely, when a method object is evaluated as the result of message send, following steps are performed:

1. The method object is cloned, creating a new *method activation object*.
2. The clone's `self` parent slots is initialized to the receiver of the message.
3. The clone's argument slots, if any, are initialized to the values of corresponding actual arguments.
4. The code of the method object is executed in the context of this new activation object.

For instance, the following SELF code is evaluated:

```
(| a <- 0. foo: = (| :x | a: x + 1) |) foo: 1
```

shows scenario sketched in Figure A.1 on the next page.

Block objects

Generally speaking, a *closure* is a data structure that holds an expression and an environment of variable bindings in which that expression is to be evaluated. When invoked with actual arguments, executes the expression in the environment of this closure.

Blocks are SELF closures, which are used to implement user-defined control structures. A block normally consists of a parent slots pointing to the object containing the shared behavior for all block objects. The code in the block is evaluated to get a *block method activation object*. The block method activation object is quite similar to the normal method activation object, except that it has not a `self` parent slot but an anonymous parent slot that is initialized to point to the activation object for the lexically enclosing block or method.

The block method object is evaluated by sending it messages like `value`, `value:`, `value:With:` or `value:With:With:`, with respect to blocks having

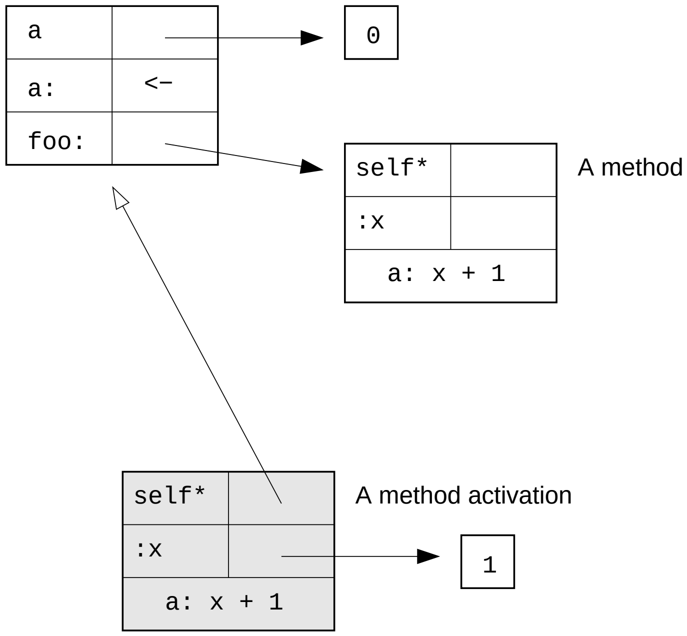

Figure A.1: Method evaluation and method activation object

zero, one, two or three arguments. For blocks with more arguments, appropriate message is just like above but extended by *enough* `With:`'s. But not like the normal message sending, more actual arguments is allowed to be sent to block, which are just simply ignored.

Block evaluation needs roughly two steps. In the first step, a block object is created and given a pointer to the activation object for its lexically enclosing environment. In the second step, the block method activation object of this newly created block object is generated with its anonymous parent set to above mentioned environment pointer. As an example, Figure A.2 on the facing page shows these two steps for the following code sample:

```
(| a <- 0. bar: = (| :x | [a: x + 1] value) |) bar: 1
```

Returns in methods and blocks

The last expression and only the last expression in the code of an object may be preceded by a circumflex ('^') operator, which denotes a *return*. A return causes the value of the given expression to be returned as the result of evaluation.

For normal method objects return operator has no effect at all, because an ordinary method object always returns the value of its final expression in any case. But in block it does play a important role, that it causes the control to be returned from the ordinary method containing that block immediately, regardless of how many method activations or block method activations lying in between, which are all abandoned. Such a return is called *non-local return*. To consider a simple example:

```
baz: = (| :x | (x isOk) ifFalse: [^ 'so bad']. 'good stuff here')
```

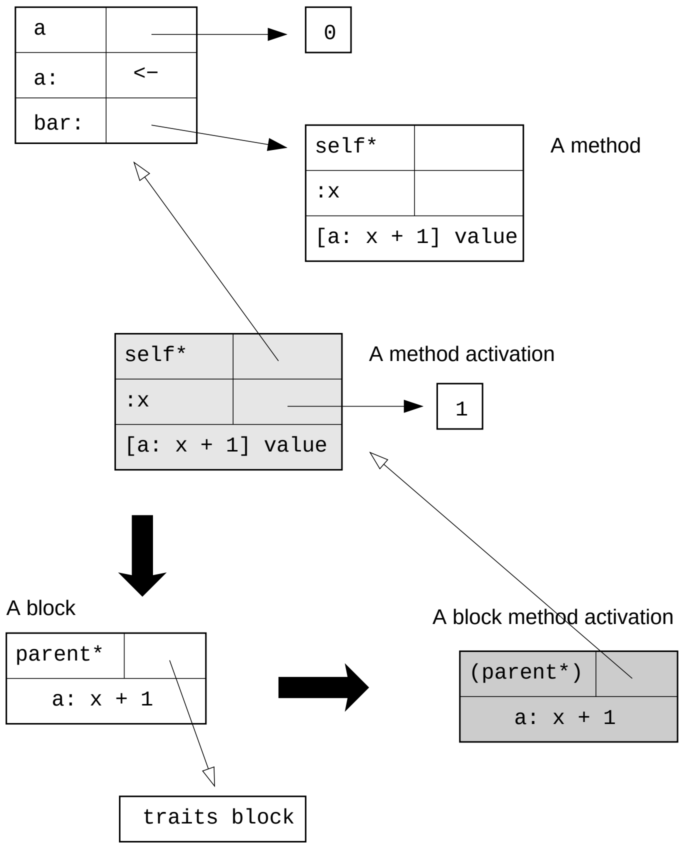

Figure A.2: Block evaluation

If `x` is not okay, `'good stuff here'` will not be executed at all, because non-local return of `'so bad'` causes method `baz:` to return immediately.

