A Study of Compile-time Metaobject Protocol

コンパイル時メタオブジェクト・プロトコルに関する研究

Shigeru Chiba

千葉　滋

A Study of Compile-time Metaobject Protocol

コンパイル時メタオブジェクト・プロトコルに関する研究

By

Shigeru Chiba
chiba@is.s.u-tokyo.ac.jp

November 1996

A Dissertation Submitted to
Department of Information Science
Graduate School of Science
The University of Tokyo

In Partial Fulfillment of the Requirements
For the Degree of Doctor of Science.

A Metaobject Protocol for Enabling Better C++ Libraries

Shigeru Chiba

Department of Information Science
The University of Tokyo
chiba@is.s.u-tokyo.ac.jp

Abstract

C++ cannot be used to implement control/data abstractions as a library if their implementations require specialized code for each user code. This problem limits programmers to write libraries in two ways: it makes some kinds of useful abstractions that inherently require such facilities impossible to implement, and it makes other abstractions difficult to implement efficiently.

The OpenC++ MOP addresses this problem by providing libraries the ability to pre-process a program in a context-sensitive and non-local way. That is, libraries can instantiate specialized code depending on how the library is used and, if needed, substitute it for the original user code. The basic protocol structure of the OpenC++ MOP is based on that of the CLOS MOP, but the OpenC++ MOP runs metaobjects only at compile time. This means that the OpenC++ MOP does not imply runtime penalties caused by dispatching to metaobjects.

Biographical Sketch

Shigeru Chiba was born in Tokyo, Japan, in 1968. After living in Osaka for a few years, he spent most of his childhood in Yokohama, a suburb of Tokyo. As many other children living in a suburb of Tokyo do, he went to a high school in Tokyo, and first met meta computation there when his good friend at the high school let him borrow a book, *Gödel, Escher, Bach: an Eternal Golden Braid*, by Douglas R. Hofstadter. Looking back now, it was natural that he was captured by the idea of meta computation, which was plainly explained in the book.

But the life at university was attractive enough to have him forget meta computation. It had actually gone from his mind until it emerged again in front of him at the department of information science. He was supposed to walk away from it, but what he really did was to dance along a circle and come back to the same place. It might be his karma although he hopes he drew not a circle but a helix.

The second encounter eventually took him to California, where a guru is living. He has not yet known what is the next.

Acknowledgments

This thesis was supervised by Gregor Kiczales. I would like to express my deep gratitude to him. I also profoundly thank Takashi Masuda, my supervisor at The University of Tokyo. He continuously supported and encouraged me during this work. John Lamping gave me a plenty of constructive comments and criticism, which greatly helped me write this thesis. Finally, I greatly thank my thesis committee, Akinori Yonezawa, Kei Hiraki, Yoshio Oyanagi, Masami Hagiya, Hiroshi Imai, and Naoki Kobayashi.

I am grateful for financial support received from Japan Society for the Promotion of Science (Nihon Gakujyutsu Sinkoukai), from The Information-technology Promotion Agency, Japan (IPA), from Xerox Corporation, from Japan Foundation for the Education and Training of Information Processing (Jyouhousyori Kyouiku Kensyu Jyosei Zaidan), and from International Information Science Foundation, Japan (Jyouhou Kagaku Kokusai Koryu Zaidan). Also, Department of Information Science at The University of Tokyo let me extend my stay at PARC with exceptional courtesy.

This thesis project was conducted at Xerox Palo Alto Research Center (PARC) in Palo Alto, California, U.S.A.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In computing science, elegance is not a luxury
but a matter of life and death.
— E. W. Dijkstra

Execution speed is the only metric in computer science.
— Kei Hiraki

One of today's significant concerns in software industry would be to decrease time and costs of software development. Anybody would agree that good libraries promote code reuse and thereby contribute to rapid and low-cost software development. This thesis deals with a language mechanism for programmers to write good libraries.

What are good libraries? Good libraries provide useful control/data abstractions, which are commonly used for a number of applications, high-level to improve readability of programs, and simple and intuitive to avoid leading the library users to misuse the libraries and cause serious errors. Also, efficiency is another criterion of good libraries. Since writing a good library is a very difficult task, the programmer is required to have not only good programming skill but also deep knowledge about the application domain in that the library helps programmers. For example, library developers need to know typical functions and data structures used in that domain.

Development of good libraries also needs assistance of language designers, who should provide language mechanisms for writing good libraries. This is a more realistic option for the designers than including all desirable control/data abstractions in the language specifications, since the designers cannot expect every desirable abstraction in advance and because the language would be too complex if built-in language mechanisms cover all the desirable abstractions, such as I/O, persistency, distribution, concurrency, and so on. Programming languages should have a minimum set of built-in mechanisms and most of abstractions should be supplied by libraries.

Motivating problem

Language mechanisms that have been developed so far for writing libraries are not powerful enough to support every desirable control/data abstractions. For example, the C++ language [58] is one of languages that have a richest set of language mechanisms, but a number of useful abstractions cannot be included in a C++ library with satisfying ease-of-use and efficiency. As shown in Chapter 2, C++ programmers cannot write a library class that gives distribution extension to its subclasses.

Our observation on this problem is that such abstractions as distribution support are used to give extended features to another abstraction, and thus their implementations are tightly tangled with the implementation of the other abstraction. Those abstractions need a different implementation if they are used with a different abstraction. This means that those abstractions are difficult to include in a library, which is an independent software component and can provide only a single general-purpose implementation.

There are other kinds of abstractions the implementations of which are tightly tangled with other parts of the program. Abstractions such as a vector data type can be provided by a library but the implementation that the library can supply is not efficient. An implementation specialized for a particular user program can improve the execution performance although that implementation is effective only for the particular program and thus it cannot be included in the library.

This problem can be avoided if programming languages provide a mechanism for library developers to supply specialized code through a library to a particular user program. Current language mechanisms, however, allow only limited kinds of specialization of library code. For example, C++'s template mechanism allows only type parameterization; library developers can specialize only type names appearing in library code for a particular user program.

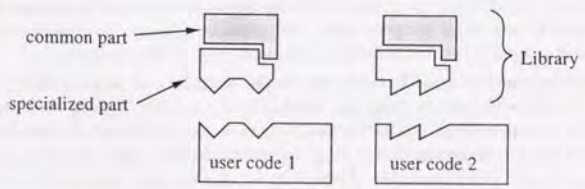

Figure 1.1: The common part and the specialized part of a library

Solution by this thesis

To solve the problem mentioned above, this thesis proposes a new language mechanism for enabling libraries to supply specialized code and include more useful control/data abstractions. With the proposed mechanism, programmers can build a library consisting of two parts: commonly-used code and specialized code for every user program (Figure 1.1). The commonly-used code is supplied *as is* to the user program, but the specialized code is automatically generated on demand for a particular user program. The specialized code fills a gap between the common part and the user program.

The generation of the specialized code is *programmed* by the library developer. The proposed mechanism allows to preprocess a program with interacting with a collection of code such as class definitions and member function calls. For example, it provides the ability to insert specialized code, to rewrite class definitions, to substitute different code for member function calls, and so on.

The language mechanism proposed by this thesis is the OpenC++ MOP [11], which is a metaobject protocol for C++. OpenC++ is the name of a version of C++ language with that metaobject protocol. This mechanism provides class metaobjects, which are regular objects representing a class, so that library developers can program the generation of specialized code. Although the class metaobjects might seem similar to Smalltalk's class objects [26], the class metaobjects receive source code at compile time and preprocess it if needed. The programmers can define a new metaclass (i.e., a class for class metaobjects) and thereby they can program desired preprocessing of code involved with the class. The reason that the proposed mechanism is called a metaobject protocol is that it is essentially a protocol for defining and accessing metaobjects. Protocol is the Smalltalk terminology and it means object interface.

The OpenC++ MOP is a more powerful mechanism than other similar mechanisms like Lisp macros. Unlike Lisp macros, the OpenC++ MOP provides contextual information of the processed code other than a abstract syntax tree. The contextual information includes the type of a variable, class members, a base class, and so forth. This feature makes it possible to perform context-sensitive preprocessing. Furthermore, implementing non-local processing is easy with the OpenC++ MOP. Preprocessing for implementing abstractions often spreads out over the whole source code, but the description of the preprocessing involved with a single class is centralized into the class metaobject. Since all code fragments are automatically dispatched according to the static types to an appropriate class metaobject, programmers can easily specify preprocessing that is effective only for a particular class.

Preceding techniques known as reflection has a notable influence on the OpenC++ MOP. Especially, we took the basic structure of the metaobject protocol from the CLOS MOP [36], while we took the basic architecture

from Lisp macros. Thus the OpenC++ MOP is also metacircular as the CLOS MOP is, and it is easy to learn and to write an efficient meta-level program. The difference from the CLOS MOP is that the OpenC++ MOP employs static typing and executes class metaobjects only at compile time; this means that it does not involve runtime penalties due to class metaobjects. Since CLOS is a dynamically-typed languge and thus the CLOS MOP executes metaobjects at runtime, avoiding runtime penalties in the CLOS MOP requires complex implementation techniques.

The structure of this thesis

From the next chapter, we present background, design details, and applications of the OpenC++ MOP. The structure of the rest of this thesis is as follows:

Chapter 2: Limitations of C++

We first discuss limitations of current language mechanisms provided by C++ for library developers. We illustrate that the inheritance mechanism and the template mechanism do not work for including some kinds of desirable abstractions in a library. And we claim that this problem is caused by the lack of the capability of C++ to preprocess a program in a context-sensitive and non-local way and instantiate specialized code for a particular program.

Chapter 3: Techniques for Processing a Program

Next, we overview existing mechanisms that other programming languages provide for processing a program. As the representatives, we show Lisp macros, 3-Lisp, and the CLOS MOP. The feature shared by the three systems is that they provide meta representation of programs for the programmers. We overview detailed architectures of the systems and discuss their pros and cons. The bottom line of the discussion is that metacircularity of the CLOS MOP and compile-time executability of Lisp macros are preferable properties.

Chapter 4: The OpenC++ MOP

On the basis of the discussion in the previous chapter, we proposes a new C++ mechanism for processing a program. This mechanism is the OpenC++ MOP and it has the two preferable properties discussed in the previous chapter: metacircularity and compile-time executability. The OpenC++ MOP allows programmers to process a program in a context-sensitive and non-local way in order to include useful abstractions in a library that regular C++ cannot handle. We also mention comparison between the OpenC++ MOP and other early compile-time MOPs.

Chapter 5: Meta Helix

Although metacircularity is a good property, pure metacircular systems can lead to a problem we called implementation level conflation. This problem, found in the CLOS MOP, confuses programmers and often causes programming errors such as circular definition. To avoid this problem, we – a MOP designer – has adopted an improved version of metacircular architecture for the OpenC++ MOP. We present that this improved architecture, named the meta helix, preserves advantages of metacircularity and also addresses implementation level conflation.

Chapter 6: Libraries in OpenC++

The OpenC++ MOP makes it possible to include a number of control/data abstractions in a library. We show examples of these abstractions in this chapter. The abstractions shown here include abstractions that regular C++ cannot handle, a metaclass library for helping write a new metaclass, abstractions implemented by meta-metaclass for facilitating meta-level programming, and abstractions that the OpenC++ MOP makes more efficient than in regular C++.

Chapter 7: Conclusion

Finally, we conclude this thesis in Chapter 7. We present contributions of this thesis and future directions.

Chapter 2

Limitations of C++

C++ cannot pre-process a program in a context-sensitive and non-local way. The lack of this capability makes it impossible to include some useful control/data abstractions in a library, or makes it difficult to implement some abstractions efficiently as a library. This problem is solved if C++ has a mechanism to process a user program and allow the library to supply specialized code for a particular user program. Single general-purpose codes cannot implement those abstractions or have difficulty in making the abstractions efficient.

Existing C++ mechanisms for building a library — inheritance or templates — do not provide this capability sufficiently. This means that some practically important abstractions have not been included in a library with an ideal interface and efficiency. In fact, some abstractions have been even provided by a specialized C++ language in which the abstractions are embedded in, or implemented with a dedicated code generator. For example, in the academic world, a number of distributed C++ languages have been developed for making distributed objects available [48, 28]. In industry, programmers who want to use distributed objects have needed to write an extra program in an IDL (interface description language) and combine the code generated from the extra program with their C++ programs.

