Simulation of Object-Oriented Concepts and Mechanisms in Scheme

Kurt Nørmark

Institute of Electronic Systems
Aalborg University
Fredrik Bajers Vej 7
DK-9220 Aalborg
DENMARK

Email: normark@iesd.auc.dk

Phone: +45 98 138788

Contents

List of Figures

Chapter 1

Introduction

This report is about simulation of object-oriented concepts and mechanisms in the programming language Scheme.[1] The report addresses the subject from both a technical and a pedagogical point of view. It is assumed that the reader is familiar with Scheme as well as with the fundamentals of object-oriented programming.

As observed in [2], Scheme is nearly an object-oriented programming language. Procedures in Scheme have the power to represent objects (with time-dependent state) together with a protocol of operations, which may be used to manipulate the object.

Although much research has been, and still is, centered around object-oriented concepts and mechanisms, there are many issues that are not fully explored. The simulation approach described in this report represents one particularly easy way to explore new mechanisms. Besides that, I just find it fascinating to play around with object-oriented concepts and mechanisms in an "elastic" language like Scheme. Scheme is, in my opinion, an excellent "object-oriented playground".

In the rest of this introductory chapter I will first more carefully discuss strengths and weaknesses of the programming language Scheme, from the perspective of simulation of object-oriented concepts and mechanism. Next in section 1.2, I will compare the "simulation" strategy with the more well-known "compilation" and "interpretation" strategies. Section 1.3 contains a brief description of related Scheme-oriented work, and section 1.4 contains an outline and a summary of the rest of the report.

[1]All the Scheme programs in this report are intended to conform with the Scheme Report [9]. Concretely, the practical programming has been done in MacScheme (version 2.0).

1.1 Scheme

Scheme [9, 1] is a dialect of the programming language Lisp. As for this paper, the most important differences between Scheme and traditional Lisp languages are:

1. *Static binding of free names in procedures.*
   Free names in a Scheme procedure are bound in the context of the procedure definition. More traditional Lisp dialects bind the names in the context of the procedure call (dynamic binding).

2. *First class procedures.*
   Being "first class" means that a procedure can be stored and retrieved from data structures, passed as a parameter, and returned as the result from another procedure.

3. *Uniform evaluation of all positions in a procedure call form.*
   If `(p a b c)` is a procedure call in a traditional Lisp dialect, `p` is supposed to be a symbol, and the procedure property of the symbols must be a procedure object. In Scheme, the `p` position does not need to be a symbol. The only important thing in Scheme is that the `p` position is an expression the value of which is a procedure object. Variables in Scheme do not have a separate procedure value besides the "normal" value. All positions in a (non-special) Scheme form are evaluated in the same way.

4. *Program is not data.*
   In a traditional Lisp dialect, the list representation of a piece of program can be manipulated (examined, aggregated, and taken appart) via the primitives of the languages. Furthermore, the main interpreter primitive `eval` is considered as part of the language. In Scheme a piece of program cannot easily be accessed as a list structure, and `eval` is not part of the language.

Points number one and two make it possible to model objects using only procedures. To make this more concrete for readers who have not already gained this insight, look at the following simple example:

```
(define (a)
  (let ((a-var nil))

    (define (b x)
      (set! a-var x))

    b)
```

`A` is a procedure, which has a local variable `a-var`. Local to a there is also a procedure `b`. Due to point number one, the free variable `a-var` in `b` is bound to `a-var` in `a`. Due to point number two, `b` can be returned from `a` because procedures are first class objects. If `c` is defined in the following way

```
(define c (a))
```

`c` refers to a procedure object, which has access to the local variable `a-var` in `a`. Consequently the local state of `a`, represented by `a-var` in the example, cannot be deallocated upon return from `a`. `C` can be thought of as a representation of an object with state hold by the variable `a-var`.

I will elaborate a little bit more on the example, in order to illustrate point number three from above. It is possible to call the local procedure `b` in an `a-object` in the following way

```
((a) (+ 2 3))
```

There are two positions in this form. Both positions are evaluated using exactly the same rules. This is due to point number three from above. The first position is supposed to return a procedure object, which is applied on the result of the other position (the number 5).

In chapter 2 I will describe more carefully how these principles can be used to simulate classes, instances, and message passing between the instances.

1.2 Simulation

This paper is concerned with simulation of language concepts and mechanisms. I find it interesting to compare *simulation* with the more traditional language implementation techniques, *compilation* and *interpretation*. As a matter of terminology, I talk about a *source language* and a *implementation language* (their roles will be described in each of the cases below). I look at the qualities of the three language implementation techniques in the following way:

1. *Compilation.*
   A program in a source language is transformed to an equivalent program in the implementation language, which in this case also is known as the target language. It is possible to execute the target language program via interpretation. When making the transformation tool one usually goes for efficiency

of the produced target language program. Furthermore, one usually assumes that the source language is stable over a considerable time, mainly because it is a relatively complex affair to modify the transformation tool.

2. *Interpretation.*
   On a case by case basis, but following some fixed patterns, the source language constructs of a program are simulated in an implementation language. Flexibility during the program development process rather than efficiency of the end result is emphasized following this scheme. Compared with the compilation approach, it is usually easier to modify the source language via changes in the interpreter. However, the source language is still supposed to be relatively stable.

3. *Simulation.*
   On a case by case basis, constructs in the source language are expressed directly by equivalent constructs in the implementation language. This provides for an extreme degree of flexibility, but it also tends to require that the programmer must deal with many details in a disciplined manner. The identity of the source language may be weak; in return the source language is allowed to be non-stable and fluctuating.

Implementation of a language—or some aspects of a language—via simulation is mainly useful in experimental situations. Using simulation, one is not limited within the concepts and mechanisms of a single and already frozen language. Rather it is possible to gain experience by selecting concepts and mechanism from a "spectrum" of variations, which the implementation language makes attractive.

The simulation approach is also interesting when one wants to explore "the inherent power" of an implementation language. In this context, "power" means the ability to express foreign concepts or mechanisms in simple and elegant ways using existing means of the implementation language.

As mentioned above, the identity of the source language tends to be weak when using the simulation approach. It is the disciplined application of certain "patterns" that makes it possible at all to talk about a source language. If the identity of the source language patterns becomes too weak, one may decide to amplify the identity via the use of syntactic abstractions over the patterns of the simulation. In the Lisp world, such abstractions can be defined via macros. In this report I do not use syntactic abstraction beyond procedure definitions.

1.3 Related Work

The paper *Object-Oriented Programming in Scheme* by Adams and Rees [2] is probably the most central paper about simulation of object-oriented mechanisms in Scheme. Compared with the present report, the Adams-Rees paper contains little or nothing about multiple inheritance, method combination, and metaclasses. The paper is addressed towards the very skilled Scheme programmer.

SCOOPS is an object-oriented programming system implemented on top of Scheme. Among the most interesting features, SCOOPS supports multiple superclasses and active values. SCOOPS extends Scheme with a couple of new special forms for the definition of classes and methods. SCOOPS was originally developed by Texas Instruments (and it is delivered as part of MacScheme).

There also exist several Scheme-like programming languages with object-oriented mechanisms. T [10, 11] and Oaklisp [8, 7] are among the most important of these languages.

1.4 Outline of this Report

In chapter 2 it is summarized how to define classes and instances of classes. It is shown how methods in classes can be activated via message passing. More interesting for all but novices in the field, it is also worked out how to use generic procedure calls instead of message passing.

Chapter 3 introduces hierarchies of classes and inheritance of methods. Two particular state variables are introduced, namely `self` and `super`. It is discussed how `self` can be defined in case we want to simulate virtual-like procedures (in the Simula sense). Two different representations of objects are discussed in this chapter. The most elaborate of these is called the object precedence list representation, and it is really introduced as a preparation for the chapter on multiple inheritance.

Multiple inheritance is the theme of chapter 4. Two different approaches to the handling of multiple inheritance are discussed. The first of these can be characterized as a natural "first try". The other one deals with how to avoid multiple instantiations of parts of objects. In addition I discuss in this chapter how a method combination facility can be simulated. The chapter is concluded with a section on a simple method caching technique, which speeds up the method lookup process.