A.1.2 Messages

The message syntax in SELF is quite similar to Smalltalk's. There are three kinds of messages inSELF: unary messages, binary messages and keyword messages. By sending message to receiver all expressions can be formulated. The receiver of message can be explicit or implicit.

Unary messages

A *unary message* has no argument. It has the highest precedence and is written as an *identifier* following the receiver. Unary messages are evaluated from left to right.

Identifier is a sequence of letters, digits and underscore('_') and must begin with a lowercase letter or an underscore. The case of the identifier is significant. Identifiers beginning with underscores are reserved for primitives, identifiers `self` and `resend` are also reserved.

Some examples of unary messages:

```
'Hello, world!' print
['for ever' print] loop
```

and

```
tom uncle nephew
```

is the same as:

```
(((tom) uncle) nephew)
```

Binary messages

Binary messages are mostly used to express mathematical operation, in order to make the SELF expression looks more similar to the syntax used widely in schools. Binary messages have lower precedence than unary messages.

A binary message has one receiver and one argument, which are separated by a binary operator. For example,

```
-1 << 1
2 * 10
```

Binary messages have no associativity, except between identical operators, in this case it is from left to right. So,

```
1 + 2 + 3
```

is interpreted as

```
(1 + 2) + 3
```

But

```
1 + 2 - 3
```

is illegal.

Keyword messages

A *keyword message* may have one or more arguments and be written as sequence of one or more keyword argument pairs. First keyword must begin with a lower case letter or underscore, following keywords must begin with capital letter. Such sequence of keyword argument pairs are parsed as longest as possible. An initial underscore means it is a primitive, like in unary messages.

```
lobby _AddSlots: (| |)
[| :x. :y | x + y] value: 1 With: 2
```

Keyword messages associate from right to left and have the lowest precedence among three kinds of messages.

SELF runtime

The implementation of SELF provides a runtime environment. This runtime environment offers some basic primitives, which can not be implemented or efficiently implemented by user, and some organizational aids served as guidelines in certain degree. The runtime environment may also provide a graphical user interface to facilitate the user's development and smooth the learning curve.

Normally in the SELF runtime environment, which already contains some objects after initialization (these objects in the whole is often called a SELF world), there is an object named `lobby` that serves as an entry point. An exemplar `lobby` may seem as follows:

```
lobby = (| traits = ...
           globals* = ...
           mixins = ...
        |)
```

In `traits` there are objects that describe shared behavior. Typically, each prototype object has a parent with the same name here. Slot `globals*` contains prototypical objects and is itself a parent slot to make these prototypes easy to access. Slot `mixins` provides room for some parentless objects, which contain mostly simple single purpose methods and are designed to be mixed into some other objects later.

A.2 dSelf—A Distributed SELF

dSelf is an implementation of derivative of the object-oriented programming language SELF in Java, with the ability of distributed programming built-in. Objects in dSelf work together seamlessly, regardless of the runtime environment where they are situated. After the different runtime environments are connected by means of Java RMI (Remote Method Invocation), they behave in all just like a larger runtime environment.

A.2.1 How to connect

Distributed in dSelf means that slots can contain object that exists in other dSelf runtimes. Before different dSelf runtimes can communicate with each other, they must be connected. dSelf provides two primitives for this purpose:

- `_AddSlot:` *slotname* `ConnectedTo:` *URL*
- `_AddSlotIfAbsent:` *slotname* `ConnectedTo:` *URL*

The *slotname* is the door to another dSelf runtime environment. After invoking one of these two primitives *slotname* contains a remote reference pointing to the `lobby` object in the other runtime.

Every instance of dSelf runtime environment has a name, which must be specified by user at the start of runtime and unique on the same host computer. The host name together with runtime name uniquely specifies a dSelf runtime. *URL* serves just this purpose and must be a string in the following format:

'//*hostname[:port]*/*runtime_name*'

The port number is optional, which is defaulted to 1099. An example is

```
'//localhost/another'
```

So in all, if user wants to connect a runtime named "local" to another on same host named "remote", she can add a arbitrary named data slot somewhere in "local", for example in the `lobby` of "local" as follows:

```
lobby _AddSlot: 'remote' ConnectedTo: '//localhost/remote'
```

A.2.2 Primitive and ordinary dSelf object

In dSelf there are two kinds of objects: primitive objects and ordinary objects. A remote reference is created if and only if its target is an ordinary object. Primitive objects are treated differently, since they are always copied when accessed from another runtime.

The reason for the introduction of primitive objects comes from the efficiency consideration. Because it is much more efficient to access an object locally (in the same process address) than via a remote reference across the boundary of runtime environments, which may situate far away from each other.

Now assuming there are two runtimes called "local" and "remote" and shown in the Figure A.3 (for simplicity the corresponding assignment slots are not shown). In the runtime "remote" slot `x` contains a primitive integer object `34` and slot `y` contains an ordinary object. After connection like the example shown in the end of Section A.2.1 on the preceding page, a remote reference is created in runtime "local", as shown in Figure A.4 on the next page.