This chapter presents that C++'s inheritance mechanism or the templates mechanism do not work for including some kinds of abstractions in a library. Then it mentions that, to do this, C++ needs a mechanism for processing a user program in a context-sensitive and non-local way. This mechanism makes it possible to develop a library including the abstractions that regular C++ cannot handle.

2.1 Inheritance

The most basic mechanism C++ provides for library developers is instantiation. The idea is that the library writer writes a reusable class, which the

library user can instantiate. This mechanism is limited in that it enables only abstract data types (ADTs); the library user have to use a library class *as is*. Even if the library class does not exactly fit her requirements, she cannot change the definition of the class at all, and thus she might need to write a new class even though a library provide a similar class. This means that the reusability of the library is significantly limited.

Inheritance is a more powerful mechanism for developing a good library. It allows the library user to incrementally define a new class extending a class provided by the library. However, the ability of the inheritance mechanism to reuse part of a library class is significantly limited. The inherited code must be always the same; the library cannot supply different code to different subclasses.

This limitation makes it impossible to develop a library that provides certain kinds of important abstractions such as distributed objects. These abstractions require that the inherited code be *adapted* to the user program. This section discusses this limitation of the inheritance mechanism and begins to outline what kind of support for adaptation of reused code needs to be provided.

Distributed objects

Distributed objects are extended objects, which are accessible over the network without concern for their location. An ideal library for distributed objects might allow the user to write something like this:

```
class Point : public Distribution {
public:
    int x, y;
    void Move(int nx, int ny) { x = nx; y = ny; }
};
```

This program defines a class of distributed objects called `Point`. The class `Point` inherits from the `Distribution` library class, and this is what gives makes it be a distributed class. The user can deal with `Point` objects as regular objects even if they are on a remote machine:

```
Point* p = ...;
p->Move(3, 11);
```

In this ideal library, no special syntax is required to call a member function for a `Point` object p that might be on a remote machine.

Unfortunately, this ideal class `Distribution` is not feasible in regular C++ because it cannot supply all the member functions that the subclass `Point` needs to inherit. For example, to make the member function `Move()` callable from a remote machine, the class `Point` needs to inherit a member

function for marshaling arguments to `Move()`. This marshaling function converts the arguments into a byte stream, which lower-level network routines can directly handle.

Within the confines of the inheritance mechanism, however, the class `Distribution` cannot supply the marshaling function to the subclass `Point`. Since the marshaling function for `Move()` performs a kind of type conversion, its implementation strongly depends on the signature of `Move()`, such as the number of arguments and their types. But the inheritance mechanism does not allow the class `Distribution` to alter the implementation of the marshaling function to adapt to the signature of `Move()`.

Named objects

Since the example of the distributed objects is too complex to show the details here, we instead use a much simpler example and articulate limitations of the inheritance mechanism and what C++ needs to handle some kinds of abstractions that it cannot currently handle. The problem we showed in the example of the distributed objects is that a super class cannot supply some kinds of member functions to a subclass even though supplying them seems desirable from the library users' viewpoint.

To discuss this problem, let's implement a simple library with only the inheritance mechanism. This simple library allows the users to get the class name of an object at runtime. This function is one of requirements of building the distributed object library. The users may write something like the following program:

```
class Complex : public NamedObject {
public:
    double r, i;
};

void f(Complex* x)
{
    cout << "x is " << x->ClassName();
}
```

If invoked, `f()` displays "`x is Complex`". The class `NamedObject` is a library class, which supplies a member function `ClassName()` to `Complex`.

Although this particular function is already provided in regular C++ by the `typeid` operator of RTTI [59], we assume that there is no RTTI (Run-Time Type Information) in C++ and try to implement this function with the inheritance mechanism. Unfortunately, this is impossible because of limitations of the inheritance mechanism. For example, the following definition of `NamedObject` works for the class `Complex` but not others:

```
class NamedObject {
public:
```

```
    virtual char* ClassName() { return "Complex"; }
};
```

This version of the class NamedObject supplies a member function ClassName(), which returns a character string "Complex", but this implementation of ClassName() is obviously wrong. If another class Real inherits from NamedObject and the member function ClassName() is called for a Real object, ClassName() returns an inappropriate character string "Complex". To make ClassName() work for the class Real, the implementation of ClassName() should be:

```
virtual char* ClassName() { return "Real"; }
```

However, the class NamedObject cannot switch the implementations of ClassName() to make it work for different subclasses. It has to select either of the implementations and supply the selected one to all the subclasses.

We cannot implement the ideal version of the class NamedObject in regular C++. To make it feasible, we have to change the specifications of the library, but this change also makes the library less easy to use. For the new version of the library, the definition of the class NamedObject is as follows:

```
class NamedObject {
public:
    virtual char* ClassName() = 0;  // not implemented
};
```

The class NamedObject does not supply a concrete implementation of ClassName() any more. ClassName() is implemented by a subclass of NamedObject. This means that the library users have to implement ClassName() by hand for their classes:

```
class Complex : public NamedObject {
public:
    double r, i;
    virtual char* ClassName() { return "Complex"; }
};

class Real : public NamedObject {
public:
    double value;
    virtual char* ClassName() { return "Real"; }
};
```

These specifications of the library are quite unsatisfying in terms of ease of use. Note that the definitions of the user classes Complex and Real now include the implementation of the member function ClassName(). In the ideal specifications, this should be supplied by the library class NamedObject.

What does C++ need?

The example of NamedObject shows us that C++ lacks the ability to allow libraries to supply code customized for the user programs. With the inheritance mechanism, the class NamedObject cannot provide an ideal interface because it cannot supply a differently implemented member function ClassName() to a subclass such as Complex and Real.

This problem is solved if C++ provides a mechanism for programmers to *program* source-code processing and include "the program" in a library. For example, the developer of the named object library would write a program to process the user program at source-code level and automatically insert the implementation of ClassName() customized for a subclass such as Complex and Real. Then the library user could avoid implementing ClassName() by hand for her class. She could write something like this:

```
class Complex : public NamedObject {
public:
    double r, i;
};
```

Note that the member function ClassName() is not included by the definition of the class Complex since it is automatically supplied by the library. The named class library reads this program before compilation and translates it into this program:

```
class Complex : public NamedObject {
public:
    double r, i;
    virtual char* ClassName() { return "Complex"; }
};
```

The implementation of ClassName() is inserted by the library. Note that the inserted implementation is specialized for the class Complex. It is not effective for other sibling subclasses like Real. The library class NamedObject declares the member function ClassName() but it does not implement it:

```
class NamedObject {
public:
    virtual char* ClassName() = 0;  // not implemented
};
```

This is the same definition that we presented for the feasible but unsatisfying version of the named object library.

The macro mechanism might seem an alternative to the mechanism proposed above, but the macro mechanism allows very limited kinds of source-code processing, such as simple word-by-word replacement and concatenation of words. To implement the named object library with ideal interface, C++ is required to have a more powerful and sophisticated mechanism, which should satisfy the following two criteria:

• **Context-sensitive**

How a program is processed should be determined with referring various contextual information of the processed program. The contextual information includes what RTTI (Run-Time Type Information) already provides but not limited. It is a class definition, type information, program text, and so on. Providing contextual information means that the information can be used even if it is defined at other locations than where the source-code processing happens. For example, if an assignment expression to a variable is processed, the type of the variable may be declared at a different location.

Context-sensitivity is a crucial property of the proposed mechanism since the processing by the macro mechanism is independent of the contextual information of the processed program. Context-sensitivity gives an advantage against the macro mechanism to the proposed mechanism.

In the example of the named object library, the library has to examine the name of the subclass that inherits from the library class `NamedObject`. This name is directly embedded in the implementation of the member function `ClassName()`, which are inserted in the definition of the subclass. The use of contextual information is more significant in the example of the distributed objects. For example, to supply a marshaling function, the library has to examine the signature of a member function and specialize the implementation of the marshaling function for the particular member function.

• **Non-local**

The proposed mechanism should make it easy to program not only local processing but also non-local one, which needs to deal with many code fragments spread out in various locations of the processed program. For ease of programming, locations where the processing must happen should be specified in a declarative way. If the programmer has to explicitly specify every location, the programming would be extremely difficult and unrealistic.

For the example of the named object library, the library has to read the whole program and insert the member function `ClassName()` in the definitions of all the subclasses of `NamedObject`. The mechanism should help the programmer easily specify locations to insert `ClassName()`. For example, it should allow the programmer to direct something like "insert `ClassName()` in a class declaration if the class is a subclass of `NamedObject`."

C++'s macro functions are rather a mechanism for local processing. It processes only places where a macro name appears, and so programmers have to place the macro name by hand wherever the processing is needed. For example, if the named object library is developed with a macro function, the library users would need to write an awkward program like the following:


```
class Complex : public NamedObject {
public:
    double r, i;
    EXPAND_CLASSNAME_HERE(Complex)
};
```

The macro function is `EXPAND_CLASSNAME_HERE`, which is expanded into an appropriate implementation of `ClassName()`. The library users have to explicitly insert the macro in the definition of all the subclasses of `NamedObject`, but inserting it in all the places by hand is quite error-prone.

2.2 Template

In the previous section, we presented that a library including some kinds of abstractions need to supply customized code for a particular user program. The template mechanism of C++ achieves this ability to a certain degree, but it cannot be a general solution of our problem. In this section, we mention the ability that the template mechanism provides and then we present why the mechanism is not a general solution.

Vector

With the template mechanism, programmers can write a library that supply customized code for a user program although the range of customization is limited. For example, the template mechanism enables a library for a vector abstraction. With this library, the users can write a program like this:

```
Vector<int> v1, v2, v3, v4;
    ⋮
v1 = v2 + v3 + v4;
```

The variables `v1`, `v2`, `v3`, and `v4` are vectors of integers. Using vectors for other types is also easy. If the users want to deal with vectors of characters, they should say `Vector<char>` instead of `Vector<int>`.

The inheritance mechanism does not enable such a useful vector library. If the vector library is implemented with the inheritance mechanism, the library users need to define a new subclass for every vector for a different type. This results from the same reason why the named object library is not feasible; to make vectors for various types available without subclassing, the vector library has to alter the implementation, depending on whether the vector is for integers or characters.

The template mechanism can absorb the difference of implementations if the difference is type names. It directs the C++ compiler to automatically produce the implementation adapted for a particular user program. The following program is the definition of the vector abstraction, which should be included in the vector library:

```
template <class T> class Vector {
    T elements[SIZE];
public:
    Vector operator + (Vector& a, Vector& b) {
        Vector c;
        for(i = 0; i < SIZE; ++i)
            c.elements[i] = a.elements[i] + b.elements[i];

        return c;
    }
        ⋮
};
```

For simplicity, this example assumes that the length of a vector is always SIZE. In the Vector template above, T is a type parameter. All occurrences of T in the template are replaced with an actual type given by a user program when the compiler produces the actual implementation of the vector abstraction. For example, the implementation that the compiler produces for Vector<int> is equivalent to this (pseudo) class definition:

```
class Vector<int> {
    int elements[SIZE];
public:
    Vector<int> operator + (Vector<int>& a, Vector<int>& b) {
        Vector<int> c;
        for(i = 0; i < SIZE; ++i)
            c.elements[i] = a.elements[i] + b.elements[i];

        return c;
    }
        ⋮
};
```

Limitations of templates

The vector abstraction can be included by a library with ideal interface if the template mechanism is used, but this implementation of the vector abstraction is not satisfying with respect to execution performance. Consider how this expression is executed if the vector abstraction is implemented as we showed above:

```
v1 = v2 + v3 + v4;
```

Since the + operator is overridden for Vector<int>, the execution of this expression is divided into two function calls, each of which executes a for loop to compute an addition of two vectors. The called function receives two vectors, executes the for loop to compute the addition of each vector element, and returns the result.