In chapter 5 it is discussed how classes themselves can be treated as objects. The underlying classes of class-objects are traditionally called metaclasses, and the objects which represent the classes are called meta-objects. I describe an exercise in the definition of the most general classes in a class hierarchy and a metaclass hierarchy. The purpose of this exercise is to demonstrate how experience in this enterprise can be collected via simulations in Scheme. As an important part of the exercise I focus on how object instantiation can be arranged via message passing to the meta-objects.

Besides a bibliography, there is an program index at the end of the report.

Chapter 2

Classes, Instances, and Message Passing

In this chapter I will first summarize how classes and instances of classes can be simulated in Scheme. The techniques that I describe are well-known from the literature [1, 2], but I chose to introduce the basic simulation techniques quite carefully because they make up the entire basis for the rest of the report.

Following the introduction of classes and instances of classes, I describe how to simulate message passing. First I introduce the quite well-known `send` primitive. Finally, I show how to arrange for what is known as a generic procedure interface to the objects.

2.1 Classes and Instances

A class can be understood as a template, from which it is possible to create objects, which are instances of the class. The following pattern outlines how a class can be simulated via a procedure definition in Scheme.

```
(define (class-name)
 (let ((instance-var init-value)
       ...)

   (define (method parameter-list)
     method-body)
   ...

   (define (self message)
     (cond ((eqv? message selector) method)
           ...

           (else (error "Undefined message" message))))

   self))
```

Here and in the following I use an ellipsis ... to indicate that the construct in front of the dots can occur an arbitrary number of times (including zero times). Elements emphasized using the italic font are considered as variables in the presented context. The elements shown in normal fonting are regarded as constant.

The name of the class is *class-name*. The `let` construct binds a number of instance variables to their initial values. In the scope of the instance variables a number of methods are defined. Each method is a procedure in the implementation language. `Self` implements the so-called method lookup procedure. `Self` takes a message as a parameter, and it represents a table that maps method selectors to the actual methods.[1] A message may be any type of object, for instance symbols, that can be discriminated by `eqv?` on their natural denotation. `Self` is returned from the procedure that simulates the class definition. Without loss of generality I will assume that procedures that simulate classes are parameter less. (Possible parameters play the role of instance variables, and consequently, they can be placed together with the other instance variables in the `let` construct.)

[1]The mapping of selectors to methods represents a level of indirection in the process of locating a method. It makes it also possible to have internal methods, simply by not including the methods in the mapping. Finally, the name of the selector, via which the method is known from the outside, is independent of the name of the method. Actually, the method may very well be a lambda expression placed directly in the conditional expression of `self`. If all methods are to be accessible via selectors identical to the method names, the mapping is a one-to-one mapping, and therefore trivial. In this case it would be much more flexible to be able to calculate the method given its selector. In Scheme, this is not possible. What is missing is the possibility to evaluate a symbol in the environment of `self`. The traditional `eval` primitive of Lisp is not part of Scheme, although most Scheme environments support it anyway. Furthermore, in order to use `eval`, it must be possible to capture the environment of `self`, and to pass this environment to `eval`.

The class can be instantiated by calling the procedure, which represents the class:

```
(define instance (class-name))
```

However, I prefer to embed the instantiation into a primitive, which I call `new-instance`:

```
(define (new-instance class)
  (class))
```

Using this primitive, the instantiation of `class-name` from above can be written in the following way:

```
(define instance (new-instance class-name))
```

Following this definition, `instance` is a reference to a new object of class `class-name`. At the implementation level, `instance` refers to the procedure `self`. `Self` directly holds on to the operations of the class, and indirectly to the local state of the object. One can think of `self` as identifying the object, and as a handle to the object from the "outside world".

Using the class template from above, it is not possible to bypass the method interface in reading or mutating the state of an object. This may be felt as a natural limitation, because it protects the state, as does an abstract datatype. Following the simulation-oriented and experimental perspective of this report it is, however, unfortunate to enforce the limited access to the instance variables. It is, of course, possible to define reader and writer methods that can access and mutate the state, but it is tedious to do so for every instance variable.[2]

2.2 Message Passing

In this section I will discuss the activation of methods via so-called message passing. The possible method selectors of objects of class `class-name` are enumerated in the procedure `self` of the class. If we, for instance, want to activate `my-method` in `instance`, it can be done in the following way.

((instance *message*) actual-parameter ...)

[2] One can attempt to come up with alternative solutions to the "instance variable access problem". The procedure `self` can be extended such that reading and mutation is supported side by side with method lookup. However, this requires an entry in `self` for each such transaction, and it is hardly attractive explicitly to deal with such a wealth of details. If `self` is generated by the programming environment, this solution might become attractive.

provided that *message* is mapped to `my-method` in the procedure `self`.[3] Again, I prefer not to use "the raw procedure call syntax". In order to amplify the message sending metaphor, I prefer the following equivalent form.

```
(send instance message actual-parameter ...)
```

The procedure `send` can be implemented in the following simple way:

```
(define (send object message . args)
  (let ((method (object message)))
    (cond ((procedure? method) (apply method args))
          (else (error "Error in method lookup " method)))))
```

2.3 Procedural Activation of Methods

I will conclude this chapter by discussing an alternative to message passing. Assume that we instead of

```
(send instance message actual-parameter ...)
```

want to activate a method via the following form:

```
(message instance actual-parameter ...)
```

I.e., we want to activate a method by calling a so-called *generic procedure*. Following this approach, we cannot syntactically distinguish an ordinary Lisp procedure call from an activation of a method in a class.

From the position of the message in the generic procedure call it is clear that *message* must designate a procedure object in Scheme. Besides this, a message must be an object that can be compared with a method selector using `eqv?` in the procedure `self`. This is not a problem because it makes sense to compare procedure objects with `eqv?` in Scheme [9].

It is easy to write a lambda expression in a pseudo notation for the kind of messages that we need. Such a lambda expression is shown in figure 2.1. There are several things to notice about this lambda expression:

1. The second actual parameter to `send`, which is the message position, is a reference to the entire lambda expression.
2. The third actual parameter to `send` is a list, which actually must be spliced into the actual parameter list of `send`.

[3]Notice that if the method selectors in the class are symbols, the message should also be a symbol. In other words, *message* will be of the form `'message`.

Figure 2.1: A pseudo lambda expression of a generic procedure.

3. In the same way as we used a symbol per selector in section 2.1, we now need a distinct procedure object for each selector.

We are now ready to express the pseudo formulation of the generic procedure in "real Scheme". Because of point 3, it is convenient to *generate* the selectors, instead of writing the needed lambda expressions repeatedly.

```
(define (make-selector)
  (letrec
    ((selector
       (lambda (instance . parameters)
          (apply send
                 (append
                    (list instance selector)
                    parameters)))))
    selector))
```

The procedure `make-selector` is the desired generator. The purpose of `letrec` is to bind a name to the needed procedure, such that it can be referenced in its own definition. The `apply` form used on `send` makes it possible for parameters to be a parameter list of unknown length.[4]

In order to use the described framework, we need explicitly to generate a selector for each of the entries in `self`. This has to be done external to the class, because the selectors play the dual role of generic procedures. Consequently, both the class definition and the selector generations need to be at the outer level.

The simulation of generic procedures, as described in this paper, is inspired by a similar technique described by by Adams and Rees in [2]. Generic procedures are used uniformly in the Common Lisp Object System (CLOS) [3, 6].

[4] The use of `append` and `list` could have been omitted, if the Scheme dialect in question supports the non-essential variation of `apply`, which takes a procedure and *a number of arguments* as parameters, the last of which is a list. MacScheme does not support this variant of `apply`.

Chapter 3

Class Hierarchies and Single Inheritance

Using the framework from the previous section it is possible to model classes, to instantiate classes, and to activate methods in the objects.

In this chapter I will extend the simulation of classes such that the classes can be arranged in hierarchies, and such that inheritance is supported. Only single inheritance is described in this chapter. The single inheritance framework will be generalized to multiple inheritance in chapter 4.

In section 3.1 I describe the basic framework. In section 3.2 I introduce a new representation of objects, which turns out to be particularly useful when we are going to deal with multiple inheritance. Finally, in section 3.3, I change the interpretation of `self`, compared with the interpretation which I first introduce in section 3.1.