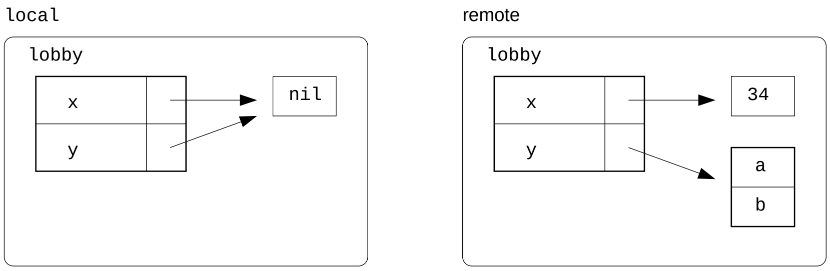

Figure A.3: Two dSelf runtimes before connection

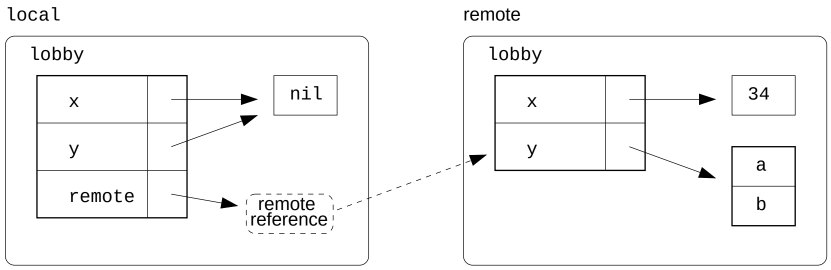

Figure A.4: Two dSelf runtimes after connection

Now we try to update the slots `x` and `y` in the runtime "local" to use the corresponding slots in runtime "remote" by sending them messages:

```
lobby x: remote x
lobby y: remote y
```

And Figure A.5 shows the new state. We can see that primitive integer object `34` is copied, but the ordinary object pointed to by slot `y` in runtime "remote" is not and a new remote reference object is create in runtime "local" pointing to it.

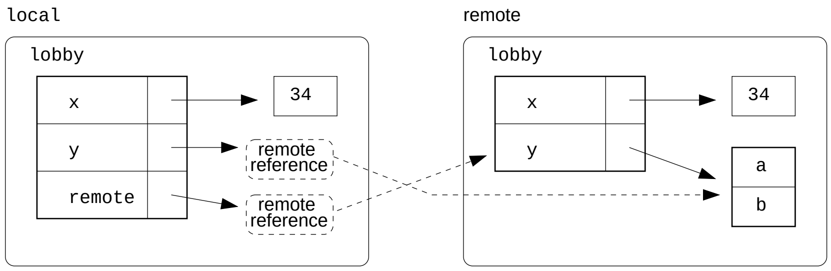

Figure A.5: Two dSelf runtimes after updating slots in "local"

Appendix B

Common Lisp Object System (CLOS)

In CLOS real world objects are represented by *classes* and *instances*. These CLOS objects are manipulated using *generic functions*. When generic functions are called, the dispatching mechanism of CLOS arranges for the appropriate implementation—the *method*—to be invoked, based on the classes of the actual arguments.

The structure of an instance is prescribed by its class. Each instance contains a set of named *slots*, where it keep its state information. All instances of a given class have the same structure and the same behavior.

One or more methods together builds up the implementation of a generic function. The methods are selected or dispatched according to the class of arguments.

A class can be built on the base of other classes by means of inheritance. It then inherits both slots and methods from its superclasses.

First we will look at generic functions and methods, which also plays an important role in objects creation in CLOS.

B.1 Generic Functions and Methods

Generic functions and their methods together provides the user the interface and implementation to the computation taking on the CLOS objects.

B.1.1 Generic Functions

Operation on the instances is provided by the generic functions. From the caller's point of view, a generic function appears exactly like an ordinary Lisp function. When a function is called, caller does not need to know whether the function is defined as an ordinary function or as a generic function.

When comparing the workings of ordinary functions and generic functions, we find there do be clear semantic differences. An ordinary function definition specifies both the interface and the implementation of the operation it performs, as shown in Figure B.1 on the next page.

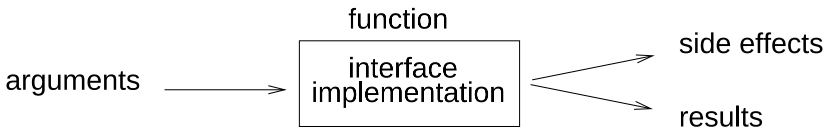

Figure B.1: Normal functions in Lisp

A generic function specifies only the interface. The implementation of a generic function is not included all in one place, but it is distributed across a set of methods. Whereas the implementation of an ordinary function does not change from call to call, the implementation of generic function does vary, depending on the classes of its arguments. Figure B.2 shows that a generic function can have several separate implementations—methods.

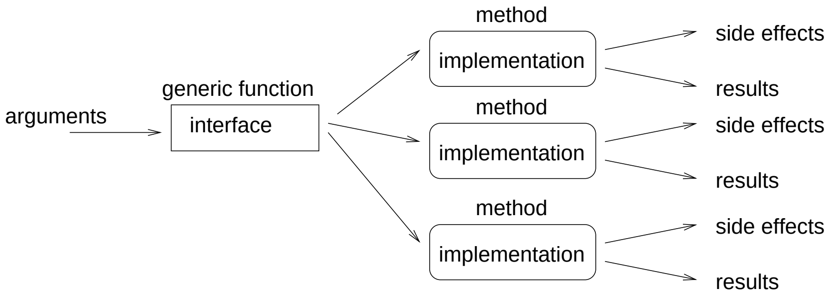

Figure B.2: Generic function and methods in CLOS

B.1.2 Methods

It is methods that actually perform the real computation and provide the functionality to their generic function. Like ordinary Lisp functions methods take arguments, return result and perhaps create side effects such as producing output. But unlike ordinary Lisp functions, methods are not called directly, they are only called by the generic dispatch procedure.