To compute v2 + v3 + v4, this implementation eventually needs two for loops from the first through the last element of the vector, but executing this loop twice is redundant. The whole expression should be computed by the following more efficient implementation:

```
for(i = 0; i < SIZE; ++i)
    v1.elements[i] = v2.elements[i] + v3.elements[i]
                   + v4.element[i];
```

This implementation executes the loop only once, and it directly sets the summation of the three elements to v1.

This inefficiency is not due to the mechanism of operator overloading. Rather, it should be thought that is caused by limitations of the template mechanism. Operator overloading is a mechanism for syntax sugar, and the problem is not solved even though various kinds of operators can be overloaded. For example, suppose that the programmer can overload the three-operands + operator. Then the developer of the vector library would include the following code in the library:

```
template<class T>
Vector<T> operator + (Vector<T>& a, Vector<T>& b, Vector<T>& c)
{
    Vector<T> v;
    for(int i = 0; i < SIZE; ++i)
        v.element[i] = a.element[i] + b.element[i]
                     + c.element[i];

    return v;
}
```

This operator function executes such an expression as v2 + v3 + v4 more efficiently, but the library is still inefficient to deal with other kinds of expressions such as v2 + v3 - v4 and v2 + v3 + v4 + v5. The library developer cannot overload all combinations of operators in advance.

C++ needs a mechanism for processing a program

The template mechanism cannot enable the efficient implementation, which executes a for loop only once. This is because of the limitations of the ability of the template mechanism to supply code adapted for a user program. The only adaptation that the template mechanism can perform is to simply fill out parameterized fields in the template with given type names.[1] This is not sufficient to implement the vector abstraction efficiently.

In general, the template mechanism is not suitable for this kind of interprocedure optimization, which needs to process several operations at a time and substitutes specialized code for all the operations. To do this, C++

[1] A template parameter may be not only type names but also any constant value.

needs to be able to handle context of the processed operations: what the group of operations are computing. If C++ has a mechanism to process a program with that context sensitivity, then the user program:

```
Vector<int> v1, v2, v3, v4;
    ⋮
v1 = v2 + v3 + v4;
```

can be translated by the vector library before compilation into:

```
Vector<int> v1, v2, v3, v4;
    ⋮
for(i = 0; i < SIZE; ++i)
    v1.elements[i] = v2.elements[i] + v3.elements[i]
                   + v4.element[i];
```

After the translation, the efficient loop is substituted for the vector expression. The overloaded + operator function is not called any more, but the additions are computed by the inlined loop.

Context-sensitivity and non-locality are also significant for this processing as in the example of the named object library. First, context-sensitivity is needed to determine which expressions should be translated. Since the translation is applied only to vector expressions, the library needs to look up the type of a variable in an expression and determine whether the translation is applied or not. Second, this processing is non local; All vector expressions in the whole program needs to be found and translated into efficient code. Without appropriate supports for programmers to specify places at which the translation should be done, programing the translation for the vector library would be difficult and awkward.

Can optimizing compilers do the same thing?

In the example above, we merged two distinct function calls and inlined the resulting optimized loop. This way of performance improvement is regarded as optimization that regular C++ compilers can perform. As we show in Chapter 6, however, the inter-procedure optimization seems difficult for practical compilers to perform within reasonable time and space. Since inter-procedure optimization requires deep flow-analysis and very clever code generation, we should not expect that compilers perform all possible inter-procedure optimization in general. What we can expect is that compilers may perform the optimization for some typical patterns of program.

Although inter-procedure optimization is difficult for C++ compilers, it is often obvious and straightforward to perform from the programmer's viewpoint. Since the programmers know semantic information of the program, they can easily find possible inter-procedure optimization without

complex flow-analysis. An advantage of our proposed mechanism is that it allows library developers to implement ad-hoc optimization which is obvious to the developers but difficult for compilers to automatically perform. The library developers can include the program for source-code processing in a library so that a user program using the library is pre-processed and efficiently compiled by a back-end compiler.

2.3 Summary

This chapter mentioned that some useful abstractions cannot be included in a library, and others are difficult to efficiently implement within the confines of C++. Such abstractions require different implementations for different library user programs, but existing C++ mechanisms do not provide the ability to do that enough to implement those abstractions.

In this chapter, first, we showed that the inheritance mechanism does not enable developers to implement the distributed object library or the named object library. As for the distributed object library, the developer cannot define a library class `Distribution` so that it supplies subclasses with a marshaling function, which needs to be differently implemented for different subclasses. This is because the inheritance mechanism forces a library class to supply member functions to subclasses *as is* without any adaptation.

Then, we presented that the template mechanism provides the limited ability to supply different implementations for different user programs, but this ability is not powerful enough to implement some abstractions such as the vector abstraction. Because of the ability to supply different implementations, the vector library implemented with the template mechanism allows the users to easily deal with vectors for various types. However, this vector library is not efficient because of the limitations of the ability of the template mechanism. If the library can supply cleverer implementations customized for a particular user program, the provided vector abstractions would be more efficient.

To solve the problem above, this chapter claimed that C++ needs a more powerful mechanism for processing a program at source-code level. If this mechanism is available, programmers can develop a library that pre-processes a user program and inserts code needed for implementing an abstraction in an adapted way to the user program. Through the discussion with the concrete examples, we presented that the proposed mechanism should have two important properties: context-sensitivity and non-locality. First, a user program should be processed in a context-sensitive way. For example, to produce a marshaling function, the distributed object library needs contextual information of a user program such as signatures of member functions and type information of variables. Second, the proposed mechanism should make it easy for developers to program non-local processing,

which deals with code fragments spread out over the user program. For example, the efficient vector library has to find all vector expressions included in a user program and translate them into efficient loops. Without appropriate supports, programming such non-local processing would be difficult and error-prone.

Chapter 3

Techniques for Processing a Program

The previous chapter presented that C++ needs a more sophisticated mechanism for processing a program. The existing mechanisms, the inheritance mechanism or the template mechanism, do not enable context-sensitive or non-local processing. This chapter overviews currently known mechanisms for program processing and mentions their pros and cons. The basic idea shared by these mechanisms is to provide the *meta representation* of the programs. The meta representation gives programmers the capability for context-sensitive and non-local processing, which we need for C++.

As the representatives of these mechanisms, this chapter shows Lisp macros, 3-Lisp, and the CLOS MOP. We compare the meta representation they provide, and discuss pros and cons. The focus of this chapter is on illustrating essential ideas behind the mechanisms rather than showing the exact specifications. Hence the description in this chapter is not exactly faithful to the original syntax or specifications. We carefully alter the syntax and the specifications so as to help clarify the differences among the mechanisms but not to lose the essence of their ideas.

3.1 Lisp Macros

Unlike C++ macros, which perform simple word-based replacement ignoring the syntax, Lisp macros allow the programmers to manipulate program text as data; A program is manipulated through an ordinary data structure. We call this data structure a *meta representation* of the program. In Lisp macros, the meta representation of a program is an abstract syntax tree. The program text is represented in the form of the tree whose leaf nodes are the lexical tokens of the program text.

Lisp macros enable programmers to implement some kinds of abstractions that Lisp functions cannot implement. Those abstractions are called

special forms in the Lisp terminology. For example, the following macro implements a special form `rbegin` that sequentially evaluates expressions from right to left (in the reverse order of `begin`) and returns the resulting value of the leftmost expression:

```
(define rbegin
  (macro exprs
    `(begin ,@(reverse exprs))))
```

A backquote (`) and `,@` are convenient notation for constructing a tree structure. If the readers are not familiar to this notation, see Appendix A).

This macro is used as follows:

```
(rbegin (+ 3 4) (list 1 2) (* 5 8))
```

This program is processed by the macro function `rbegin` before being executed. The macro function is an ordinary Lisp function except that it receives and returns program text. The macro argument `exprs` is bound to a list `((+ 3 4) (list 1 2) (* 5 8))`. Then the macro function returns this program:

```
(begin (* 5 8) (list 1 2) (+ 3 4))
```

This is substituted for the original program `(rbegin (+ ...)` before the execution. The resulting value of the program is that of this substituted program, that is, 7.

Note that a Lisp function `reverse` is called during the macro expansion. It reverses the order of the list that `exprs` is bound to. Since the processed program text is the first-class data represented in the tree structure,[1] this Lisp function can process it as it processes ordinary Lisp data. This capability makes Lisp macros different from other simple macros like C++ macros. Indeed, C++ macros cannot implement the special form `rbegin` since the given program text is not the first-class data. They can only perform the limited operations on the program text. Only word-by-word replacement and concatenation are allowed.

Applicability to our problem

Lisp macros give a partial solution of the problem we discussed in the previous chapter. They allow programmers to process a piece of code following a macro name. An advantage of Lisp macros is that programmers can *program* how the piece of code is expanded. This provides the ability to generate more specialized code than what the template mechanism can do by simple

[1] The first-class data are the data that the program can handle as the object of the computation. For example, numbers, symbols, lists, and vectors, are the first-class data in Lisp.

replacement of type names. On the other hand, Lisp macros cannot handle context-sensitivity or non-locality and thus just porting the Lisp macro system to C++ does not solve our problem.

3.2 3-Lisp

Lisp macros provide the meta representation of programs so that they can implement some kinds of abstractions that Lisp functions cannot implement. Although the meta representation in Lisp macros is only program text, 3-Lisp [54, 53] provides not only program text but also other information as the meta representation. This feature of 3-Lisp enhances the variety of abstractions that programmers can implement.

Meta representation

The meta representation of programs in 3-Lisp consists of program text, the current environment, and the current continuation. Adding the latter two makes it possible to implement abstractions that Lisp macros cannot handle. Suppose that we implement a special form `defined?`, which returns true if the given variable is defined in the current environment:

```
> (define x 1)
x
> (defined? x)
#t
> (defined? y)
#f
```

The special form `defined?` returns `#t` (true) for the variable `x`, but `#f` (false) for the variable `y`.

Because Lisp macros cannot examine the current environment, they cannot deal with this special form. However, 3-Lisp can do. See the following program written in 3-Lisp:[2]

```
(define defined?
  (meta (expr env cont)
    (if (is-bound? env (car expr))
        (cont #t)
        (cont #f))))
```

The special form `defined?` is implemented by a meta function (originally called *lambda reflect* in 3-Lisp). Unlike macro functions, which receive only program text, the meta function receives the current environment `env` and the current continuatin `cont` as well as the program text `expr`. The current environment represents the bindings between symbol names and values,

[2] To make it easy to read, we modify the syntax of 3-Lisp. We believe that this modification does not affect the essential idea of 3-Lisp but rather helps articulate it.

while the current continuation represents the control flow after this special form finishes. Note that the meta function may access and change the environment and the continuation since they are the first-class data within the meta function. In fact, to implement `defined?`, the meta function calls a built-in function `is-bound?` and looks up a symbol name obtained by `(car expr)` in the received environment `env`. Then it calls the received continuation `cont` with `#t` or `#f`, which is the resulting value of the special form.[3]

Base level and meta level

Meta functions are not extended macro functions that receive more arguments. There are significant difference between macro functions and meta functions. Meta functions run at runtime because they manipulate the current environment and the current continuation, which are only available at runtime. Hence a meta function directly interpret the received program text. The value returned by the meta function is the result of the interpretation. The meta function also may cause side effects on the environment as the result of the interpretation. On the other hand, macro functions may run at either runtime or compile time since they deal with only static information of the program, that is, program text. They are functions that receive program text and transform it, but they do not directly interpret it. The value returned by a macro function is the transformed text.

In 3-Lisp, both ordinary functions and meta functions run together at runtime. To distinguish the two kinds of functions, 3-Lisp has two execution levels, which are the base level and the meta level. The two execution levels are identical except that the objects of the computation at the meta level is the interpretation of the base-level program. The result of the computation at the meta level *reflects* on the computation at the base level. This relation between the meta level and the base level is called *causal connectivity*. For example, meta functions may change the environment at the base level in order to define a new symbol name, or to change the value that a symbol name is bound to.

Meta functions can use a built-in meta function if they interpret the received program text in the default way. For example, the program below is another implementation of the special form `rbegin` with a meta function:

```
(define rbegin
  (meta (expr env cont)
    (let loop ((rest expr)
               (c    cont))
      (if (null? rest)
          (c '???)
          (loop (cdr rest)
                (lambda (r)
                  (eval-expr env        ; environment
                             (car rest) ; sub-expression
                             c)))       ; continuation
```

[3] In reality, the base-level value has different representation at the meta level. For example, `#t` (a boolean) becomes `'#t` (a symbol) at the meta level. Therefore, the meta function must convert `#t` into the meta representation before passing it to the continuation `cont`.