3.1 The Introduction of `super`

Classes can be arranged in a tree structure by introducing a designated state variable, which specifies the superclass of the class being defined.

```
(define (class-name)
   (let ((super (new-part super-class))
         (self nil))
     (let ((instance-variable init-value)
           ...)

       (define (method formal-parameter...)
         method-body)
       ...

       (define (dispatch message)
         (cond ((eqv? message selector) method)
               ...
               (else (method-lookup super message))))

       (set! self dispatch))

     self))
```

Compared with the class pattern shown in section 2, the variables `super` and `self` are new. `Super` and `self` correspond roughly to the Smalltalk [5] pseudo variables of the same names. (In section 3.3 I will change the interpretation of `self` such that `self` and `super` play the same role as in Smalltalk.)

Before I explain the remaining new elements of the class pattern from above I will introduce some terminology. If a class C has a superclass S, an object which is an instance of C is said to have an *S object part* (or just an *S part*.) Sometimes I will talk about an S object part as the *super part* of a C object.

Using this terminology, an instance of a class with a super class consists of a sequence of object parts.[1] Figure 3.1 shows an object with two super parts, together with the `super` and `self` variables at the three levels. `Self` refers to the procedure `dispatch`, which corresponds to `self` from chapter 2. `Super` refers to the "higher level part", which is allocated of the primitive `new-part`.

The procedure `new-part` is identical to the procedure `new-instance`, as defined in section 2.1.

```
(define (new-part class)
  (class))
```

Later in the report it turns out to be convenient to be able to distinguish the instantiation of whole classes from the instantiation of parts of classes.

[1] Actually it would be even closer to reality to describe an object as consisting of *nested object parts*.

Figure 3.1: The internal organization of an object with three parts.

The procedure `method-lookup` returns the method with a given selector in an object.

```
(define (method-lookup object selector)
  (cond ((procedure? object) (object selector))
        (else (error "Inappropriate object in method-lookup: " object))))
```

Given that `dispatch` in each class propagates unknow messages to its superclass (if any), `method-lookup` searches from the most specific to the most general class for a method with the given selector.

Let me illustrate in some more details what happens when a class with a super class is instantiated. Externally, an instance of a class is created as explained in section 2. Internally, the instantiation may start a chain of instantiations of parts, which stops when a root of the class hierarchy is reached (see below). As already explained, and as illustrated in figure 3.1, each part of the object contains a reference to the level above in the object-hierarchy.

Similarly, let me explain what happens during message passing when classes are organized in hierarchies. If a message `message` is sent to an object, and if the message "falls through" the `cond`-construct in `dispatch`, the form

```
(method-lookup super message)
```

is executed. `Super` is a reference to the superpart of the object. Eventually, the method whose selector matches the message is located, or the root of the class hierarchy is reached without finding a method with the given selector. In the former case the method is activated with actual parameters supplied in the `send` form. In the latter case, we want to get a error message, which signals that there are no methods that match the message.

To accommodate the behavior just described, the root of the class hierarchy may be defined in the following simple way:[2]

```
(define (object)
  (let ((super ())
        (self nil))

   (define (dispatch message)
       ())

   (set! self dispatch)
   self))
```

This defines a trivial class, in which `super` is bound to the empty list, and with an empty `dispatch` procedure. Given this definition of the root of the hierarchy, the `send` primitive from the previous section can be modified to react properly in the case that a message is unknown to an object:

```
(define (send object message . args)
  (let ((method (method-lookup object message)))
    (cond ((procedure? method) (apply method args))
          ((null? method) (error "Message not understood: " message))
          (else (error "Inappropriate result of method lookup: " method)))))
```

Compared with the version of `send` in section 2.1, this version handles the case where the method lookup does not return a method (signaled with the empty list, which is supposed to be returned from `dispatch` of the root class.) Notice that this version of `send` uses `method-lookup`, which was defined above.

3.2 Object Precedence Lists

As explained above, the description of the method lookup process is distributed across the various dispatching procedures in the classes. The procedure

[2]The definition of `object` may be even simpler if `dispatch` is defined “inline” in the `let` construct at the place of `nil`. However, I believe that it is easiest to understand the classes and class patterns, if a standard “style” is being used.

`method-lookup` is only syntactic sugar for the procedure call, which realizes the method lookup in another object part. If a given procedure `dispatch` cannot find a selector that matches a message, it propagates the lookup to the dispatch procedure of the super part of the object. This is done via a (recursive) call to the procedure `method-lookup`. In this section I will describe an alternative representation of objects that allows this propagation process to be described in a single procedure.

The idea is to represent an object as a list of parts (concretely as a list of `dispatch` procedures) instead of as a single part. Let us assume that c_n is a class, and that the superclass of c_i is c_{i-1}, $i = 1..n$. Given these assumptions, we want to represent an instance of c_n as the list

(c_n-part c_{n-1}-part ... c_0-part)

where c_i-part, at the Scheme level, is the dispatch procedure of a c_i object part. This list will be called the *object precedence list* (or just the *precedence list*) of the object.

Given the new representation of objects, it is necessary to modify both the `dispatch` procedure and the assignment of `self`. Let me show a class template under the assumption that objects are represented as precedence lists of parts.

```
(define (class-name)
   (let ((super (new-part super-class))
         (self ()))
     (let ((instance-variable init-value)
           ...)

       (define (method formal-parameter...)
         method-body)
       ...

       (define (dispatch message)
         (cond ((eqv?  message selector) method)
                    ...
               (else ())))

       (set!  self (class-handle dispatch super)))
     self))
```

As of here, `class-handle` can be defined to be an alias for `cons`.

```
(define class-handle cons)
```

`(Class-handle dispatch super)` returns a list, the head of which is `dispatch`, and with a tail that is the object precedence list of the super part of the object (returned by another incarnation of `class-handle` via the procedure `new-part`.) In section 4.3, `class-handle` will be redefined to handle the changes introduced by having multiple superclasses.

Notice that `dispatch` no longer propagates messages to the super part of the object. If a message isn't mapped to a method in `dispatch`, the dispatching procedure returns the empty list.

The procedure `method-lookup` now has to be redefined. `Method-lookup` is not used directly in the class definition any more, but it is, among other places, used in the `send` primitive.

```
(define (method-lookup object-precedence-list message)
  (let ((result (linear-search
                  object-precedence-list
                  (lambda (object)
                    (method-lookup-single-part
                      object
                      message)))))
    (method-lookup-single-part result message)))

(define (method-lookup-single-part object message)
  (object message))

(define (linear-search list found?)
  (if (found? (car list))
      (car list)
      (linear-search (cdr list) found?)))
```

`Method-lookup` is implemented as a linear search on `object-precedence-list`. The object precedence list is searched for a part, which responds to the given message. The auxiliary procedure `method-lookup-single-part` serves a dual role. It is used rather directly in the predicate passed to the linear search procedure, and it is used on the result of the search to locate the method. (This is possible in Scheme, because of the relatively free interpretation of boolean values).

Both the message sending primitive `send` and the instantiation primitive `new-instance` survive the shift of object representation.

The advantages of the precedence list representation of objects, as introduced in this section, are not great compared with the original representation. If, however, we introduce multiple inheritance it turns out that the precedence list representation is better suited than the more simple representation. I will come back to that in section 4.2. Until then the precedence list representation will not be used.

Figure 3.2: Two different interpretations of `self`.

3.3 Another Interpretation of `self`

In the framework explained in section 3.1, `self` refers to that part of an object, in which it textually is contained. Figure 3.2(a) shows an object with three parts, and it is illustrated to which part of the object `self` refers.

Figure 3.2(b) shows an alternative way of interpreting `self`. Following this approach, `self` always refers to the object part, which corresponds to the most specialized class involved. This is the way `self` is used in Smalltalk-80 [5]. In Simula terms [4], this makes all methods virtual-like. I will now show how the situation in figure 3.2(b) can be obtained.

First, I will assume that each class definition contains an operation `set-self`, and that the message `set-self!` activates that operation.

```
(define (set-self object-part)
    (set! self object-part)
    (send super 'set-self! object-part))
```

`Set-self` assigns a new value to `self`, and it propagates a similar request to its super class. In `object`, which is assumed to be the root of the class hierarchy, no propagation should take place. I.e., `(send super ...)` should not be included in the `set-self` method of the class `object`.