A method is attached to the generic function that it implements by stating in the method's lambda-list the classes of arguments the method handles. The method is called only if the arguments to generic function are of the appropriate classes. There is not necessarily only one method for a certain arguments combination that the generic function accepts. When a generic function is called, the arguments may select an implementation consisting of more than one method, because there may be more than one methods for the same generic function, which have the same lambda-list but play different roles.

B.1.3 Method Roles

CLOS makes it possible to split up the work or implementation of a generic function by means of methods, which differentiate themselves by the classes of the arguments. But that is not the whole story. Sometimes there is an implementation of a generic function which is so complicated or convoluted to be dealt in one single method that it is worth of splitting it again up. These resulting split methods have same lambda-list, in other words, they may be all called by the generic dispatch procedure when their generic function is called. But how can they be differentiated and in which order should they be called?

The answer to the questions is that they have different roles, which is defined by the method combinations.

In the standard CLOS method combination the bulk of the work is done by the *primary method*. Sometimes, one primary method performs all the work of generic function for a particular set of arguments. In other cases, where we consider just now, the work of primary method can be split up. The primary method can be assisted by auxiliary methods, such as *before-methods* and *after-methods*.

Before-methods are called before the primary method and they can do set-up work in advance of the other methods, for example, some extra checking. After-methods are called after the primary method and can do clean-up work or any other computation. The primary method can return the values of the generic function. Before-methods and after-methods are intended for side effects only. They are not called for value, and their return values if any are simply discarded.

B.1.4 Multi-Methods

Most of object-oriented programming languages do not provide multi-methods like CLOS. In these languages methods can only specialize one parameter, mostly the receiver of message. Many object-oriented programs can be written with this style of specialization without awkwardness. However, sometimes it is useful or more natural to write methods that specialize more than one parameter. These are called multi-methods. Multi-methods are intended for operations whose implementation truly depends on the type of more than one argument.

B.2 Classes and Instances

A Class is a Common Lisp type that defines the structure and behavior of its set of objects. These objects are called instances of the class. Each instance of a class shows the same structure, behaves the same and carries the same type information.

B.2.1 Slots

The structure of an instance is in the form of slots. A slot has a name and value. The value describes the state of slot. this state information can be accessed (read and/or written) by *accessors*. Accessors are one kind of generic functions.

Slots in CLOS can be divided into two categories: *local slots* and *shared slots*. For local slots, every instance maintains its own value for the slot. For shared slots, all instances of a class share a single value for the slot. Local slots are used more frequently.

B.2.2 Defining Classes

In CLOS defining a new class or redefining an existing one the macro `defclass` is used. The syntax of `defclass` is

`defclass` *class-name* ({*superclass-name*}*)
  ({*slot-specifier*}*) {*class-option*}*

class-name A non-nil symbol naming this class.

superclass-name A non-nil symbol naming a direct superclass of this class.

slot-specifier Defines a slot of the new class. Either a symbol naming the slot, or as a list containing the name of the slot followed by zero or more slot options:

slot-name
(*slot-name* {*slot-option*}*)

The *slot-option*s are listed here:

`:reader` *reader-function-name*
Defines methods for a reader generic function. Then the reader named *reader-function-name* can be used to get the value of this slot.

`:writer` *writer-function-name*
Defines methods for a writer generic function. If the *writer-function-name* is a symbol, the writer can be called such as (*writer-function-name new-value instance*). If the *writer-function-name* is a list such as (`setf` *symbol*), the writer named (`setf` *symbol*) can be called with `setf` syntax, such as (`setf` (*symbol instance*) *new-value*)

`:accessor` *reader-function-name*
Defines methods for a reader and a writer generic function. Then the reader named *reader-function-name* can be used to get the value of this slot, and the writer named (`setf` *reader-function-name*) to set the value of this slot.

`:allocation` *allocation-type*
Specifies if this is a shared or local slot. There are two choices: `:instance` for the local slot, which is the default; and `:class` for the shared slot.

`:initarg` *initarg-name*
Specifies an initarg for the slot. Then it can be used in the call to `make-instance` as a keyword to initialize the value of the slot when making an instance.

`:initform` *form*
Gives a default initial value form for the slot. *form* is evaluated each time it is used to initialize the slot, and in the lexical environment in which `defclass` was evaluated.

`:type` *type-specifier*
: Specifies that the contents of the slot will always be of the type named by *type-specifier*. But CLOS does not guarantee error checking when the value is stored in the slot.

`:documentation` *string*
: Specifies documentation for the slot.

class-option This option applies to the class as a whole. They can be as follows:

`(:documentation` *string*`)`
: Specifies documentation for the class.

`(:default-initargs .` *initarg-list*`)`
: Specifies default values for the initargs. *initarg-list* is a list of alternating *initarg-names* and default initial value forms.

`(:metaclass` *class-name*`)`
: Specifies class of class, therefore metaclass. The default metaclass is `standard-class`.

Now we show a simple example of using `defclass`: a 2-dimensional `point`.

```
(defclass 2d-point ()
  ((x :accessor 2d-point-x :initarg :x :initform 0
      :documentation "x coordinate")
   (y :accessor 2d-point-y :initarg :y :initform 0
      :documentation "y coordinate"))
  (:documentation "This is a two dimensional point."))
```

B.2.3 Creating Instances

For the purpose of creating and initializing an instance CLOS provides a generic function `make-instance`. Normally user programs use own defined constructor to make instance, which encapsulates the call to `make-instance`. By means of `make-instance` CLOS offers the users a great control many aspects of the initialization, for example, supplying a default value for a slot to customizing the initialization by writing a method.