Note that a built-in meta function `eval-expr` (`normalize` in the 3-Lisp terminology) is called to interpret each sub-expression of `rbegin`. According to the given sub-expression, `eval-expr` causes side effects on the environment and calls the continuation with the resulting value of the sub-expression.

Self modification

In the examples shown so far, we have introduced new keywords such as `rbegin` and `defined?`, and defined their meanings to implement new abstractions. Some kinds of abstractions, however, require altering the meanings of the existing keywords or syntax. In other words, they need to modify the default behavior of the language rather than to extend it. A feature of this *self modification* is that the effects of the modification are applied to programs even though the programs are not edited. Editing them or inserting new keywords are not necessary.

Some followers of 3-Lisp, such as Black [2], enables the self modification. They provide many built-in meta functions, which carrry out a primitive base-level operation, and allow programmers to redefine them to change the default behavior of the operations. For example, they may alter the behavior of the language when reading a variable. By default, reading an undefined variable causes an error. We alter this behavior so that `'undef` is returned if the variable is undefined:

```
(define change-eval-read
  (meta (expr env cont)
    (set! eval-read
          (lambda (expr env cont)
            (let ((var (car expr)))
              (if (is-bound? env var)
                  (cont (lookup env var))
                  (cont 'undef)))))
    (cont #t)))
```

If the meta function `change-eval-read` is executed, it calls `set!` to substitute a new meta function for the built-in meta function `eval-read`. Note that `set!` has to be invoked at the meta level. Otherwise, `set!` would replace a base-level function `eval-read`, if any, with the new one. The substituted meta function first checks whether the given variable is defined or not in the current environment. If not, it uses `'undef` for the value of the variable:

```
> x
ERROR: undefined symbol
> (change-eval-read)
#t
> x
UNDEF
```

This sort of self modification is difficult for Lisp macros. Some implementations of the Lisp macro system allow a macro function to override an existing keyword. For example, programmers may define a macro function named `if` to alter the behavior of the `if` special form. However, Lisp macros cannot handle such modification that we showed in the `change-eval-read` example because the expression for reading a variable is not preceded by any keyword. To process an expression by a Lisp macro, programmers have to explicitly place the macro name in front of the processed expression.

To use a Lisp macro and get the same result that `change-eval-read` provides, programmers have to use a programming convention when reading a variable. For example, they may have to write `(read-variable x)` instead of just writing `x`, whenever reading a variable `x`. This programming convention makes it possible for a macro `read-variable` to process the expression `x`. Another approach is to surround the whole program by a macro call. This technique is called the code walker. The macro function receives the whole program as the argument, looks up expressions for reading variables, and translates the expressions to alter the behavior. Writing the code walker is not difficult in Lisp because the grammar is simple, but it is difficult in other languages such as C++.

Reflective languages

3-Lisp is one of the earliest languages that can handle richer meta representation of the programs than Lisp macros.[4] 3-Lisp has been followed by many languages, and this family of languages are often called *reflective languages* or languages with a meta architecture. For example, Brown [60], Blond [18], and Black [2] are Lisp-based successors of 3-Lisp. These successors has been developed to study the semantics and the implementation of the infinite tower of the execution levels, that is, the base level, the meta level, the meta-meta level, and so on. This was also one of the main issues of the study of 3-Lisp.

The meta architecture developed by 3-Lisp has been also applied to object-oriented languages. Early representatives of such languages are CLOS [56, 36] and 3-KRS [38]. The two languages have been developed under different design strategies. Since we discuss CLOS in the next section, we introduce only 3-KRS here.

[4] In Artificial Intelligence, a similar idea was proposed earlier than 3-Lisp.[3, 25, 62]

3-KRS is designed mainly for customizing the default behavior of the language on demands, rather than implementing new abstractions on top of the language. In 3-KRS, each base-level entity such as objects and messages is associated with a special object called *metaobject*. The metaobject is the meta representation of the base-level entity. Calling a method for the metaobject, programmers can obtain the meta information of the entity. For example, they can inspect what methods an object has through the metaobject for the object. Also, the metaobject provide similar capability of a built-in meta function in 3-Lisp. For example, the metaobject for an object has a method for invoking a method on the object.

As in 3-Lisp, programmers can alter the default behavior of 3-KRS. Instead of redefining built-in meta functions, the programmers define a new metaobject to implement the new behavior. Suppose that they alter the default behavior of the object creation so that the history of object creation is recorded. With this language customization, programmers may write something like this:

```
> *history*
()
> (defclass point ()
    (variable x y) :meta recorded-object)
POINT
> (defclass rect ()
    (variable top bottom left right) :meta recorded-object)
RECT
> (define p (make-instance point))
P
> (define q (make-instance rect :meta recorded-object))
Q
> *history*
(RECT POINT)
```

If a class annotated with ":meta recorded-object" is instantiated, the class name of the instance is added to the list indicated by `*history*`. For the example above, since a `point` object and a `rect` object are created, the class `point` and the class `rect` are added to the list.

To implement this language customization, the programmer first defines a new class `recorded-object` for metaobjects:[5]

```
(defclass recorded-object (metaobject-for-object)
  (defmethod create-object (expr env)
    (let ((class-name (car expr))
          (h          (env-ref env '*history*)))
      (env-set! env '*history* (cons class-name h))
      (<- super create-object expr env))))
```

The class `recorded-object` inherits from the default class `metaobject-for-object` and overrides a method `create-object` to maintain `*history*`.

[5] Again, we use an altered version of 3-KRS for clarifying our argument.

The method `create-object` is invoked when an object is created. The new `create-object` first adds the class name to the list indicated by the base-level variable `*history*`. Then it calls the overridden method of the super class `metaobject-for-object`, which creates a new object in the default way. Note that `create-object` has to call built-in meta functions `env-ref` and `env-set!` to access the base-level variable `*history*`. It cannot directly access this variable since `create-object` is at the meta level. Any base-level entity must be dealt with through built-in meta functions.

Other object-oriented reflective languages include ABCL/R [61, 41, 40], Ferber's language [21], RbCl [32], The MIP for C++ [9], OpenC++ version 1 [13], AL-1/D [46, 45], CodA [42], a reflective version of BETA [6], and Iguana [29, 30]. They have explored various applications of the meta architecture. ABCL/R is a parallel language, and it allows programmers to customize the default scheduling policy. OpenC++ version 1 enables programmers to implement language extensions such as distribution, persistence [57], and fault-tolerance [20], within the confines of the language. AL-1/D's application is similar to ABCL/R. It allows programmers to customize the policy of object migration. Coda employs the meta architecture to run the same Smalltalk program on different platforms. For example, if the programmer ports a program written for a single processor machine to run on a multi-processor machine, she has only to define new metaobjects. Since the platform-dependent code is separated into the metaobjects, the programmer does not need to edit the base-level program.

Applicability to our problem

Reflective languages represented by 3-Lisp show us how meta representation of a program should be exposed to the programmer. Especially, object-oriented reflective languages such as 3-KRS shows that metaobjects can be good abstraction to deal with complexity of the meta representation.

However, reflection cannot be a solution of our problem since programmers describe interpretation of a base-level program to define a new mechanism or change an existing mechanism. Although programmers can describe interpretation that causes the same effects that we want, describing source-code processing is more intuitive and straightforward as for our motivating applications. Also, runtime penalties implied by the interpretation is another problem of reflection.

3.3 The CLOS MOP

We claim that the reflective languages introduced in the previous section, such as 3-Lisp and 3-KRS, should be called non-metacircular, or weakly-metacircular, languages. They are metacircular in a certain sense because

base-level and meta-level programs are written in the same syntax, but the two "languages" for the base-level and meta-level programs are not identical.

This section discusses (truly) metacircular reflective languages represented by the CLOS MOP. Unlike 3-Lisp or 3-KRS, its base level and meta level programs are written in an identical language. This means that the customization by the meta-level program affects not only the base-level language but also the meta-level language in which the meta-level program is written. This feature gives some benefits to reflective languages.

Metacircularity

Metaobject protocols (MOPs) are another name to indicate a meta architecture such as 3-Lisp's one. Particularly, MOPs mean programming interfaces for customizing the language. The word "metaobject protocol" has been first used in CLOS (Common Lisp Object System) [56]. The CLOS MOP [36] enables programmers to incrementally customize CLOS in CLOS itself with a meta architecture.

A unique feature of the CLOS MOP is that the system is metacircular (Figure 3.1). In 3-Lisp, meta functions and base-level functions are written in the same language, but the two "languages" for the meta and base-level functions may not be identical; they may be two distinct instances of the same language. At beginning, the two languages are identical and are thus meta-circular in some sense. But once the base language is customized, the meta language is left unchanged and the two languages are therefore different.

This fact would be clear if we reason about the object of language customization; the language customized by a meta-level program is just the language for a base-level program. For example, a new special form defined by a meta function is only available for base-level functions but not available for other meta functions. The customization by meta functions reflects only on the language for base-level functions, but it does not circularly reflect on the language for the meta functions.

The reason for this non-metacircularity would be to avoid apparent infinite regression caused by a circular definition. If 3-Lisp is naively made metacircular, programmers would easily define a meta function with using the special form defined by the meta function itself and the special form would cause infinite regression. But this problem can be avoided with keeping metacircularity if another scope control mechanism is introduced.

To keep metacircularity while letting programmers avoid a circular definition, the CLOS MOP uses a class system for controlling the scope of language customization. Hence both base-level programs and meta-level programs are written in the same instance of the language, and language customization by a meta-level program is also applied to the language in that the meta-level program is written. The relation is circular between the

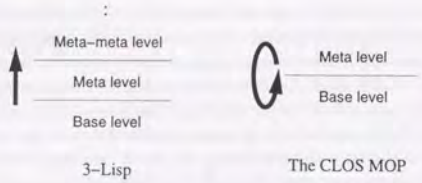

Figure 3.1: The CLOS MOP is metacircular

semantics of the customized language and the meta-level program customizing the language.

Despite of the metacircularity, CLOS programmers can avoid a wrong circular definition.[6] In the CLOS MOP, customization is specified on a class basis; it is applied only to particular classes and their instances. Thus, if programmers carefully distinguish classes, they can avoid a circular definition. Suppose that a meta-level program alters the behavior of a class `Point`. Since the CLOS MOP is metacircular, the programmer may use the customized class `Point` in the meta-level program, but she can still avoid a circular definition unless she explicitly implements the customization with `Point` to be a circular definition.

The origin of the metacircular architecture would be found in Smalltalk-80 [26]. In fact, Foote reported the use of the metacircularity in Smalltalk-80 in his paper [22]. Also, CommonLoops [4] should be noted as an early metacircular language. Furthermore, the metacircular architecture is found in ObjVlisp [17], Classtalk [8], EuLisp [7] and STklos [24].

Direct access to the base level

The CLOS MOP provides the capability for customizing the behavior and implementation of the object system of Common Lisp (i.e., CLOS). For example, programmers may define a new metaobject for classes to alter the rule of multiple inheritance. This customization makes it easy that CLOS runs programs written in other Lisp-based languages such as Loops [5] and Flavors [10].

Despite the metacircularity, the way of customization with the CLOS MOP is quite similar to the way of customization in non-metacircular languages such as 3-KRS. In the previous section, we showed the 3-KRS implementation of the language customization for recording object creation. In

[6] The solution by the CLOS MOP is not a complete solution. We revisit this issue in Chapter 5.

the CLOS MOP, this customization is implemented by this program:[7]

```
(defclass recorded-class (metaobject-for-class)
  (defmethod create-object (init-args)
    (let ((class-name (<- self name)))
      (set! *history* (cons class-name *history*))
      (<- super create-object init-args))))
```

Since the CLOS MOP does not provide the metaobject for an object, the program defines a new metaclass, which means a new class for class metaobjects. It inherits from the default class `metaobject-for-class` and overrides a method `create-object`. When an object is created, the new `create-object` first calls a method `name` for the metaobject (`self`) to obtain the class name. Then it updates the history of object creation and calls the overridden method for the super class, which creates an object in the default means.