A call to the procedure `virtual-operations` defined as

```
(define (virtual-operations object)
  (send object 'set-self! object))
```

declares that `self` in the object parameter, and in its (direct and indirect) super-parts, follow the new interpretation.

The decision about the role of `self` can be postponed until class instantiation time. If we want the new interpretation, the primitive `new-instance` from section 2.1 should be changed to the following procedure:[3]

```
(define (new-instance class)
 (let ((instance (class)))
   (virtual-operations instance)
   instance))
```

Notice that we don't want to change `new-part` in a similar way, because this would result in a bunch of temporary assignments of `self` to gradually larger and larger parts of the object under construction.

There are other ways to obtain the Smalltalk-like interpretation of `self`. In [2] it is proposed that `self` of the outer part of the object is passed as a parameter to each method. In that way the methods that I call `set-self` are not necessary, but in return each method of every class must take an extra parameter.

As the conclusion of this section, let me illustrate how one can take advantage of the new interpretation of `self`. I want to implement a method `responds-to`, which tells whether a given object responds to a given message. Because every object should have access to this method, it is natural to place it in the root class. The method is here shown in context of the whole `object` class (which is an extension of the trivial `object` class shown in section 3.1.)

[3] It is, of course, also possible to have two variations of `new-instance`; one which results in objects as those in figure 3.2(a), and one which results in objects as those in figure 3.2(b). It is also easy to make the decision a parameter of `new-instance`.

```
(define (object)
  (let ((super ())
        (self nil))

   (define (set-self obj-part)
     (set! self obj-part))

   (define (responds-to message)
     (let ((method (method-lookup self message)))
       (if method #t #f)))

   (define (dispatch message)
     (cond ((eqv? message 'set-self!) set-self)
           ((eqv? message 'responds-to?) responds-to)
           (else ())))

   (set! self dispatch)
   self))
```

The operation `responds-to` relies on the fact that `self` in object refers to the "top part" of the object, of which it is a part. `(Method-lookup self message)` searches through an object from the most specific part towards the more general parts. If an operation is found, which responds to `message`, it is returned from `method-lookup`. If not, the search reaches the `dispatch` procedure in the class `object`, which returns the empty list.

It can be concluded that method lookup, as known from, e.g., Smalltalk, is easy to simulate in our framework via a change of the interpretation of `self`.

Chapter 4

Multiple Inheritance

The class hierarchies that I have described in the previous chapter have all been strictly tree-structured. I.e., for every class there exists at most one superclass, and every class inherits properties from at most one direct superclass. In this chapter I will describe how to generalize the framework in such a way that multiple inheritance is accounted for.

First I will describe a simple simulation approach to multiple inheritance. Based on an analysis of the simple approach, I proceed in section 4.2 and 4.3 with a technique, which is based on the object precedence list representation, as introduced in section 3.2. The chapter is concluded with a description of how to simulate so-called method combination.

4.1 A Simple Approach

The basic idea behind my simulation of multiple inheritance is to let the variable `super` refer to a list of "super object parts" instead of only one part, as in the single inheritance case.

```
(define (class-name)
  (let ((super (new-part-list super-class-name ...))
        (self nil))
    (let ((instance-variable init-value)
          ...)

      (define (method formal-parameter ...)
        method-body)
      ...

      (define (dispatch message)
        (cond ((eqv? message selector) method)
              ...
              (else (method-lookup super message))))

      (set! self dispatch))

   self))
```

The only difference between this template and the class template shown in chapter 3 is that `new-part` has been replaced with `new-part-list`, which takes a variable number of super classes as parameter. `New-part-list` is a mapping of `new-part` on the list of super classes.

```
(define (new-part-list . super-class-list)
  (mapcar new-part super-class-list))
```

Furthermore, it is necessary to change the procedure `method-lookup` such that it can handle the case where a method is searched for in a list of classes:

```
(define (method-lookup objects message)
  (cond ((procedure? objects) (objects message))
        ((null? objects) ())
        ((pair? objects)
           (let ((leftmost-method ((car objects) message)))
             (if leftmost-method
                 leftmost-method
                 (method-lookup (cdr objects) message))))
        (else (error "Inappropriate objects in method-lookup: " objects))))
```

This version of `method-lookup` realizes a left-to-right, depth-first search in the super object parts. Notice that the new version of `method-lookup` is an extension of `method-lookup` from section 3.1.

Figure 4.1: The sharing of parts in a “diamond” of four classes.

If `self` is interpreted along the lines described in section 3.3, the methods called `set-self` must be sure to propagate the `set-self!` message to each of the super parts of the actual object part.

There are two problems with this simple approach to the handling of classes with multiple superclasses:

1. *Multiple instantiation of parts.*
   If the situation is as in figure 4.1(a), i.e., if two super classes `b` and `c` of `d` are joined together in a common superclass `a`, then the `a`-part will be allocated twice, as shown in figure 4.1(b). In most situations we want to change the situation in figure 4.1(b) to that of figure 4.1(c), where the `a`-part is shared between the `b`-part and the `c`-part.

2. *Redundant searching during method lookup.*
   A search procedure like the one programmed above will make repeated method lookup in the same object parts, because a given object part can be reached from different sub-parts. This is clearly not elegant, an it is a waste of time.

In the following section I will describe how the first problem can be solved. It will be assumed that objects are represented as precedence lists of object parts, as described in section 3.2. The precedence list representation of objects does not directly solve problem number one, but the needed mechanism is closely akin to the precedence list mechanism.

In section 4.3 I will do the actual construction of the precedence list representation of objects in the multiple-inheritance case. This contribution solves problem number two from above. In section 4.4 a CLOS-like method combination technique is elaborated. And finally in section 4.5 I discuss how to make the method lookup more efficient. This is especially relevant when method combination is in use.

4.2 Shared Object Parts

In order to avoid multiple instantiations of an object part, it is necessary to test for every instantiation of an object part, whether the part already has been instantiated. This section shows how this problem is dealt with.

Let me as the starting point introduce the template of a class with multiple super classes, which uses the precedence list representation of objects.

```
(define (class-name parts)
  (let ((super (super-class-list parts super-class-name ...  ))
        (self ()))
    (let ((instance-variable init-value)
          ...)

    (define (method formal-parameter...)
      method-body)
    ...

    (define (dispatch message)
      (cond ((eqv?  message class-id) class-name)
            ((eqv?  message super-classes) super)
            ((eqv?  message selector) method)
            ...
            (else ())))

   (set!  self (class-handle dispatch super))
   self)))
```

Overall, a class describes how to instantiate the super parts of the class. This is programmed in the procedure `super-class-list`, which returns a *list* of object parts, where each part is represented as an object precedence list (see 3.2 and 4.3).

If no sharing of parts is involved, `super-class-list` "asks the super class to instantiate itself" in the normal way (by calling the Scheme procedure underlying the class). If, however, `super-class-list` finds out that a superclass already has been instantiated once, the object precedence list representation is reconstructed from the existing part and, in turn, its super parts.

In more details, there are a number of new elements in the class template compared with that from section 4.1:

1. The `parts` parameter.
   The procedure which simulates a class takes a parameter called `parts`. During the instantiation of the class, this parameter is supposed to contain a list of already instantiated parts.

2. `Super` is bound to the result of a call to `super-class-list`.
   The procedure `super-class-list` replaces the procedure `new-part-list` in the superclass definition clause. `Super-class-list` instantiates the super parts of the object, and it ensures that no part of the object is instantiated more than once.

3. An object can return its class.
   Given the selector `class-id`[1], `dispatch` responds with the lambda expression of the procedure, which implements the class. In other words, it is possible for an object `obj` to return the class of which it is an instance. It can be done via the following procedure call: `(method-lookup obj class-id)`, which at the end of this section will be abstracted to `(class-of-object obj)`.

4. An object can return its super objects.
   Similarly, an object can return the list of super objects, which are referred to by super. This can be done with `(method-lookup object super-classes)`, or with the procedure `supers-of-object`, which will be defined later in this section.