The syntax of `make-instance` is much simpler than that of `defclass`:

`make-instance` *class* `&rest` *initargs*

The first argument is the class from which an instance shall be created. This argument can be either the name of the class specified by a symbol or the class itself. In this way `make-instance` can be used more flexibly.

The `&rest` argument consists of initialization arguments, which controls how initialization is done, for example, filling a slot of the instance. Each initarg is a symbol and value pair, normally the symbol is a lisp keyword, and the symbol specifies the name of initarg.

When `make-instance` is called, following steps happens roughly:

1. Combines the initargs supplied to `make-instance` with the default value for any initargs not supplied explicitly, and makes some checking to ensure the initargs are valid. The result is list of initargs, defaulted initarg list.
2. Storage for instance is allocated and instance created with all slots still being unbound and kept not initialized.
3. Calls the generic function `initialize-instance` with the newly created instance and defaulted instance list. The default primary method for `initialize-instance` does initialization by:
   (a) Initializes slots according to the defaulted arguments list.
   (b) Initializes rest slots that have `:initforms` and are still unbound.
4. Returns the initialized instance.

There are two choices for defaulting the initial value of slots: one is using `:initarg` and `:default-initargs`, the other is using `initform`. They have somehow overlapping function, on the other hand they also serve different purposes.

If the intension is allow users to initialize a slot, then it'd better using `:initarg` together with `:default-initargs`. If this is not the intension, `:initform` should be used and no `:initarg` be provided.

If two approaches are used together, it must be noted that the initialization values provided by `:default-initargs` have higher priority than those provided by `:initforms`.

Now going back to our example of `2d-point`. If what we want is just to create an instance of `2d-point` and we do not care the the coordinate which it represents, this is a simplest case[1]:

```
(setq p1 (make-instance '2d-point))
```

Because of the `:initform` of slot `x` and slot `y`, we actually get a `2d-point` at the origin. And we can now read the slot value of `p1` using its readers:

```
(2d-point-x p1)          ; reads slot x, return 0
(2d-point-y p1)          ; reads slot y, return 0
```

And we can also change its slot values, for example move the point to (1, 2), using its writers:

```
(setf (2d-point-x p1) 1) ; changes slot x to 1
(setf (2d-point-y p1) 2) ; changes slot y to 2
```

Supposing later a second point `p2` is wanted, which is one unit larger than point `p1` in the positive direction of x axis. We can create `p2` and at the same time initialize it, as follows:

```
(setq p2 (make-instance '2d-point
                        :x (1+ (2d-point-x p1))
                        :y (2d-point-y p1)))
```

and get a point at (2, 2).

[1] `setq` is quite like assignment operation in other languages.

B.2.4 Inheritance in CLOS

Some aspects of CLOS inheritance have been described at length in the main text. For example, linearization of class precedence graph. Here is just a short summary.

Inheritance makes it a possible way to sharing of characteristics and behavior among set of classes. Slots are one example of inherited characteristics. Behavior is represented by methods, which are also inherited.

It is important that the inheritance works in an orderly and predictable way. When designing a collection of classes, precedence relationship of these classes must be clearly specified. Because if conflicts occur, because of more than one implementation for desired behavior, the dominating implementation with higher precedence wins.

Each class has a class precedence list, which includes the class itself and all its superclasses. The classes in precedence list are ordered from most specific to least specific. When one class in more specific than another in this list, it has precedence over that class.

Appendix C

Object Orientation in Perl

In this appendix we will examine the scripting language Perl's object orientation facilities.

Perl is not born with the object-oriented programming paradigm in mind, but during its evolution people find it is beneficial to have some facilities to ease some practical programming experience. So the object-oriented programming facilities are amended afterwards, and they show some peculiarities which might not that intuitive in the eyes of pure object-oriented programmers.

We first take a overlook at some basic concepts of object-oriented programming languages, i.e., what are classes, objects and methods in Perl. In some short words:

- A *class* in Perl is simply a package that happens to provide methods to deal with objects.
- A *object* is simply a referenced thingy that happens to know which class it belongs to.
- A *method* is simply a Perl subroutine that expects an object reference (or a package name, for class methods) as its first argument.

We will cover these points more detailed in the following sections.

C.1 Some Background Information

Because of the heritages in the Perl's object-oriented concepts, we'd better look some background information at first.

C.1.1 Packages

In Perl, packages provide the fundamental building block upon which the higher level concepts are constructed, for example, Perl modules and classes. Basically, a package is a namespace, which is a mechanism to protect different sections of code from inadvertently clobber each other's variables.

Packages are independent of files that physically contain them. Several packages can exist in a single file, or a single package can spans several files. But the usual size of package is one file. There is even some special help for the case of one package in one file, as long as the name of file is same as the package it contains and has an file extension of ".pm", which is short for "perl module". The module is the unit of reusability in Perl and can be *used* with the built-in `use` command. Classes are another concept built on the concept of package. The typical class is implemented with a module.

Apart from that there are some magical variables, all other variables belong to its package. The code is always compiled in the current package. The initial current package is package `main`, but at any time the current package can be changed to another one using the `package` declaration. The current package determines which symbol table is used for the name lookups (for names that are not package qualified). Names in other packages can be directly referred by qualifying (prefixing) the name the with the package name and double colon, which is quite similar with that of C++.

The scope of `package` declaration is from the declaration itself through the end of innermost enclosing block, or until another `package` declaration at the same level. All subsequent names (except those package qualified names or lexical local names declared with `my`) will be placed in the symbol table of this package. Typically a `package` declaration is placed at the beginning of file as the first declaration. But `package` declarations can be placed anywhere, even at end of block, in which case it would have no effect at all. The `package` declaration influences merely which symbol table is chosen during the compilation or evaluation.