Because of metacircularity, there is no explicit (syntactical) distinction between base-level programs and meta-level programs in the CLOS MOP. The two kinds of programs are written in the same language and run under the same runtime environment. They may even coexist in a single function or method. Therefore, metaobjects in the CLOS MOP can directly access the base-level data. The metaobjects can use the base-level primitives to access the base-level data as the objects can do. There is no difference in primitives that the metaobjects and the objects can use. For example, the method `create-object` directly reads and writes the variable `*history*` with base-level primitives like `set!`. Recall that `create-object` in 3-KRS has to access through built-in meta functions `env-ref` and `env-set!` since the base-level objects and the metaobjects are run at distinct execution levels.

Ease of learning

Base and meta levels in non-metacircular systems are written in different languages. This separation makes access across levels be complex and tend to be inefficient. On the other hand, the metacircularity of the CLOS MOP avoids gratuitous differences between levels, while still being effective, and gives a few advantages.

First, the meta architecture employing metacircularity is easy to learn. Since programmers can use base-level primitives to access base-level data from the meta level, they do not have to learn built-in meta functions unless they need to access meta-level data, which there is no base-level primitive to access.

[7] Again, we use an altered syntax for emphasizing difference from the 3-KRS implementation.

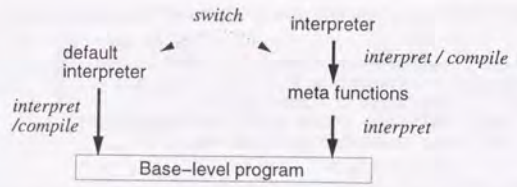

Figure 3.2: An improved implementation of 3-Lisp

For example, a metaobject in the CLOS MOP can call a method for an object in the same syntax that an object calls it at the base level. The metaobject uses a built-in meta function only when it needs to access a meta representation of the program, for example, the class name of an object. Recall that `create-object` of `recorded-class` executes `(<- self name)` to obtain the class name. On the other hand, a metaobject in 3-KRS has to call a built-in meta function even when it calls a method for an object. In 3-KRS, any object must be indirectly manipulated through the metaobject associated with that object. To call a method for the object, an appropriate built-in meta function for the metaobject has to be called.

Ease of implementing efficiently

Another advantage of metacircularity is execution performance. Metacircular systems make it intuitive and straightforward for implementors to develop an efficient interpreter and compiler. To do the same thing in non-metacircular systems like 3-Lisp, they need sophisticated implementation techniques or they need to force programmers to use advanced programming techniques.

If naively implemented, 3-Lisp is extremely slow. The typical implementation of 3-Lisp uses two interpreters for keeping two distinct execution levels. The first interpreter executes the second interpreter, while the second interpreter executes base-level functions. In this implementation, meta functions are regarded as part of the program of the second interpreter. They are executed by the first interpreter and execute base-level functions. This double interpretation maintains the causal connectivity between the base level and the meta level, but it significantly decreases the execution speed of the base-level program even though no meta functions are used.

Improving the execution performance is relatively easy if the base-level program does not use meta functions. We can prepare the third interpreter, named the default interpreter, and switch to it while the base-level program runs without meta functions (Figure 3.2). Note that the first interpreter and

the default interpreter are not identical; at least, they are distinct instances of an interpreter and they maintain different runtime environments. Because the default interpreter does not handle meta functions, it can achieve as good efficiency as an optimized interpreter for ordinary Lisp. Or, we may use a compiler for better efficiency instead of an interpreter.

To use the default interpreter from a meta function, programmers need to call a built-in meta function like `eval-expr` and explicitly switch to it. Recall the implementation of `create-object` in 3-KRS. It can be rewritten to explicitly switch to the default interpreter:

```
(defmethod create-object (expr env)
  (let* ((class-name (car expr))
         (expr2 '(set! *history*
                       (cons ,class-name *history*))))
    (eval-expr env expr2)
    (<- super create-object expr env)))
```

The built-in meta function `eval-expr` executes the expression `expr2` with the environment `env` by the default interpreter, so that the expression is executed faster. For example, since the expression accesses the base-level variable `*history*` twice, the default interpreter may memoize the memory location of the variable and reuse it for the second access. However, the optimization that the default interpreter can do for `eval-expr` is limited to runtime optimization since `eval-expr` receives an expression and an environment at runtime. Even if an compiler is used instead of the interpreter, no static optimization is applicable.

On the other hand, metacircular systems naturally give more freedom to the default interpreter or compiler. For example, in the CLOS MOP, there is no explicit distinction between the base level and the meta level, so the default interpreter can execute both base-level and meta-level programs (Figure 3.3). This means that programs implicitly switch direct interpretation by the default interpreter and double interpretation via metaobjects. Recall the implementation of `create-object` in the CLOS MOP:

```
(defmethod create-object (init-args)
  (let ((class-name (<- self name)))
    (set! *history* (cons class-name *history*))
    (<- super create-object init-args)))
```

This method directly accesses a base-level variable `*history*` with the base-level primitive `set!`. Since the expression `(set! ...)` is written in the syntax for the base-level program, it can be directly executed by the default interpreter, and hence the system implicitly switches to direct execution by the default interpreter.

This implicit switching gives a lot of room for optimization especially to a compiler. Unlike the way by using `eval-expr`, an executed expression

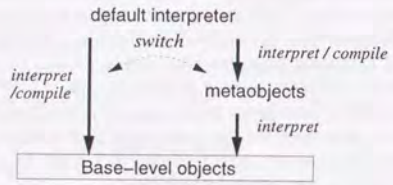

Figure 3.3: An metacircular implementation

is statically given to the default interpreter and hence various optimization techniques using static information are naturally applicable.

To get equivalent effects in non-metacircular systems, a sophisticated optimizing compiler or complex programming techniques are needed. For example, an optimizing compiler using a technique called partial evaluation [19, 23] or inlining will compile the following program in 3-KRS:

```
(defmethod create-object (expr env)
  (let* ((class-name (car expr))
         (expr2 '(set! *history*
                       (cons ,class-name *history*))))
    (eval-expr env expr2)
    (<- super create-object expr env)))
```

into as efficient code as the equivalent program in the CLOS MOP. The compiler will statically determine the values of `env` and `expr2` as much as possible, and specialize `eval-expr` to be more efficient. This approach has been studied by a few researchers including the author [52, 14, 39], but implementing this technique is very difficult in practical languages such as C++; for example, no compiler using partial evaluation has been developed for C++ yet.

Another approach is to provide more built-in meta functions for programmers to be able to optimize a meta function by hand:

```
(defmethod create-object (expr env)
  (let ((compiled-code (lookup-hash expr)))
    (if (null? compiled-code)
        (let* ((class-name (car expr))
               (expr2 '(begin
                         (set! *history*
                               (cons ,class-name *history*))
                         ,expr)))
          (set! compiled-code (compile-expr expr2))
          (record-hash expr compiled-code)))
    (execute env compiled-code)))
```


The underlined functions are built-in meta functions. The above implementation explicitly compiles an expression and memoizes it when it is first executed, and reuses the compiled code from the second time. Although this implementation will run a bit faster, the built-in meta functions for optimization make the language complicated and difficult to use. Programmers have to learn how the built-in meta functions help for optimization.

When metaobjects run

Normally, metaobjects in metacircular systems run at runtime as in non-metacircular systems. Metacircularity has nothing to do with when metaobjects run at runtime or compile time. However, because running metaobject at runtime impairs execution performance, the actual CLOS MOP employs a technique called *currying*[8] so that the metaobjects less frequently run at runtime. This technique is based on the observation that some of computation by metaobjects can be statically performed, or, once it is done, the result can be memoized and it does not need to be performed again. So the currying technique explicitly splits the protocol into one for computation that has to be done at runtime and the rest, and let the underlying interpreter improve performance by avoiding redundant execution of that computation.

For example, if the currying technique is not adopted, a method `call-method` for class metaobjects is implemented as something like this (Note: this is pseudo code):

```
(defmethod call-method (object method-name args)
  (let ((method (<- self lookup method-name)))
    (apply method (cons object args))))

(defmethod lookup (method-name)
  (let ((entry (assq method-name
                     (<- self direct-methods))))
    (if entry
        (cdr entry)
        (let loop ((supers (<- self super-classes)))
          (if (null? supers)
              (error "invalid method name")
              (or (<- (car supers) lookup method-name)
                  (loop (cdr supers))))))))
```

The method `call-method` is a built-in meta function for calling the method specified by `method-name` for the `object`. The implementation is divided into two parts. It first calls a helper method `lookup`, which finds the function body of the targeted method. Then `call-method` actually invokes the found

[8] This technique should not be called the memoization technique. It is a technique for protocol designers, who perform currying on documented functions so that protocol implementors can perform memoization.

method. `lookup` first checks methods directly supplied by the class, and, if not found, it searches super classes.

This implementation is not efficient because the targeted method is looked up in the class hierarchy for every call. It should be looked up only once when it is first called, and the found method should be memoized and reused for all subsequent calls to the method.

To avoid this inefficiency, the actual protocol has been designed with the currying technique. In the actual design, the computation by `call-method` shown above is split into two parts. The redesigned `call-method` receives only `method-name`, looks up the targeted method, and returns a function:

```
(defmethod call-method (method-name)
  (let ((method (<- self lookup method-name)))
    (lambda (object args)
      (apply method (cons object args)))))
```

If called, the function returned by `call-method` receives `object` and `args` and invokes the targeted method with them. The targeted method is not invoked until the returned function is called. Note that the targeted method is looked up only once when `call-method` runs. It is never looked up when the returned function runs.

The CLOS interpreter and compiler can employ this curried protocol to improve execution performance. It should call `call-method` in advance at load time or at compile time, and memoize the returned function with the method name. Then, if the method is actually called at runtime, it can directly call the memoized function for invoking the method. No look up is needed at runtime. The targeted method is looked up only once at load time or at compile time.

Applicability to our problem

The CLOS MOP shows that there is another design of the meta system of a reflective language. Because of its metacircularity, a meta-level program is not an interpreter of the base-level program but can be regarded as a collection of base-level code substituted for parts of the original base-level program. Recall that a meta-level program describes base-level behavior with the base-level primitives instead of built-in meta functions.

This fact means that metacircular reflection can be the basis of a solution of our problem. In the previous chapter, we presented that C++ needs a mechanism for processing source code with context sensitivity and non locality. Metacircular reflection allows programmers to specify source-code substitution with context sensitivity and non locality.

However, we cannot use metacircular reflection represented by the CLOS MOP *as is* to solve our problem. It is in principle a runtime system, which may involve overheads, and is difficult to handle optimization such as the

Table 3.1: Lisp Macros, 3-Lisp, and the CLOS MOP

	Lisp Macros	*3-Lisp*	*CLOS MOP*
Meta representation	△	○	○
Self modification	×	○	○
Metacircular	○	×	○
When running	CT or RT	RT	(CT and) RT

RT: runtime, CT: compile time

example of the vector library in the previous chapter. It is not suitable for dealing with adjacent but independent operations at a time.

3.4 Summary

This chapter illustrated currently known mechanisms for processing a program. We showed the tree representative mechanisms, Lisp macros, 3-Lisp, and the CLOS MOP. As an object-oriented version of 3-Lisp, we also showed 3-KRS.

Table 3.1 summarizes the features of the three mechanisms. All of them provide meta representations of the base-level programs for the programmers. The meta representations enable the programmers to process the programs as computable data. As the meta representations, Lisp macros provide program text in the form of the abstract syntax tree. 3-Lisp provides not only program text but also the current environment and the current continuation. The CLOS MOP provides classes, generic functions, methods, and so on.

Although providing meta representations allows to implement a new kind of abstraction that are not available within the confines of the original language, some kinds of abstractions also need to alter the default behavior of the language. This self modification of the language is allowed only by 3-Lisp and the CLOS MOP. Lisp macros cannot change the behavior of the language when reading a variable, for example. 3-Lisp and the CLOS MOP, therefore, enable a larger number of kinds of abstractions than Lisp macros.