5. Simplified response on non-matching message in `dispatch`.
   In the else clause of `dispatch` there is no method lookup to the super class. Rather, `dispatch` returns the empty list if the message "falls through" the dispatch procedure. This is because objects are represented as precedence lists of object parts (see section 3.2.)

6. Change of the assignment of `self`.
   `Self` is assigned to the result of `(class-handle dispatch super)` at the end of the procedure. `Class-handle` returns an object precedence list when the class is instantiated. The description of `class-handle` is postponed to section 4.3.

Because of the `parts` parameter, it is necessary to introduce a slightly changed version of the class instantiation primitive, `new-instance`.

```
(define (new-instance class)
  (let ((instance (class ())))
    (virtual-operations instance)
    instance))
```

In the rest of this section I will describe how the procedure `super-class-list`, together with the procedures on which it depends, can be implemented. Recall that the purpose of `super-class-list` is to instantiate the super parts of the actual object part, and to avoid multiple instantiations of the same object part.

The procedure `super-class-list` is just syntactic sugar for `super-class-list-1`, which is more convenient to work with because it has a fixed number of parameters.

[1]For improved readability, `class-id` is a variable, which is bound to a "gensymed symbol". The same is true for the variable `super-classes`.

```
(define (super-class-list existing-parts . class-list)
  (super-class-list-1 existing-parts class-list))

(define (super-class-list-1 existing-parts class-list)
  (if (null? class-list)
      ()
      (let ((parts (instantiate-super-class
                     existing-parts
                     (car class-list))))
        (cons parts
              (super-class-list-1
               (append parts existing-parts)
               (cdr class-list))))))
```

`Super-class-list-1` basically maps the procedure `instantiate-super-class` on each class in `class-list`. During this mapping, it keeps track of already instantiated parts in its first parameter.

Before it is explained how `instantiate-super-class` works, let us look at a simple example of how the class instantiation proceeds. The example is based on the class hierarchy shown in figure 4.1(a). The call `(new-instance d)` first instantiates the b branch of the class hierarchy. When instantiating the c branch, `existing-parts` refers to a list of the b-part and the a-part, which are the already existing parts of the object. During the instantiation of the c branch it can be discovered that the a-part shouldn't be instantiated again.

`Instantiate-super-class` is defined in the following way:

```
(define (instantiate-super-class existing-parts class)
  (let ((prev-i (previous-instantiation class existing-parts)))
    (if prev-i
        (class-handle prev-i
                      (supers-of-object prev-i))
        (class existing-parts))))
```

It uses the procedure `previous-instantiation` to test whether there exists an instantiation of `class` in `existing-parts`. If no such instantiation exists, `class` is instantiated and returned. The result of this instantiation is an object precedence list (see section 4.3). If `class` already has been instantiated, a *reconstruction* of the existing object part is returned. At the Scheme level, `prev-i` becomes bound to the `dispatch` procedure (`self`) of the already existing instance. In order to reconstruct the object precedence list representation of the object, I also need information about the super objects of the object. The list of super parts (represented as object precedence lists) is returned by the procedure `supers-of-object`.

```
(define (supers-of-object object)
  (method-lookup-single-object object super-classes))

(define (method-lookup-single-object object message)
  (object message))
```

Based on "self and super information", `class-handle` can reconstruct the precedence list of the already existing object part.

In the procedure `previous-instantiation`, it is necessary to be able to tell whether a given object is an instance of a given class.

```
(define (is-class-of? class object)
  (eq? class (class-of-object object)))

(define (class-of-object object)
  (method-lookup-single-object object class-id))
```

`Is-class-of?` uses a function `class-of-object`, which for a given object returns its class—a lambda expression at the implementation language level. In Scheme, it is possible to compare two procedures for equality using `eqv?`.

Previous-instantiation can now test if a class already is instantiated in the following way:

```
(define (previous-instantiation class existing-parts)
  (let ((res (memb is-class-of?
                   class
                   existing-parts)))
    (if res
        (car res)
        #f)))
```

If, according to `existing-parts`, `class` already has been instantiated it returns the already existing part (which also serves as *true*.) Else it returns *false*.

`Memb` is similar to the Scheme primitive `member`. `Memb` tests whether an element is a member of a list using an explicitly passed comparison procedure.

```
(define (memb comparison obj plist)
  (cond ((null? plist) #f)
        ((comparison obj (car plist)) plist)
        (else (memb comparison obj (cdr plist)))))
```

Thus, `(memb is-class-of?  c object-parts)` tests whether an object in the list `object-parts` is an instance of the class `c`.

Figure 4.2: A sample class hierarchy with multiple inheritance.

4.3 The Object Precedence List Representation

In section 3.2 I introduced the precedence list representation of objects in the single-inheritance case. In this section I will discuss a similar representation when using multiple inheritance in the class hierarchy. The precedence list representation is inspired by the class precedence lists from the Common Lisp Object System (CLOS) [3].

A *class precedence list* of a class C is a total ordering of C and all its (direct and indirect) superclasses. In this ordering of classes, each of the classes occurs exactly once. If C is a class with the direct super classes C_1, C_2, ... C_n (in this order), the following rules define a class precedence list:

1. C precedes C_1, C_2, ... and C_n in the ordering, and
2. C_i precedes C_{i+1}, for i from 1 to $n-1$.

For more details about class precedence lists, see [3].

An object precedence list is similar to a class precedence list. I.e., an *object precedence list* of an instance of a class C is the similar total ordering of the object parts of the instance, where each part occurs only once. The same constraints as described in the two points above should also hold for the object parts.

Let me illustrate class precedence lists and object precedence lists with the example in figure 4.2. For each class in the figure, precedence lists of the components (classes or instances) are shown in boxes. Considering for example the class F, an instance of F consists of the following ordered collection of parts:

F-part B-part C-part A-part.

A is the last one, because it must be preceded by both B and C according to (two applications of) rule number one from above; B precedes C because of rule number two; F is the first one, again because of rule number one. Using a precedence list representation of objects, an instance of the class F is represented as the list

(F-part B-part C-part A-part)

where each part is a reference to the dispatch procedure of the part.

It is worth noticing, that using this representation of objects, the method lookup procedure in the multiple-super-class case is the same as in the single-super-class case. The method lookup procedure which applies has already been shown in section 3.2. Compared with the version of `method-lookup` from section 4.1 (the so-called simple approach), in which the search strategy is encoded into the lookup procedure, the procedure shown in section 3.2 is clearly a simplification. However, there is of course a price for this simplicity. The price is so to speak paid in `class-handle`, which is described next.

The procedure `class-handle` makes and returns a precedence list representation of an object. Recall that `class-handle` is used directly in the class definition template, and it is used in the procedure `instantiate-super-class` to fake the instantiation of an object part from already existing parts.

```
(define (class-handle self super-list)
  (cons self (merge-parts super-list)))
```

This procedure reflects rule number one in the definition of the class precedence list. `Self` is supposed to be a dispatch procedure, and `super-list` is a *list of* object precedence *lists* of the super parts of the object. The procedure must return an object precedence list of the object, to which `self` belongs.

The procedure `merge-parts` reflects rule number two in the definition of the class precedence list. `Merge-parts` is defined through an iterative (tail-recursive) helping procedure called `merge-parts-1`.

```
(define (merge-parts lists)
  (reverse (merge-parts-1 () lists)))

(define (merge-parts-1 result input)
  (if (null? input)
      result
      (let* ((first-list (car input))
```

```
            (first-el (car first-list))
            (rest-lists (cdr input)))
        (merge-parts-1
         (if (multi-member object-eq? first-el rest-lists)
             result
             (cons first-el result))
         (if (cdr first-list)
             (cons (cdr first-list) rest-lists)
             rest-lists)))))

(define (object-eq? x y)
  (eq? (class-of-object x) (class-of-object y)))
```