Packages can be nested inside other packages, for example, a inner package `INNER` within a another package `OUTER`, such as `$OUTER::INNER::variable`. All undeclared symbols must be either belonging to the current package, or fully qualified from the outer package name down. For example, there is no way to find a `$INNER::variable` in package `OUTER` that refers to `$OUTER::INNER::variable`. It would treat package `INNER` as a totally different global package, not that nested package `OUTER::INNER` in package `OUTER`.

Package initializers and finalizers

Package can have two special subroutine definitions that function as package initializers and finalizers: `BEGIN` and `END`, which have their root to the language AWK.

A `BEGIN` subroutine is executed as soon as possible, so long it is completely defined, even before the rest of the containing file is parsed. There can be more than one `BEGIN` subroutines within one file, and they will be executed in the order of definition. Because a `BEGIN` block executes so early, that it can fetch the definitions of subroutines and such from somewhere else for the rest of code in the file even during the compilation. This is important because this can change how the rest of the file will be parsed.

An `END` subroutine, on the contrary, is executed as late as possible, that is when the interpreter is being exited, even if it is exiting because of the `die` function or internally generated exception. Like `BEGIN` subroutines, there can also be more than one `END` subroutines in one file, and they will be executed in reverse order of definition. As the result, related `BEGIN` and `END` subroutines can be nested as expected, if they are paired up rightly.

Autoloading

Normally an undefined subroutine can not be called, as it is quite obvious. However, if there is a subroutine named `AUTOLOAD` in the undefined subroutine's package, the `AUTOLOAD` subroutine is called with the same arguments as would have been passed to original subroutine, and the fully qualified name of the original called subroutine appears in the package global variable `$AUTOLOAD`, in the same package as the `AUTOLOAD` subroutine.

Mostly `AUTOLOAD` subroutines will be used to pull in a definition for the undefined subroutines in question. But an `AUTOLOAD` subroutines can also just emulate the desired undefined subroutines, and act as a fall-back or defaulting mechanism.

C.1.2 References

Pointers in Perl are called references. References are scalars like numbers and characters. They can be created by several means and used (dereferenced) to get the referent.

Creating references

One way to create references is using the backslash ('\') operator. Such a reference can be created to any named variable or subroutine. It works much like address-of ('&') operator in programming language C. For example,

```
$refvar   = \$ARG;
$refconst = \1.95583;
$refarray = \@ISA;
$refhash  = \%ENV;
$refsub   = \&foo;
```

A reference can also be created to an anonymous array by using square brackets (anonymous array composer), such as:

```
$ref_anonymous_array = [1, [2, 3], 'a'];
```

Here we get a reference to an anonymous array of three elements. The second element is again a reference to another anonymous array of two elements.

Similarly a reference can be created to an anonymous hash by using braces (anonymous hash composer):

```
$ref_anonymous_hash = { 'Jan' => 1, 'Feb' => 2 };
```

The above examples use literal numbers and strings, but variables and expressions would work as well. Except the keys of hashes, anonymous array and anonymous hash composers can be mixed freely to create complicated data structures.

A anonymous subroutine can also be referenced by the anonymous subroutine composer as follows:

```
$ref_anonymous_subroutine
                 = sub { print "Hello, world!" };
```

And as stated previous, an object in Perl is just a referenced thingy that knows its class. If a subroutine is used as a constructor for a certain class of objects, it returns such a reference created by built-in function called `bless`. We will see examples of this kind a little later.

Using references

We have seen several ways of creating references, now we look at how the references are used.

If a reference of correct type is in a scalar variable, it can be used anywhere ordinarily an alphanumeric identifier is used as part of a variable or subroutine name. For example:

```
$hello     = "Hello, world!";
$ref_hello = \$hello;
print $$ref_hello;       # same as print $hello
```

Some more dereference examples:

```
print $$ref_anonymous_array[2];     # 'a'
print $$ref_anonymous_hash{'Feb'};  # 2
&$ref_anonymous_subroutine;         # print "Hello, world!"
```

It's important to note that dereferencing of scalar variable happens before any array or hash lookups. To dereference anything more complicated than a simple scalar variable, other ways must be chosen.

The second way is just like the first, except using a block—a syntactic construct consisting of a sequence of Perl statements delimited by braces—instead of a variable. Anywhere an alphanumeric identifier as part of variable or subroutine can be used, this identifier can be replaced with a block returning a reference of correct type. So previous examples could also be wrote like this:

```
print ${$ref_anonymous_array}[2];     # 'a'
print ${$ref_anonymous_hash}{'Feb'};  # 2
&{$ref_anonymous_subroutine};         # print "Hello, world!"
```

Granted it is no use of blocks here, but block can contain any arbitrary expression. In particular, it can contain subscripted expressions, so dereferencing can be done on the elements of the arrays or hashes. As an example,

```
$x1 = 0;
$x2 = 1;
@xs = (\$x1, \$x2);
print ${$xs[0]}, ${$xs[1]};    # $$xs[0] is not what you want
```

There is still another way of dereferencing the references to arrays or hashes, which involves the uses of `->` operator. This is a form of syntax sugar that makes it easier to get at the array or hash elements, specially when the reference expression is complicated. For example:

```
print $$ref_anonymous_array[0];       # 1
print ${$ref_anonymous_array}[0];     # the same
print $ref_anonymous_array->[0];      # the same

print $$ref_anonymous_hash{'Feb'};    # 2
print ${$ref_anonymous_hash}{'Feb'};  # the same
print $ref_anonymous_hash->{'Feb'};   # the same
```

Now we have talked about packages and references, then the understanding of object orientation in Perl will be quite easy.