Metacircularity is a good property because metacircular systems are easy to learn and easy to implement efficiently. Only Lisp macros and the CLOS MOP have this property. 3-Lisp is not metacircular; although the base level and the meta level uses the same language, the customization by meta functions is not applied to the language for the meta level. The customization is applied only to the language for the base level.

Furthermore, macro functions may run at either runtime or compile time,

whereas meta functions in 3-Lisp and metaobjects in the CLOS MOP run at runtime. This means that Lisp macros have an advantage in terms of execution performance since macro functions run at compile time. 3-Lisp has to process a program at runtime even though the processing can be done at compile time. To avoid the performance problem of 3-Lisp, the CLOS MOP employs the currying technique and performs most of meta computation at the load time or at compile time.

Through this chapter, we have discussed that a meta architecture with metacircularity, such as the CLOS MOP, has great benefits for processing a program. But this architecture has a drawback with respect to execution performance because metaobjects normally run at runtime unless an elaborate technique like currying is used. Lisp macros have an advantage for the performance issue; macro functions can run at compile time and they can involve no performance penalties at runtime. This feature of Lisp macros is significant to develop a mechanism for processing a program in C++ since C++ programmers are particular about execution performance.

In the next chapter, we propose a new C++ mechanism with both advantages of the CLOS MOP and Lisp macros. It enables source-code processing in a context-sensitive and non-local way so that programmers can develop better libraries in C++.

Chapter 4

The OpenC++ MOP

This chapter presents a new C++ mechanism for processing a program. The proposed mechanism enables context-sensitive and non-local processing, which is not supported by existing C++ mechanisms, such as the inheritance mechanism and the template mechanism. Because of those two features, this mechanism makes it possible to include some kinds of useful abstractions in a library and implement other abstractions efficiently. In regular C++, those abstractions are impossible to include in a library or, otherwise, they are difficult to implement efficiently.

The proposed mechanism is called the OpenC++ MOP (Metaobject Protocol). OpenC++ is an enhanced version of the C++ language for this mechanism.[1] The OpenC++ MOP has been developed by a synthesis of ideas of the techniques illustrated in the previous chapter. Especially, we took the basic protocol structure from the CLOS MOP [36], and we took the basic architecture from Lisp macros. The OpenC++ MOP is also influenced by Intrigue [37], Anibus [50, 51], and MPC++ [33, 34].

4.1 Overview

The OpenC++ compiler is fed with two kinds of code. One is ordinary source code and the other is meta code that specifies how the source code is processed. Both of them are written in OpenC++. The compiler first runs the C++ preprocessor, and then performs source-to-source translation from OpenC++ to regular C++. This translation is specified by the meta code. The translated code is passed to the back-end C++ compiler and processed into executable code.

[1] To distinguish OpenC++ version 1, this language is called OpenC++ version 2.

Introductory Example

The OpenC++ MOP is a protocol that the meta code uses for specifying source-code processing. The protocol structure of the OpenC++ MOP is based on that of the CLOS MOP. Programmers define a new class metaobject to specify a new kind of source-code processing. For example, we show the implementation of the language extension that records object creation. This is the example that we repeatedly used in the previous chapter:

```
class RecordedClass : public Class {
public:
  Ptree* TranslateNew(Environment* env, Ptree* header,
                      Ptree* new_op, Ptree* placement,
                      Ptree* type_name, Ptree* arglist)
  {
    return Ptree::Make("(history[n++]=\"%p\",%p)",
                       type_name,
                       Class::TranslateNew(env, header, new_op,
                                           placement, type_name,
                                           arglist));
  }
};
```

We define a new metaclass, which is a class for class metaobjects. The new metaclass `RecordedClass` corresponds to `recorded-class` that the previous chapter showed while mentioning the CLOS MOP. It inherits from the default metaclass `Class` and overrides a member function[2] `TranslateNew()` as the class `recorded-class` inherits from `metaobject-for-class` and overrides `create-object`.

Although `create-object` in the CLOS MOP directly interprets the program and executes the object creation, `TranslateNew()` in the OpenC++ MOP just translates the program at the source-code level as Lisp macros do. It receives program text and returns the resulting text of the translation.

To use this language extension, programmers write something like this:

```
metaclass Point : RecordedClass;
class Point {
public:
    int x, y;
};

void f(){
    . . .
    Point* p = new Point;
    . . .
}
```

[2] A member function means a method in the C++ terminology. Similarly, in C++, an instance variable is called a data member, and a super class is called a base class.

The first line beginning with `metaclass` is a metaclass declaration. It declares that the class metaobject for `Point` is an instance of `RecordedClass`. Thereby, the program above is translated by the class metaobject into this program:

```
class Point {
public:
    int x, y;
};

void f(){
    . . .
    Point* p = (history[n++]="Point",new Point);
    . . .
}
```

Note that the metaclass declaration is eliminated and the `new` expression "`new Point`" is replaced with "`(history[n++]=...)`". If the function `f()` is executed, a character string "`Point`" is stored in an array `history`. The variable `n` specifies the number of the recorded class names.

What Class Metaobjects do

The translation shown above is performed by `TranslateNew()`. It receives the program text for the `new` expression and the environment, and returns the translated `new` expression. The environment represents the bindings between names and their static types. Unlike the environment in 3-Lisp, it does not represent the dynamic bindings between names and their runtime values. The received program text is represented in the form of the parse tree as in Lisp macros and 3-Lisp. `Ptree` is the type for the parse tree. For convenience, `TranslateNew()` does not receive the `new` expression as a single tree. The tree is divided into several subtrees before passed to `TranslateNew()`. For example, the fifth argument `type_name` is bound to the type-name field of the `new` expression, that is, `Point`.

`TranslateNew()` constructs a new parse tree that is substituted for the original `new` expression. To do this, `TranslateNew()` calls a built-in function `Ptree::Make()`. Here we show the function body of `TranslateNew()` again:

```
return Ptree::Make("(history[n++]=\"%p\",%p)",
                   type_name,
                   Class::TranslateNew(env, header, new_op,
                                       placement, type_name,
                                       arglist));
```

The expression `Class::TranslateNew(env, ...)` calls the overridden member function of the base class `Class`. Since `Class` is the default metaclass, this member function call returns the `new` expression without any change. The function `Ptree::Make()` constructs a parse tree according to the format

given as the first argument. It substitutes given subtrees for the occurrences of %p in the format. For example, the first occurrence of %p surrounded by double quotes is replaced with the subtree indicated by type_name.

A class metaobjects handles all kinds of expressions involved with the class. They can handle class declaration, the new expression, member function calls, and even data member accesses. The default metaclass Class has member functions for translating each kind of expression. Although these member functions of Class do not transform the received program text, programmers can define a new metaclass to override them and customize the translation. Note that a class metaobject handles only the expressions involved with the class. Hence programmers can restrict the customized translation to only some classes. Even though a new metaclass is defined, expressions are not translated unless the metaclass is specified for the class that the expressions are involved with.

4.2 Context-sensitive and Non-local

The OpenC++ MOP enables context-sensitive and non-local processing of programs, which the inheritance mechanism or the template mechanism cannot do. This feature of the OpenC++ MOP makes it possible for programmers to write libraries that they cannot do in regular C++.

Context Sensitive

The OpenC++ MOP provides rich meta representation of programs so that programmers can refer to various contextual information of the programs during the processing. For example, the programmers can refer to the program text represented in a parse tree, the static type environment, and class definitions. The OpenC++ MOP provides different metaobjects for each kind of information. The programmers use these metaobjects and determine how to process the programs.

We below present brief overviews of all the kinds of metaobjects: Ptree, Environment, TypeInfo, and Class. The detailed specifications are shown in Appendix B. These metaobjects represent program text, static types, type definitions, class definitions, respectively. They cover information needed to determine the semantics of a given code fragment.

• Ptree metaobject

The Ptree metaobjects represent the parse tree of a program. The parse tree is implemented by a linked list of lexical tokens. For example, this program:

```
int a = b + c * 2;
```


is represented by a Ptree metaobject:

```
((int) (a = (b + (c * 2))) ;)
```

Here, () denotes a linked list. We denote a parse tree with the notation that Lisp uses for the S expression. Note that operators such as = and + make sublists.

The OpenC++ provides many functions for manipulating a parse tree. Most of them were taken from Lisp. For example, to obtain the second sublist of the list that a variable expr is bound to, the programmer writes:

```
Ptree::Second(expr)
```

The function Second() is a static member function[3] of the class Ptree. It returns the Ptree metaobject that represents the second sublist.

Moreover, since the grammar of C++ is relatively complex, the Ptree metaobjects provide a member function WhatIs() for examining the kind of the syntax represented by the parse tree. WhatIs() returns a unique constant according to the kind of the syntax. If the parse tree represents a declaration, WhatIs() returns PtreeDeclarationId; if the tree represents a class name, WhatIs() returns LeafClassNameId.

As we already saw, Ptree metaobjects can be constructed by calling Ptree::Make(). This static member function constructs a Ptree metaobject according to the given format. All occurrences of %c (character), %d (integer), %s (character string), and %p (Ptree) in the format are replaced with the arguments following the format. For example, suppose that array_name is "xpos" and offset is 3. This function call:

```
Ptree::Make("%s[%d]", array_name, offset)
```

constructs a Ptree metaobject that represents:

```
xpos[3]
```

Unfortunately, the current implementation of Ptree::Make() does not construct a fully-capable Ptree metaobject. Since the C++ grammar is context sensitive, Ptree::Make() cannot correctly parse the constructed metaobject without syntactic context and hence WhatIs() does not work for it. Except this limitation, however, programmers can use the Ptree metaobject returned by Ptree::Make() at any place in a meta-level program.

The function Ptree::Make() makes it significantly easy to write a meta-level program. It is a conversion function from standard C++ syntax given as a C++ string to a Ptree metaobject. Thus programmers can construct a Ptree metaobject with standard C++ syntax, which is more intuitive and easy to handle than bare new operators.

[3] A static member function is a sort of class method in C++.

The OpenC++ MOP also provides a function for pattern matching. The static member function Ptree::Match() compares the given pattern and the given Ptree object. If they match, it returns TRUE and binds the given variables to the appropriate sublists. See the following sample program:

```
if(Ptree::Match(expr, "[%? + %?]", &lexpr, &rexpr))
    cout << "this is addition.\n";
else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
    cout << "this is subtraction.\n";
else
    cout << "unknown\n";
```

The pattern [%? + %?] matches a Ptree metaobject if the length of the linked list is three and the second element is +. If expr matches the pattern, lexpr gets bound to the first element of expr, and rexpr gets bound to the third element. Note that the type of lexpr and rexpr is Ptree.

• Environment metaobject

The Environment metaobjects represent bindings between names and their static types. Programmers can call Lookup() for the metaobjects to determine the type of a variable name. The returned type is represented by a TypeInfo metaobject.

• TypeInfo metaobject

The TypeInfo metaobjects represent types. The types are not limited to class types. They also include other kinds of types, such as built-in types like int, pointer types, function types, template types, and so on.

The most important member function of TypeInfo is WhatIs(). It returns a unique constant according to the kind of the type. For example, if the type is a class type, WhatIs() returns ClassType.

The TypeInfo metaobjects also supply member functions for obtaining detailed information of each kind of type:

- uint IsBuiltInType()
  This works for built-in types. It returns what the built-in type is, char, int, double, or others.

- Class* ClassMetaobject()
  This works for class types. It returns the Class metaobject for the class.

- void Dereference(TypeInfo& t)
  This works for derived types, such as pointer types, reference types, and function types. It returns the dereferenced type of the type. For example, if the type is int*, Dereference() returns the TypeInfo metaobject for int. If the type is a function type, Dereference() returns the return type of the function.

- BOOL NthArgument(int nth, TypeInfo& t)
  This works for function types. It returns the type of the nth argument to the function. If there is not the nth argument, NthArgument() returns FALSE.

• Class metaobject

The Class metaobjects supply member functions for introspection. Programmers can call these functions to inspect the class; for example, they can obtain the class metaobject for the base class. The followings are part of these member functions:

- Ptree* Name()
  This returns the name of the class.

- char* MetaclassName()
  This returns the name of the metaclass.