It is probably easiest to understand these merge procedures through an example of the merge process. Let us assume that we are about to instantiate an H object, relative to figure 4.2. In order to do that the three precedence lists of the super parts of H have to be merged. I.e., `input` to the procedure `merge-parts-1` becomes:

$$((p_E\ p_B\ p_A)\ (p_F\ p_B\ p_C\ p_A)\ (p_G\ p_C\ p_D\ p_A)$$

`Merge-parts-1` first checks if p_E is a member of either of the lists $(p_F\ p_B\ p_C\ p_A)$ or $(p_G\ p_C\ p_D\ p_A)$. It is not, and therefore p_E is included into `result`. Next, it is checked if p_B is part of the two lists $(p_F\ p_B\ p_C\ p_A)$ or $(p_G\ p_C\ p_D\ p_A)$. That is the case, and therefore this instance of p_B is "postponed" until it is met later during the merge process. Continuing this way, the result of merging the three lists becomes $(p_E\ p_F\ p_B\ p_G\ p_C\ p_D\ p_A)$.[2]

The remaining procedures are `multi-member` and, in turn, its helping procedures. `Multi-member` tests whether an element is a member of a list, in a list of lists. The following implementation of `multi-member` just maps the `member`-like procedure, called `memb`, over the list of lists, and `or`-reduces the result to an boolean value:

```
(define (multi-member comparison el list-of-lists)
  (reduce or-proc
          (map (lambda (l)
                 (memb comparison el l))
               list-of-lists)
          #f))
```

[2]For people who are familiar with the details of the definition of class precedence lists in CLOS [3], it may be interesting to compare the "merge technique", as described here, with the topological sorting procedure used in [3]. First it can be noticed that the "merge procedure" does not discover inconsistencies in the partial orderings generated by the class definitions. If, however, there are no inconsistencies, the "merge procedure" seems to give the same class precedence list as the *deterministic* topological sorting procedure.

`Reduce` is well-known, and `or-proc` is `or`, just in procedure form (in Scheme it is not possible to use a special form as a parameter to a higher order procedure). For the sake of completeness, these procedures are shown next:

```
(define (reduce combiner list initial-value)
  (if (null? list)
      initial-value
      (combiner (car list)
                (reduce combiner
                        (cdr list)
                        initial-value))))

(define (or-proc x y)
  (or x y))
```

This concludes the description of how to construct the precedence list representation of objects in the case of multiple super classes.

4.4 Method Combination

Given the similarities between the precedence list framework described above and elements of CLOS, I feel that it is interesting to go one step further in the direction of CLOS, namely to simulate method combination.

It is the purpose of this (and the following) section to show how it is possible to experiment with an "advanced topic". I do not, by any means, intend to deal with all the necessary details of method combination (and an efficient implementation of method combination).

A method in CLOS has associated a *role*. "Ordinary methods" are called *primary methods*. In addition, there are *before methods*, *after methods*, and *around methods*. The methods of a *generic function*, say `m`, contribute to the so-called *effective method* of `m`. Effective methods are activated via the generic function, much along the lines described in section 2.3 of this report.

Considering only before methods, primary primary, and after methods, the following rules represent a simple, but useful method combination strategy.

1. All the before methods of `m` are called in most-specific-first order.

2. The most specific primary `m` method is called. The result of the primary method becomes the result of the effective method.

3. All the `m` after methods are called in least-specific-first order.

This is similar to the so-called standard method combination of CLOS (without around methods). For more details on method combination see [3, 6].

The method definition technique in our simulated object-oriented language must be changed to account for the roles of method. I chose to represent the roles of methods in the `dispatch` procedure of the class. The following is an example of a `dispatch` procedure with a specification of roles.

```
(define (dispatch message)
   (cond ((match? message class-id) class-name)
         ((match? message 'set-self!) set-self)
         ((match? message '(m before)) m)
         ((match? message '(n after)) n)
         (else ())))
```

In the class which contains `dispatch`, the method `m` is designated as a before method, and `n` is designated as an after method. `Set-self` has not been associated a role, and therefore it is considered a primary method. Notice that the introduction of roles is a proper extension of the existing framework, because "role-less" methods are allowed (they are considered as primary methods).

The procedure `match?` has the responsibility to match messages against method selectors, doing the appropriate defaulting. Both messages and selectors may be two element lists with roles.

```
(define (match? message selector)
  (cond ((symbol? message)
         (or (eq? message selector)   ;(1)
             (and (pair? selector) (eq? (cadr selector) 'primary)   ;(4)
                                  (eq? (car selector) message))))
        ((pair? message)
         (or (equal? message selector)  ;(2)
             (and (symbol? selector) (eq? (cadr message) 'primary)  ;(3)
                                    (eq? (car message) selector))))
        (else (error "Malformed message in match?"))))
```

In `match?` there are four cases to consider (marked in the program above):

1. Both `message` and `selector` are symbols, which must be `eq?` to match. The `selector` selects a primary method.

2. Both `message` and `selector` are lists, which must be `equal?` to match.

3. `Message` is a list of the form `(mes primary)`. The selector `mes` matches this message.

4. `Selector` is a list of the form `(sel primary)`. The message `sel` matches this selector.

In Scheme it is not difficult to simulate the aggregation of primary, after, and before methods into an effective method. The following procedure describes the aggregation proposed above.

```
(define (method-combination object message)
  (let ((before-methods (lookup-all-methods object message 'before))
        (primary-methods (lookup-all-methods object message 'primary))
        (after-methods (lookup-all-methods object message 'after)))
    (if (null? primary-methods)
        ()
        (lambda pl
          (for-each
           (lambda (m)
             (apply m pl))
           before-methods)

          (let ((result (apply (car primary-methods) pl)))

            (for-each
             (lambda (m)
               (apply m pl))
             (reverse after-methods))

            result)))))
```

It is easy to make experiments with other combination rules. This can be done entirely local to the procedure `method-combination`.

The procedure `lookup-all-methods` is a "generalization" of the procedure `method-lookup`. `Lookup-all-methods` returns a list of all methods of a particular role.

```
(define (lookup-all-methods object-precedence-list message . role)
  (let ((actual-role (if role (car role) 'primary)))
    (filter
     procedure?
     (map
      (lambda (object)
        (method-lookup-single-object object message actual-role))
      object-precedence-list))))

(define (method-lookup-single-object object message . role)
  (if role
      (object (list message (car role)))
      (object message)))
```

If no `role` is passed as a parameter to `lookup-all-methods`, the role is defaulted to `primary`. `Method-lookup-single-object` is an extension of the similar procedure from section 3.2, which now supports message roles.

Having defined `method-combination`, it may play the exact same role as `method-lookup`. I.e., instead of calling `method-lookup` in `send`, we now call `method-combination`.

```
(define (send object message . args)
  (let ((effective-method (method-combination object message)))
    (cond ((procedure? effective-method) (apply effective-method args))
          ((null? effective-method)
              (error "Message not understood: " message))
          (else (error "Inappropriate result from method lookup: "
                       effective-method)))))
```

4.5 Caching of Effective Methods

The overhead caused by method lookup is present in the single inheritance as well as in the multiple inheritance case. However, the overhead may be outrageous when using method combination, along the lines described in section 4.4. A message sending primitive (a generic function call) causes extensive searching in the class hierarchy for the necessary methods. In this section I will sketch a simple caching technique that can alleviate some of these problems.

Instead of using `method-combination` in, for instance, `send`, the following procedure with an identical parameter profile is used.

```
(define (get-method-combination object message)
  (let ((cached-effective-method (get object message)))
    (if cached-effective-method
        cached-effective-method
        (let ((effective-method (method-combination object message)))
          (put object message effective-method)
          effective-method))))
```

The `get` and `put` primitives access and mutate a two-dimensional table (see, e.g., [1]).

The first time a particular method (identified by `method`) in a particular object is called, the effective method is being cached. Subsequent activations of the same method in the same object do not compute the method combination; rather, the cached value of the effective method is used.

It would have been more natural to cache the effective method on the class rather than on the objects of the class. In that way all instances of the same class could share the cached method. However, two instances of the same class do not share the procedure objects, which simulate the methods. Each instance has its own, local procedure objects. (This is, by the way, what makes it possible to represent an object as a "dispatch procedure"). In that way also the effective method becomes specific to one and only one instance of a class. For more details about this problem, and for elements of a solution, see [2].

Empirical measurements indicate that the caching strategy pays well off with respect to time consumption. In an example where an effective method was formed by two before methods, three after methods, and a primary method, the caching technique was 10 to 20 times faster than without caching.[3]

When a new method that contributes to an effective method is being introduced, it is important to get rid of the cached value. In the framework of this report, it would make sense to clear the cache of all methods of an object, upon redefinition of the class. In turn, this requires that we can get our hand on all instances of a class. This kind of administration may very well turn out to be more complicated than the caching proper.