C.2 Classes

There is in fact no special syntax in Perl for class definition. A package is used as a class by putting method definitions into the package/class.

Each package can have a special array named `@ISA` that tells Perl where else to look for a method if it can't find one in that package. This is how inheritance is implemented in Perl. Each element of `@ISA` array is just the name of another package that happens to be used as class. The packages are searched depth-first and recursively for missing methods, in the order that packages are mentioned in `@ISA`. Classes accessible through `@ISA` are know as parents or superclasses of the current class, which itself is child or subclass.

Perl implementation caches internally the methods that are found in one of its superclasses in the current class for efficiency, so next time it has to find this method, it does not have to look so far. Changing `@ISA` or defining new subroutines invalidates this cache and causes Perl to do lookup again.

If a method is not found by this process but an `AUTOLOAD` subroutine is found, that subroutine is called on behalf of the missing method, with that package's `$AUTOLOAD` variable set to the fully qualified method name.

But if neither a method nor an `AUTOLOAD` subroutine is found in `@ISA`, the last try is made for the method or an `AUTOLOAD` subroutine in the special predefined class called `UNIVERSAL`. `UNIVERSAL` can be thought as a superclass of all other classes which inherit implicitly from it.

After all these efforts if still no suitable subroutine is found, Perl can do nothing but complains by raising an exception.

Inheritance in Perl is only automatically done for methods. Data inheritance is left up to the class itself, which is mostly implemented using an anonymous hash.

C.3 Objects

Perl does not provide a special syntax for constructors. A constructor is simply a subroutine that returns a reference to a thingy that has been blessed into a class. The constructor's name is not significant, conventionally it is called `new`. The constructor does this work using the built-in `bless` function, which marks a reference to a thingy as belonging to a particular class.

The built-in `bless` takes either one or two arguments: the first argument is a regular reference and the second argument if supplied is the package that will own the referenced thingy. If no second argument presents, the current package is used. Here is an example for constructor:

```
package Example;
sub new { return bless {}; }
```

The {} composes a referenced to an anonymous hash. The `bless` function takes that hash reference and makes the referenced anonymous hash a member of the class `Example`, and return the reference.

Once a constructor is defined for a class, it can be used to create object instances of that class. For the above example:

```
$example = Example->new();
```

We will discuss the method invocation in more detail below.

If the constructor is intended to be inherited by subclasses, the two-argument form of `bless` must be used. That is because the methods in Perl are executed in the context of the original class where the methods were defined, and not in the context of the subclass where the methods are called. Taking an example, suppose we have a class `Example2` that is a child of `Example`. In `Example2` the constructor is inherited from `Example`. When an instance of class `Example2` is created by calling `new` as follows:

```
$not_expected = Example2->new();
```

the returned reference in `$not_expected` is actually bless into class `Example` not `Example2`. The only way for that constructor to build an object of proper class is by using the two-argument form of `bless`, which gets the proper class from the argument supplied to constructor, here `new`.

```
package Example;
sub new {
    my $class = shift;        # first argument is class
    return bless {}, $class;
}
```

Then the call of `Example2->new()` will do the right thing. As we will see a little later, `Example2->new()` is equivalent to the call of `new Example2`, then it is clear the constructor `new` returns `bless {}, Example2` as intended.

A final clarification: It is referenced thingy is blessed, not the reference itself. Once blessed, the referenced thingy knows which package they belongs to, the reference to it does not. The `bless` operator merely uses the reference in order to find the referenced thingy.

C.4 Methods

Methods in Perl are nothing but the ordinary subroutines. There is no special syntax for method definition, thought there are some sugar for method invocation. A method just expects its first argument to be the object or class name it is being invoked on.

C.4.1 Class Methods and Instance Methods

If a method expects its first argument to be a class (package) name, it is called a class method. Such methods provide functionality for the class as a whole, not for the individual object instance of the class. Constructors are typically implemented as class methods.

If the first argument expected by the method is an instance, such methods are called instance methods. Conventionally this first argument is shifted into a private variable called `$self` or `$this`.

But actually there is no real difference in definition of method between class method and instance method, and this classification is just the two possible choices of method writer. User could actually have the same method work for both purposes, so called dual-nature method. The method just has to check whether it was passed a reference (instance) or not (class). Suppose we want our previous example class `Example` to own a constructor that can apply to both class itself and instance of same class as follows:

```
$obj1 = Example->new();
$obj2 = obj1->new();
```

We could use built-in `ref` function to find out the type of the first supplied argument. If `ref` returns false, then the first argument is not an object, so it must be a class name.

```
package Example;
sub new {
    my $self = shift;         # first argument is class
    my $type = ref($self) || $self;
    return bless {}, $type;
}
```

C.4.2 Method Invocation

Perl provides two different syntactic forms for invoking class or instance methods. Method calls always receive as their first argument the appropriate class name or object reference upon which they are invoked.

The first syntax is called indirect object form. Suppose we have a class `List` that contains a method `append` that concatenates all of its arguments together and appends them to the `List` instance upon which `list_append` is invoked. And assuming some instances of class `List` are `$list`, and `$list1`, `$list2`, etc. We can get a concatenated list by:

```
list_append list list1, list2;
```

The second syntax uses the `->` notation, and is sometimes called object-oriented form, which is quite similar like that in C++. Our above example would be written as:

```
list->append(list1, list2);
```

Which form is chosen is just the users' preference. There are times when one syntax is more readable, and times when the other is more clear.

There may be occasions when a method of a specific class is to be called. In that case, methods can also be called as the ordinary subroutines, like:

```
List::append(list, list1, list2);
```

However, this does not do any automatic inheritance.