- Class* NthBaseClass(Environment* env, int nth)
  This returns the nth base class of the class. Recall that C++ allows multiple inheritance. The leftmost base class is the first, and the rightmost is the last. Note that this member function returns only immediate base classes. It does not return a base class of a base class.

- Ptree* NthMemberName(int nth)
  This returns the name of the nth member, which is either a member function or a data member.

- BOOL LookupMemberType(Environment* env, Ptree* name,
  TypeInfo& t)
  This returns the type of the member specified by name.

Non Local

The OpenC++ MOP also enables non-local processing. Because a single class metaobject processes all the code fragments relevant to the class, programmers usually define only one new metaclass for implementing one abstraction. Even if the processed fragaments spread out over the whole program, they are automatically dispatched to the class metaobject. Although programmers have to define more than one metaclasses if the processing involves multiple classes, we believe that a class is good granularity for source-code processing in C++. This is because most of abstractions are usually implemented by a single class in C++.

For non-local processing, the `Class` metaobjects supply a different member function for each kind of program text. One of these functions is `TranslateNew()` presented in Section 4.1. These functions receive program text and translate it. The translated text is substituted for the original text in the program. Although the member functions supplied by `Class` do not change the received text at all, a subclass of `Class` can override them and implement new source-code translation. The followings are some of the member functions that the subclass can override (For the complete list, see Appendix B):

- `Ptree* TranslateClassName(Environment* env, Ptree* keyword, Ptree* name)`
  This translates the class name appearing in the program.

- `Ptree* TranslateSelf(Environment* env)`
  This translates the class declaration.

- `Ptree* TranslateMemberFunctionBody(Environment* env, Ptree* name, BOOL inlined, Ptree* body)`
  This translates the body of a member function supplied by the class.

- `Ptree* TranslateUnary(Environment* env, Ptree* op, Ptree* object)`
  This translates an expression including a unary operator op. It is called if the `object` that the operator is applied to is an instance of the class.

- `Ptree* TranslateNew(Environment* env, Ptree* ...)`
  This translates the `new` expression.

- `Ptree* TranslateMemberRead(Environment* env, Ptree* member)`
  This translates an expression for reading a data member of an object of the class.

- `Ptree* TranslateMemberCall(Environment* env, Ptree* member, Ptree* arglist)`
  This translates a member function call on an object of the class.

4.3 Syntax Extension

The OpenC++ MOP allows limited syntax extension. Because the C++ grammar is heavily context dependent, the full extensibility for syntax is difficult to provide in C++. However, programmers can define new keywords and implement the following kinds of new syntax:

- Type Modifier
  Programmers can define a new type modifier. It may appear in front of type names, the `new` operator, or class declarations. For example, programmers may define new keywords, `distributed` and `remote`, and then they can write:

```
distributed class Dictionary { ... };
remote Point* p = remote(athos) new Point;
```

  Here, `distributed`, `remote`, and `remote(athos)` are new type modifiers.

- Access Specifier
  Programmers can define a new member-access specifier, which appears within class declarations. `public`, `protected`, and `private` are the built-in access specifiers. For example, if `after` is a keyword for a new access specifier, then programmers may write:

```
class Window {
public:
    void Move();
after:
    void Move() { ... }     // after method
};
```

- Statement
  Programmers can define a new kind of statement that is similar to either the `if` statement or the `while` statement. The following examples are valid syntax extensions:

```
Matrix m1;
m1.forall(e){ e = 0.0; }    // extended syntax

ButtonWidget b;
b.press(int x, int y){      // extended syntax
  cout << "pressed at" << x << "," << y;
};
```

  In these examples, `forall` and `press` are new keywords. Like the `while` statement, they are followed by a `()` expression and a block statement.

To make these syntax extensions available, the programmers first register the keywords to the parser. The `Class` metaobjects supply the following registration functions. Each one registers a keyword for a specific syntax extension (the followings are not all the functions. See Appendix B for more details):

- `void RegisterNewModifier(char* keyword)`
  This registers a `keyword` for a new type modifier.

- `void RegisterNewAccessSpecifier(char* keyword)`
  This registers a `keyword` for a new access specifier.

- `void RegisterNewWhileStatement(char* keyword)`
  This registers a `keyword` for a new kind of statement that is similar to the `while` statement.

The semantics of a new syntax extension is defined by the class metaobject that is involved with the extension. For example, the `forall` statement in this program:

```
Matrix m1;
m1.forall(e){ e = 0.0; }
```

Should be translated into regular C++ code by the class metaobject for `Matrix`. The programmer, therefore, has to define a new class metaobject to handle the translation. The new class metaobject will override this member function:

- `Ptree* TranslateUserStatement(Environment* env,`
  `Ptree* object, Ptree* op,`
  `Ptree* keyword, Ptree* rest)`
  This translates a new kind of statement. The default implementation by `Class` causes a syntax error.

If the `forall` statement is translated, the arguments `object`, `op`, and `keyword` are bound to `m1`, `.` (dot), and `forall`, respectively. The last argument `rest` is bound to the rest of the statement, that is, `(e){ e = 0.0; }`. The overridden `TranslateUserStatement()` should use these arguments and construct the parse tree for the translated statement.

4.4 What is New?

The OpenC++ MOP has been developed by a synthesis and re-engineering of ideas of other known techniques. Especially Lisp macros and the CLOS MOP influence the design of the OpenC++ MOP. This section discusses comparison between the OpenC++ MOP and other techniques.

Comparison with Lisp Macros

In the OpenC++ MOP, class metaobjects process a program as Lisp macros do. Their member functions receive program text and return the translated text, which is substituted for the original text.

However, the OpenC++ provides more contextual information than Lisp macros when translating a program. The member functions for the translation receive the environment as well as program text. They can use it to examine the static types of variables. Lisp macros do not provide the environment; macro functions have to translate a program only with syntactical information but without contextual information.

The OpenC++ MOP also enables self modification whereas Lisp macros do not. With only a simple annotation, it can alter the behavior of the objects of only a specific class. Error-prone programming conventions are not needed. Lisp macros require programmers to explicitly call them to translate programs. The expressions that a Lisp macro should process have to be explicitly preceded by the macro name. Hence, if programmers want to change the behavior of the language when reading a variable, they have to write something like `(read-variable x)` instead of just writing `x`. Here `read-variable` is a macro name. Otherwise, programmers have to write the code walker. On the other hand, the OpenC++ MOP does not need such a programming convention. Once a metaclass is declared, all the expressions involved with the class are automatically processed by the class metaobject. Programmers can alter the behavior of the objects without editing the original program to insert something like macro names. Moreover, programmers can restrict the range of the self modification within a specific class. The behavior for the other classes are kept as is.

Comparison with the CLOS MOP

The CLOS MOP is the immediate ancestor of the OpenC++ MOP. Both the CLOS MOP and the OpenC++ MOP are metacircular, and they employ class metaobjects instead of the metaobjects for objects. The difference is that the OpenC++ MOP is specially designed to run at compile time.

Although the currying technique allows metaobjects in the CLOS MOP to mostly run at compile time, some computation by the metaobjects is still performed at runtime. At least, which metaobject is selected for given base-level code is determined at runtime. See Figure 3.3 in page 32 again. The default interpreter (or compiler) has to determine whether it executes each expression through a metaobject or not, and which metaobject it selects if so. In the CLOS MOP, this is done at runtime.

The OpenC++ performs all meta computation at compile time. It also determines which metaobject is selected at compile time. Because of this feature and the metacircularity, the OpenC++ MOP does not imply any

performance penalty at runtime. The OpenC++ MOP uses static typing to select a metaobject at compile time. It statically types all variables and expressions in the program and determines which metaobject is responsible for the translation. Then it calls the metaobject for translating the expression and directly substitutes the result for the original expression. This means that even the code for selecting a metaobject and dispatching to it does not run at runtime.

The OpenC++ MOP is regarded as a good synthesis of the CLOS MOP and Lisp macros. We below present the definition of `RecordedClass` written in pseudo Lisp. Comparing this definition with the equivalent definition in the CLOS MOP, the readers will intuitively understand the synthesis. First, we show the definition in the OpenC++ MOP:

```
(defclass recorded-class (metaobject-for-class)
  (defmethod create-object (env expr)
    (let ((class-name (<- self name)))
      `(begin (set! *history* (cons ,class-name *history*))
              ,(<- super create-object env expr)))))
```

The next is the definition in the CLOS MOP:

```
(defclass recorded-class (metaobject-for-class)
  (defmethod create-object (init-args)
    (let ((class-name (<- self name)))
      (set! *history* (cons class-name *history*))
      (<- super create-object init-args))))
```

The readers can see that the two definitions are significantly close to each other. The primary difference is that the definition in the OpenC++ MOP returns an expression instead of directly executing the expression. This feature is the influence by Lisp macros.

Comparison with Early Compile-time MOPs

Several MOPs running at compile time, called compile-time MOPs, have been developed earlier than the OpenC++ MOP. The earliest compile-time MOPs are Intrigue [37] and Anibus [50, 51]. They are designed for Scheme [16] and parallel Scheme although they are written in CLOS. However, their design is quite different from the OpenC++ MOP. They provide meta interface mainly for customizing the behavior of the compiler rather than source-code translation. In fact, metaobjects of Intrigue are internal components of the compiler, such as a parser and an optimizer. Metaobjects of Anibus are nodes of a parse tree and controls translation from parallel Scheme to regular Scheme including low-level primitives for parallel computing. Anibus's MOP might look similar to the OpenC++ MOP since both MOPs control program translation, but it does not provide non-locality based on the class system (or the type system because Scheme does not include a class system).

Moreover, Anibus's program translation is to transform a parse tree. The metaobjects perform the transformation instead of generating substituted source code.

CRML [31] has a compile-time MOP for ML [43]. Its MOP provides the capability similar to Lisp macros and makes it possible to use new syntax in ML. However, it also involves the same limitations as Lisp macros. For example, it does not enable self modification.

The MPC++ MOP [33, 34] is another compile-time MOP for C++. As in OpenC++, it allows programmers to control source-code translation. The most significant difference between MPC++ and OpenC++ is that the MPC++ MOP does not provide non-locality based on the class system. Instead, each piece of program is dispatched to a metaobject for translation according to the kind of the parse tree, such as a declaration, an `if` statement, a `+` operator, and so forth. Metaobjects of MPC++ are nodes of the parse tree of the processed program. If a piece of program is declaration of a variable, for example, then it is dispatched to the declaration metaobject (corresponding to the `Ptree` metaobject in OpenC++) for the translation, rather than the class metaobject involved with the type of the declared variable.

Because of this feature, the MPC++ MOP is not suitable for customization that is specific to a particular class and involves several statements and expressions. If the customization needs to translate declaration statements and `->` expressions on a specific class, then the programmer needs to define two new metaclasses for declaration and the `->` operator. Then the new metaclasses must explicitly determine the class of the processed code and translate the code only if it is the specific class. This computation is implicitly performed in OpenC++ because it provides non locality.

Moreover, the MPC++ MOP does not provide the meta-meta level. Thus programmers cannot write a meta-meta level program to make it easier to write a meta-level program. Since the OpenC++ MOP provides the meta-meta level, for example, programmers can enjoy special syntax when writing a meta-level program.

4.5 Summary

This chapter proposed the OpenC++ MOP, which is a mechanism for processing a program. This mechanism provides rich meta representation of the processed program and enables context-sensitive and non-local processing. For context-sensitive processing, the OpenC++ MOP provides `Ptree` metaobjects (for program text), `Environment` metaobjects, `TypeInfo` metaobjects, and `Class` metaobjects. These metaobjects are the meta representation of various aspects of the program. The `Class` metaobjects also work for non-local processing. They control the translation of all the ex-

pressions that are spread out over the whole program but involved with the class.

Like Lisp macros, the OpenC++ MOP also enables syntax extensions. Programmers can register a new keyword and define a new type modifier, an member-access specifier, or a new kind of statement. They are appropriately translated into regular C++ code by `Class` metaobjects as other code is translated.