[3] In the measurement, all methods were empty. In that way only the method lookups affect the timing.

Chapter 5

Metaclasses

For several reasons it is attractive and interesting to represent classes as objects that are on equal footing with the objects that are instantiated from the classes.

1. It is possible to instantiate a class by sending a message to the object, which represents the class.

2. It is natural to represent the characteristics of the class in the state variables (class variables) of the object that represents the class. In that way the characteristics of a class are readily available as data to the surrounding system.

Point number two provides for so-called metaprogramming, which is particularly important when making tools in the programming environment.

The class of an object, which represents a class, is usually called a *metaclass*. In this section I will discuss how to simulate metaclasses and how to simulate classes as objects. I will also demonstrate how to simulate the upper part of the class hierarchy, which ties the whole object-oriented framework together. Finally, I will show how to instantiate classes via message passing to the object, which represents the class.

5.1 The pattern of Metaclasses

In this section I will show and discuss the syntactic pattern, which is used to simulate metaclasses. For simplicity, I assume that only single inheritance is involved. More specifically, this chapter will be based on the framework from section 3.3.

A metaclass can be simulated using the following pattern:[1]

```
(define (metaclass-name)
   (let ((self nil)
         (super (new-part super-metaclass)))

     ;; class variables and methods
     (let ((class-variable init-value)
           ...)

       (define (class-method formal-parameters...)
         body)
       ...

       (define (instance-description)
         (let ((super (new-instance-part super-class))
               (self nil))
           (let ((instance-variable init-value)
                 ...)

             (define (instance-method formal- parameter...)
               body)
             ...

             (define (inner-dispatch m)
               (cond ((eqv?  m selector) method)
                     ...
                     (else (method-lookup super m))))

             (set!  self inner-dispatch))
             self))

       (define (outer-dispatch m)
         (cond ((eqv?  m 'instantiator) instance-description)
               ((eqv?  m selector) method)
               ...
               (else (method-lookup super m))))

       (set!  self outer-dispatch))
     self))
```

[1] I want to acknowledge *Ole Lehrmann Madsen* for giving me the concept of this simulation of metaclasses.

Figure 5.1: The main components of a metaclass pattern.

Although this class pattern is slightly more complicated than the similar patterns from the previous chapters, the overall structure of it is quite simple. Figure 5.1 illustrates the main components of the class template from above.

Instances of metaclasses become "meta-objects", which represent a class. The instances of classes are described by the so-called *instance description*, which is a class-like description at the level of the class methods. A class object must return the instance description when it is passed the special message called `instantiator`. The class variables and the class methods are visible in the instance description. Both the metaclass and the instance description have `super` and `self` variables, the latter of which are bound to `inner-dispatch` and `outer-dispatch` respectively.

Super parts of a metaclass can be instantiated by `new-part`, which is described in section 3.1. Super parts of an "ordinary class", on the other hand, must be instantiated by the following primitive.

```
(define (new-instance-part class)
 (let ((instantiator (method-lookup class 'instantiator)))
   (instantiator)))
```

`New-instance-part` simply extracts the instance-description of the class. This instance description is then instantiated. The parameter `class` refers to the meta-object, which represents the class. In section 5.3 it is described how entire (non-meta) classes can be instantiated.

I will assume that there is *exactly one* instance of a metaclass:

```
(define a-class (new-instance metaclass-name))
```

Given this assumption it might be tempting to define metaclasses directly as "singular instances":

```
(define a-class
 (let ((self ...)
       (super super-metaclass))
  (let ((class-variable init-value)
        ...)
  ...)
 self))
```

If, however, metaclasses are to take part in a metaclass hierarchy, the parts of a metaclasses will, following this approach, be shared with other metaclasses. This is probably not what we want. Consequently we will leave it as a discipline only to instantiate metaclasses once.

5.2 The most General Parts of the Class Hierarchy

Let us now illustrate how to construct the most general classes and metaclasses of the class hierarchy. The purpose of this exercise is twofold. First, it allows us to experiment with different variations of the hierarchy. Second, it gives us first hand experience with the problems of dealing with the cyclic dependencies among the fundamental classes in an object-oriented programming environment.

The class hierarchies to be simulated are shown in figure 5.2. `A` and `B` are two sample classes defined by the user. `Object` is the root of both the class hierarchy and the metaclass hierarchy. As in Smalltalk, I will assume that the metaclass hierarchy is parallel with the class hierarchy. As a matter of naming, a metaclass of a class `X` is called `X-class`. Furthermore the class `class-class` is an abstract class, which is a superclass of all metaclasses. `Metaclass` is the class of the metaclasses (i.e., the metaclasses are considered as instances of `metaclass`.) In turn, `metaclass` is considered to be an instance of itself.

In appendix A the full details of the classes and the metaclasses are listed. Here I will only concentrate on the following aspects:

Figure 5.2: A sample class and metaclass hierarchy.

1. The handling of the cyclic dependencies among the classes.
2. The implementation of a class instantiation method, `new`, in `class-class`.

Point number two is postponed to section 5.3.

As can be seen from figure 5.2, `object` is an indirect superclass of `object-class`. Thus, `object` must exist at the time `object-class` is instantiated. On the other hand, `object` is an instance of the metaclass `object-class`. Consequently, `object-class` must exist at the time `object` is instantiated. This cyclic dependency means that it isn't possible to create `object` and `object-class` in the same way that we later will create, say, `A` and `A-class`.

The overall strategy for the construction of the class hierarchy in figure 5.2 can be described in the following way:

1. A temporary version of the metaclass `object-class` is defined. In this metaclass definition, there is no link to `class-class`.
2. The temporary version of the metaclass `object-class` from step 1 is instantiated, thereby creating the object, which represents the class `object`.
3. The `class-class` part and the `object` part of `object` are created. This is possible because `object` exists as the result of step 2.
4. `Object` is repaired such that its `super` refers to the instance of `class-class` created in step 3.

In the rest of this section I will show some more details of the construction of the hierarchy from figure 5.2. Readers who are not interested in these details can skip the rest of this section.

First, the temporary version of the metaclass `object-class` is defined:

```
(define (object-class-temporary)
 (let ((self nil)
       (super ()))  ; assigned in the method fix-super

   (define (fix-super super-part)
      (set! super super-part)
      'done)

   (define (instance-description)
     (let ((self nil)
           (super ())   ; empty list because root of hierarchy
       ...)))

  ...
  (set! self outer-dispatch)
  self))
```

The method `fix-super` is supposed to be activated on an instance of `object-class-temporary`, in order to make the connection to the more general parts of an `object-class` instance. `Class-class` is defined in the following way:

```
(define (class-class)
  (let ((self nil)
        (super (new-instance-part object)))

    (let ((instances () ))  ;; a list of instances made by new

      (define (new)
         )  ; described in section 5.3

      ...

      (set! self outer-dispatch))
      self))
```

We are now in a position where we can create the class `object` by instantiating the temporary metaclass `object-class-temporary`

```
(define object (new-part object-class-temporary))
(send object 'fix-super (new-part class-class))
(virtual-operations object)
```

The first line sets up an object without a link to a super class. This creates an object which among other messages responds to `fix-super`. Next we send the message `fix-super` to `object` with an instance of `class-class` as a parameter. Hereby the super part of `object` becomes a `class-class` part.

When we in the following instantiate `object-class` (in the process of instantiating a "user defined" metaclass) we certainly expect the instantiation to have an `object`-part, a `class-class`-part, and an `object-class` part. Therefore we substitute the definition of `object-class-temporary` with

```
(define (object-class)
  (let ((oc (new-part object-class-temporary)))
    (send oc 'fix-super (new-part class-class))
    oc))
```

5.3 Instantiation of Classes via Message Passing

Let us finally deal with the instantiation of classes. In this section we are only concerned with instantiation of non-metaclasses. Recall that a meta class is supposed to be instantiated only once, via the use of the primitive `new-instance`.