When in a class method there are needs for referencing superclass, the pseudo class `SUPER` can be used. Smalltalk programmers will not find it unfamiliar. Using `SUPER` pseudo class the lookup for methods in superclass can be done without having to name them explicitly.

Bibliography

[1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems, Inc. and Stanford University, 1995.

[2] Pascal André and Jean-Claude Royer. Optimizing method search with lookup caches and incremental coloring. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 110–126, October 1992. Published as ACM SIGPLAN Notices, 27(10), 1992.

[3] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 69–82, October 1996. Published as ACM SIGPLAN Notices, 31(10), 1996.

[4] Andrew Black and Jens Palsberg. Foundations of object-oriented languages. Workshop report, ESPRIT and NSF, March 1994. Published as ACM SIGPLAN Notices, 29(3):3–11, 1994.

[5] G. Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag, 1994.

[6] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification: X3J13 document 88-002R. *ACM SIGPLAN Notices*, 23(9), September 1988.

[7] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system. In *Common Lisp the Language* [41], chapter 28. This chapter presents the bulk of the first two chapters of the Common Lisp Object System specification; it is substantially identical to those two specification chapters as previously published elsewhere [6].

[8] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 303–311, Ottawa, Canada, October 1990. Published as ACM SIGPLAN Notices, 25(10), 1990.

[9] Howard I. Cannon. Flavors: A non-hierarchical approach to object-oriented programming. Technical report, Symbolics, Inc., 1982.

[10] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, number 173 in Lecture Notes in Computer Science, pages 51–67. Springer-Verlag, 1984.

[11] Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 312–321, Ottawa, Canada, October 1990. Published as ACM SIGPLAN Notices, 25(10), 1990.

[12] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Journal of Lisp and Symbolic Computation*, 4(3):21–36, June 1991.

[13] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 433–443, New Orleans, October 1989. Published as ACM SIGPLAN Notices, 24(10), 1989.

[14] Brad Cox. *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley, 1987.

[15] O-J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 common base language. Technical report, Norwegian Computing Center, Oslo, May 1968.

[16] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 211–214, October 1986. Published as ACM SIGPLAN Notices, 21(10), 1986.

[17] Karel Driesen. Selector table indexing & sparse arrays. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 259–270, Washington, DC, October 1993. Published as ACM SIGPLAN Notices, 28(10), 1993.

[18] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 16–24, October 1992. Published as ACM SIGPLAN Notices, 27(10), 1992.

[19] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 164–175, October 1994. Published as ACM SIGPLAN Notices, 29(10), 1994.

[20] Grep Ewing. Class inheritance: The mechanism and its uses. Bachelor's thesis, Department of Software Development, Monash University, October 1994.

[21] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, 1994.

[22] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag. Published as Lecture Notes in Computer Science 512.

[23] Scott E. Hudson. *CUP User's Manual*. Graphics Visualization and Usability Center, Georgia Institute of Technology, 1999. A LALR parser generator for Java.

[24] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989. Published in association with Symbolics Press.

[25] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1991.

[26] Gregor J. Kiczales and Luis Rodrigues. Efficient method dispatch in pcl. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 99–105, 1990.

[27] Gerwin Klein. *JFlex User's Manual*, 2001. A lexical analyzer generator for Java.

[28] Kai Knubben. Verteilte Implementierung der objektorientierten Programmiersprache SELF. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, December 2000.

[29] Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 93–109, Oslo, Norway, August 1988. Springer-Verlag. Published as Lecture Notes in Computer Science 322.

[30] Donald E. Knuth. *The TeXbook*. Addison-Wesley, 1986.

[31] Leslie Lamport. *LaTeX—A Document Preparation System—User's Guide and Reference Manual*. Addison-Wesley, second edition, 1994.

[32] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 397–406, New Orleans, October 1989. Published as ACM SIGPLAN Notices, 24(10), 1989.

[33] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming In The Beta Programming Language*. Addison-Wesley, June 1993.

[34] Merriam-Webster, Inc., Springfield, Massachusetts. *Merriam-Webster's Collegiate Dictionary*, tenth edition, 1999.

[35] Bertrand Meyer, editor. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[36] James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag, 1999.

[37] Kent M. Pitman and Kathy Chapman, editors. *draft proposed American National Standard for Information Systems—Programming Language—Common Lisp*. X3J13 Document 94-101, 15.17 edition, May 1994. No difference concerning the contents to the final ANSI standard X3.226-1994.

[38] Markku Sakkinen. Disciplined inheritance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 39–56. Cambridge University Press Cambridge, 1989.

[39] Andrew Shalit. *Dylan Reference Manual*. Apple Computer, Inc., October 1995.

[40] Randell B. Smith and David Ungar. Programming as an experience: The inspiration for self. In Noble et al. [36], pages 77–108. Originally in the Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 303–330, Åarhus, Denmark, August 1995. Springer-Verlag. Published as Lecture Notes in Computer Science 952.

[41] Guy L. Steele. *Common Lisp the Language*. Digital Press, 225 Wildwood St., Woburn, MA 01801, second edition, 1990.

[42] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, 6(4):40–62, 1986.

[43] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[44] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[45] Antero Taivalsaari. Classes vs. prototypes: Some philosophical and historical observations. In Noble et al. [36], chapter 1, pages 3–16. An earlier version of this chapter appeared in the Journal of Object-Oriented Programming, 10(7):44–50, November/December 1997.

[46] John Viega, Paul Reynolds, Bill Tutt, and Reimer Behrends. Multiple inheritance in class based languages, 1998. Adapted from a research paper resulting from the Jamie (a Java preprocessor) project.

[47] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what Like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 55–77, Oslo, Norway, August 1988. Springer-Verlag. Published as Lecture Notes in Computer Science 276.