As already mentioned, an important goal of having metaclasses at all is to be able to instantiate classes by sending messages to the objects that represent classes. If `A` refers to the meta-object, which represents the class A, the following shows how we want to instantiate A.

```
(define an-A (send A 'new))
```

The natural place of the method, which responds to the message `new`, is in `class-class`, because it is a superclass of all metaclasses. Let me explain what happens in the method `new` of class `class` during the instantiation `(send A 'new)`. The method `new` is defined in the following way:

```
(define (new)
   (let ((instance
           (new-instance
             (method-lookup self 'instantiator))))
      (set! instances (cons instance instances))
      instance))
```

Figure 5.3 shows the parts of the object, to which the `new` message is sent. `Self` refers to the most specific part of the object, because we are based on the interpretation of `self` from section 3.3 (virtual-like methods). Because of this interpretation of `self`, the instantiator passed to `new-instance` is that belonging to the metaclass of A. Next, `new-instance` allocates the new instance, and it arranges for virtual operations of the object. The assignment of `instances` puts the newly allocated

Figure 5.3: The parts of an A class and messages involved in its instantiation.

instance into a list of instances, which is kept as a class variable of `class-class`. Finally the new instance is returned.

Figure 5.3 also shows the list of messages that are involved in the creation of a new A object (relative to figure 5.2). The first category of messages locates and returns the method `new` of `class-class`. The second group of messages stems from the actviation of `new-instance`. The instantiators of `A-class`, `B-class`, as well as `object-class` are located and used during the instantiation. Finally new-instance sends the message `set-self!` to the new object. This starts the already described chain reaction of `set-self!` messages (see section 3.3).

5.4 Support of Metaprogramming

The metaclasses constitute the natural place of methods that reflect some knowledge of the classes. The knowledge that I have in mind is, for instance, the list of methods, the list of instance variables, and the list of class variables.

Given a class, say `A` from the previous section, it should be possible to say

```
(send A 'method-list)
```

for hereby to get a list of methods of `A` (or perhaps of `A` and all its superclasses).

It is difficult to obtain a reasonable support of this kind of functionality in Scheme. It would, of course, be possible to keep track of the constituents of the class in a manual way, but this is hardly attractive. If a procedure has access to its own syntactic structure (the lambda expression on list form) it is easy to extract the desired information. In Scheme it is not possible to get access to this kind of "meta knowledge" of procedures.

Finally, if classes and methods syntactically are defined via a macro interface, it would be possible to generate the metaprogramming interface, because the macros have access to the necessary syntactic constituents.

Chapter 6

Conclusions

The most important conclusion of this work is the ease and relative elegance with which it is possible to simulate many important and advanced object-oriented mechanisms in Scheme.

The purpose of making simulations in Scheme, along the lines described in this report, is to get quick and practical experience with new ideas in the field. The purpose is not to make an object-oriented programming language nor to make object-oriented applications.

At the more concrete level I find the precedence list representation of objects to be interesting, especially in connection with multiple inheritance. The possibility to have generic procedures instead of message passing is also interesting. Furthermore, I find that the straightforward simulation of method-combination is noteworthy. The caching of effective methods is very simple to establish, but it is a weakness that the caching has to be done on instances, and not on classes. The simulation of metaclasses is complicated, but nevertheless I find that it has been quite rewarding to solve some of the "classic" problems in the Scheme framework.

As emphasized several times in the report, it is not the purpose of this work to establish a new object-oriented programming language based on Scheme. However, it is clearly possible to do so by defining some appropriate syntactical abstractions (macros) for selected simulation patterns.

Appendix A

Program Description of Metaclasses

In this appendix the detailed program description of the metaclasses from section 5.2 is shown.

```
(define (object-class-temporary)

  ;; class variables
  (let ((self nil)  ;; must be assigned to dispatch
        (super ())) ;; Super must refer to class, which in turn refers
                    ;; to object. Fixed via the method fix-super.

   ;; class variables
   (let ()

    ;; description of instances
     (define (instance-description-object)
       (let ((super ())
             (self nil))

         (define (set-self obj-part)
           (set! self obj-part))

         (define (responds-to operation)
           (let ((method (self operation)))
             (if method #t #f)))

         (define (id)
           "I am an object instance")

         (define (class-of)
           outer-dispatch-object)
```

```
      (define (inner-dispatch-object op)
        (cond ((eq? op 'set-self!) set-self)
              ((eq? op 'responds-to?) responds-to)
              ((eq? op 'class) class-of)
              ((eq? op 'id) id)
              (else ())))

      (set! self inner-dispatch-object)
      self))


  ;; class methods

  (define (fix-super super-part)
    (set! super super-part)
    'done)

  (define (class-of)
    metaclass)

  (define (id)
    "I am object")

  (define (set-self obj-part)
    (set! self obj-part)
    (send super 'set-self! obj-part))

  (define (outer-dispatch-object m)

    (cond ((eq? m 'instantiator) instance-description-object)
          ((eq? m 'class) class-of)
          ((eq? m 'set-self!) set-self)
          ((eq? m 'id) id)
          ((eq? m 'fix-super) fix-super)
          (else (method-lookup super m))))

  (set! self outer-dispatch-object)

  self)))




(define (metaclass-class)

  ;; class variables
  (let ((self nil)
```

```
        (super (new-instance-part object)))

   ;; class variables
   (let ()

    ;; METACLASS DOES NOT HAVE AN INSTANCE DESCRIPTION

    ;; class methods

    (define (class-of)
      self)  ;; the circularity of the is-a relation.

    (define (id)
      "I am metaclass")

    (define (set-self object-part)
      (set! self object-part)
      (send super 'set-self! object-part))

    (define (outer-dispatch-metaclass m)

      (cond ((eq? m 'class) class-of)
            ((eq? m 'set-self!) set-self)
            ((eq? m 'id) id)
            (else (method-lookup super m))))

    (set! self outer-dispatch-metaclass)
    self)))
```

```
(define (class-class)

  ;; class variables
  (let ((self nil)
        (super (new-instance-part object)))

   ;; class variables
   (let ((instances ()))

    ;; THERE IS NO INSTANCE-DESCRIPTION OF THIS CLASS.

    ;; class methods

    (define (new)
      (let* ((instance
               (new-instance
```

```
                 (method-lookup self 'instantiator))))
        (set! instances (cons instance instances))
        instance))

    (define (number-of-instances)
      (length instances))

    (define (class-of)
      metaclass)

    (define (id)
      "I am class")

    (define (set-self object-part)
      (set! self object-part)
      (send super 'set-self! object-part))

    (define (outer-dispatch-class m)

      (cond ((eq? m 'new) new)
            ((eq? m 'class) class-of)
            ((eq? m 'set-self!) set-self)
            ((eq? m 'id) id)
            ((eq? m 'instances) number-of-instances)
            (else (method-lookup super m))))

    (set! self outer-dispatch-class)
    self)))



(define object (new-part object-class-temporary))
(send object 'fix-super (new-part class-class))
(virtual-operations object)


(define metaclass (new-instance metaclass-class))

(define (object-class)
  ;; return an object class, where the super is fixed.
  (let ((oc (new-part object-class-temporary)))
    (send oc 'fix-super (new-part class-class))
    oc))
```

Bibliography

[1] H. Abelson, Sussman G.J., and Sussman J. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] Norman Adams and Jonathan Rees. Object-oriented programming in scheme. In *Conference Record of the 1988 ACM Symposium on Lisp and Functional programming*, pages 277–288, 1988.

[3] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonay E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification X3J13 document 88-002R. *Sigplan Notices*, 23(Special Issue), September 1988.

[4] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Common base language. Technical report, Norwegian Computing Center, October 1970.

[5] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[6] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989.

[7] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: an object-oriented scheme with first class types. In Norman Meyrowitz, editor, *OOPSLA'86*, pages 30–37, 1986. Special issue of of Sigplan Notices, volume 21, number 11, November 1986.

[8] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: An object-oriented dialect of scheme. *Lisp and Symbolic Computation*, 1(1):37–51, June 1988.

[9] J. Rees and W. Clinger. Revised^3 report on the algorithmic language Scheme. *Sigplan Notices*, 21(11), 1986.

[10] Jonathan A. Rees and Norman I. Adams. T: A dialect of lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

[11] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual.* Computer Science Department, Yale University, New Haven CT, September 1988.