These features are advantages of the OpenC++ MOP against other existing C++ mechanisms such as inheritance and templates, and they enable C++ libraries that have been impossible or difficult to make them efficient. To enable those libraries, an extended C++ compiler specially prepared for the libraries has been necessary so far. The OpenC++ MOP allows programmers to write a meta-level program to implement such an extension to C++ on top of the compiler.

The OpenC++ MOP has been developed by a synthesis of ideas from Lisp macros and the CLOS MOP shown in the previous chapter. However, it is not just a C++ version of Lisp macros or the CLOS MOP. It is something considerably different from Lisp-style macros or traditional reflective languages.

The basic architecture of the OpenC++ MOP was taken from Lisp macros. In both of them, the meta-level program receives program text and returns the translated one. Moreover, the meta-level program can run at compile time to improve execution performance of the base-level program. On the other hand, the OpenC++ MOP provides richer meta representation than Lisp macros. It provides not only program text but also an environment and type information. Also, it provides non-locality based on the class system of C++. Furthermore, self modification is possible in the OpenC++ MOP. Programmers can alter the behavior of the language without programming conventions or the code walker.

The protocol structure of the OpenC++ MOP is based on that of the CLOS MOP. The OpenC++ MOP is metacircular, and class metaobjects are the subjects of processing a program. However, the OpenC++ MOP is designed so that metaobjects run at compile time. All the meta computation, that is, source-code processing, is performed at compile time. Even which metaobject should be selected for processing is determined at compile time. The CLOS MOP performs this selection at runtime although other meta computation can be moved to compile time with the currying technique.

Chapter 5

Meta Helix

Like the CLOS MOP, the OpenC++ MOP is metacircular since metacircular systems are easy to learn and make it easier to write efficient meta programs. Unfortunately, pure metacircularity can lead to a problem we call implementation level conflation. This problem confuses programmers and causes errors, hence we could not adopt pure metacircularity *as is* for the OpenC++ MOP.

Instead, to avoid the problem, we have developed an improved version of metacircular architecture, named the meta helix [15]. This chapter presents a problem of the pure metacircular architecture, and proposes the meta helix as a solution. The meta helix preserves advantages of metacircularity but also addresses the problem we present.

5.1 Implementation Level Conflation

In a metacircular system like the CLOS MOP, the language can be customized in the customized language itself. This means not only that the base-level program and the meta-level program are written in the same language, but also that the customization by the meta-level program reflects on the language in which the meta-level program itself is written. This feature gives advantages that we presented in Chapter 3, but it can also lead to a problem which we call *implementation level conflation*.

Lisp macros

First of all, we show an example of implementation level conflation in Lisp macros, which is also metacircular. Since Lisp macros are relatively simpler than the CLOS MOP, this example might look trivial. However, we believe that showing this example help the readers understand a more serious example we show later.

Suppose that we modify the behavior of a special form `define` so that

more than one variables can be defined at the same time. With this extension, for example, we can define variables x and y with the initial value 7 by this single expression:

```
(define x y 7)
```

This expression should be expanded by a macro function named define into this:[1]

```
(begin (define x 7) (define y 7))
```

Therefore, the definition of the macro should be this:

```
(define-macro (define . args)
  (let ((vars (list-head args (- (length args) 1)))
        (init (car (last args))))
    `(begin ,@(map vars
                   (lambda (v) `(define ,v ,init))))))
```

Unfortunately, this macro does not work; it will fall down into an infinite loop. Because the symbol define is now a macro name, occurrences of define in the expanded expression are repeatedly processed by the macro function. For example, the expanded expression:

```
(begin (define x 7) (define y 7))
```

Will be expanded again into:

```
(begin (begin (define x 7)) (begin (define y 7)))
```

And this expression also invokes macro expansion, and so on.

This problem can be fixed although the way of fixing is quite implementation dependent. To fix this problem, we have to define the macro as follows:

```
(define define* define)                          // alias
(define-macro (define . args)
  (let ((vars (list-head args (- (length args) 1)))
        (init (car (last args))))
    `(begin ,@(map vars
                   (lambda (v)
                     `(define* ,v ,init))))))  // use define*
```

This program first binds a symbol define* to the original define special form. Now define* is an alias of define. Then, the program defines a macro define and uses define* for the expanded expression. Using the alias makes the macro function avoid the infinite loop since the expanded expression does not include the macro name define any more.

[1] For simplicity, we assume that the sub-expression for the initial value does not cause side-effects.

A More Serious Example

The problem of the example shown above is that two distinct concepts, which are the define special form and the define macro, are conflated into a single name. We call this problem implementation level conflation because the implemented level (define macro) and the implementing level (define special form) are conflated into a single structure.

This conflation becomes more serious in metacircular systems like the CLOS MOP. The power of the CLOS MOP is that programmers can enjoy metacircularity and use the existing facilities of CLOS as much as possible when they write meta-level programs. But this is a double-edged blade. It can also easily lead to conflation of a new abstraction and the facilities implementing that abstraction.

To illustrate more serious implementation level conflation, we show an example of an extension written with the CLOS MOP. This extension records all accesses to slots (i.e. data members) of objects. The following program is an example of a program that uses this extension:[2]

```
> (defclass point ()
    (variable x y) :meta history-class)
POINT
> (define p1 (make-instance point))
P1
> (<- p1 x 3)              ; set x to 3
3
> (<- p1 y 2)              ; set y to 2
2
> (<- p1 x)                ; read x
3
> (slot-history p1)        ; show the access log
((GET X) (SET Y) (SET X))
```

Since the metaclass for point is history-class, all the accesses to the slots of point objects are recorded, and the access log is available through the function slot-history.

Slots with access history can be implemented using the existing slot mechanism. The way this works is simply for instances of point to actually have three slots, the two visible slots x and y as well as a third, "hidden" slot history for storing the access log. The definition of history-class implemented in this way is as follows:

```
(defclass history-class (metaobject-for-class)
  ; what slots do the instances have?
  (defmethod compute-slots ()
    (cons 'history (<- super compute-slots)))

  ; read a slot of an instance
```

[2] Again, we use the same altered syntax that we used in Chapter 3.

```
(defmethod read-slot (object slot-name)
  (<- object history
      `((get ,slot-name) ,@(<- object history)))
  (<- super read-slot object slot-name))

(defmethod set-slot ...))
```

This metaclass overrides three methods. First, it overrides `compute-slot` so that every history class has an extra slot to store the access log. Also it overrides `read-slot` and `set-slot`, which reads and writes a slot of the instance of history classes. They updates the access log before actually reading or writing. Since the access log is stored in the hidden slot `history`, they read and write the slot to update the access log.

Unfortunately, this example includes implementation level conflation. There are two distinct but conflated concepts in this implementation. One is a `point` object that has extended slots. The accesses to the slots are recorded in the access log. The other concept is a `point` object that has three non-extended slots, `x`, `y`, and hidden `history`. This object is used to implement the former object. Since the CLOS MOP is metacircular, the two `point` objects are identical — the same object. Again we are dealing with the implemented level (extended slots) and the implementing level (non-extended slots) but the two levels are conflated into a single structure (`point` object).

Problems for Users of the Extension

Implementation level conflation results in confusion for users of the extension. For example, consider that a programmer uses one of the MOP's introspective facilities to ask what slots a class has:

```
> (class-slots (find-class 'point))
(HISTORY X Y)
```

According to the specifications of the CLOS MOP, `class-slots` returns the list of the slot names that the given class has. The result, however, includes the `history` slot. Is this right? What does "the given class" mean in the specifications? Is it the implementing class or the implemented class? It should be the implemented class and hence, since the `history` slot exists only for the implementation, including the slot is entirely inappropriate.

This problem particularly shows up when using browsers and debuggers that rely on the introspective part of the CLOS MOP to work. Exposing this detail of how slots with history is implemented can leave programmers confused, or worse yet, can tempt them to rely on this implementation details in ways that they should not.

Problems for the Implementor of the Extension

Implementation level conflation also can cause problems for the implementor of the extension. A careful reading of the method `read-slot` supplied by `history-class` provides an example of this.

The implementation of `read-slot` has a bug which manifests itself as infinite recursion although it is better understood as resulting from the implementation level conflation. Operationally, the bug is that the body of the method, as part of updating the access log, must read the `history` slot, which runs this method recursively, which starts to update the access log, which reads the `history` slot, and so on ad infinitum. This bug happens because `read-slot` is implemented with a non-extended slot `history` but `read-slot` alters the language to record all the accesses to the slots, including `history`.

The standard solution to this problem is to introduce a special purpose test that prevents the infinite recursion. So the revised code ends up looking something like this:

```
(defmethod read-slot (object slot-name)
  (unless (eq slot-name 'history)
    (<- object history
        `((get ,slot-name) ,@(<- object history))))
  (<- super read-slot object slot-name))
```

This solution, while effective, seems ad hoc, can be difficult to reason about, and is not effective in general.

The Importance of an N-to-N Correspondence

The problems shown above can be better described as having to do with implementation level conflation. As mentioned at the beginning, there are two concepts of slots in play: extended slots `x` and `y`, the accesses to which are recorded, and non-extended slots `x`, `y`, and `history`, which are used to implement the extended slots. But, because of the conflation, there is only one structure, or a handle to refer to the two concepts. The bug happens since the handle always refers to the extended slots. If, somehow, the slots within the body of `read-slot` could refer to non-extended slots, then the ad-hoc solution to the infinite recursion could be avoided.

Fundamentally, if there are n distinct important views of an object — or any other structure — there needs to be n distinct handles to it. For example, we need some way for instances of the class `point` to be viewed in terms of either the implemented functionality or the implementing functionality, not a conflation of both. The users of the extension want a view in terms of the implemented functionality, but the implementor of the extension wants to be able to take one view or the other at different times.

5.2 Inadequate Solutions

Implementation level conflation is a flaw of the metacircular architecture. Despite the advantages, this architecture makes metaobject protocols lose elegance; the metaobject protocols become inconsistent and difficult to use. Because of this fact, we could not adopt the pure metacircular architecture for the OpenC++ MOP. Instead, we have developed a improved metacircular architecture since the OpenC++ MOP should be elegant, simple, and easy to use, as much as possible.

Before presenting our solution to the problem of implementation level conflation, we first present two earlier solutions that, in different ways, fail to our needs. These solutions serve to further flesh out the criteria which the more general solution should meet.

Change the Implementation

One possible solution to this problem involves implementing the extension in a different way, specifically by storing the access log in the class metaobject rather than directly in the objects themselves. The following program implements history classes in this way:

```
(defclass history-class (metaobject-for-class)
  (variable history)    ; place to store the access log

  (defmethod read-slot (object slot-name)
    (let ((log (assq object (<- self history))))
      (set-cdr! log '(get ,slot-name) ,@(cdr log))
      (<- super read-slot object slot-name)))

  (defmethod set-slot ...))
```

All the access logs for the instances are stored in the slot `history` of the class metaobjects (that is, a class variable in the Smalltalk terminology). So the value of `history` is a list of pairs of an object and its access log. This association list can be searched by `assq` with using `object` as a search key.

This solution solves the specific problems mentioned in the previous section, but it loses advantages of metacircularity, which we would like to keep. This solution must manually implement the mapping from individual objects to their access logs, even though that basic functionality is already present in CLOS. Implementing that mapping is not needed if each access log is directly stored in a slot of the object. This solution is not only redundant, but also difficult to achieve sufficient execution performance. The implementor of `history-class` has to implement as efficient a mapping mechanism as the default one, which is a slot of an object. At least, the implementation shown above does not satisfy this performance criterion; the association list is significantly inefficient.

Tiny CLOS MOP

If we give up metacircularity, we can avoid implementation level conflation. Non-metacircular systems such as 3-Lisp and 3-KRS do not involve this problem because they have two distinct handles for implemented functionality and implementing functionality. In 3-KRS, if programmers want a view in terms of the implemented functionality, they should refer to the base-level objects, but if they want a view in terms of the implementing functionality, they should refer to the associated metaobjects. Since there is no metacircularity, the base-level objects and the metaobjects are never conflated. The behavior of the metaobjects does not change depending on the metaobjects themselves.

The Tiny CLOS MOP developed by Gregor Kiczales et al avoids implementation level conflation by partially giving up metacircularity. The Tiny CLOS MOP provides two different abstractions for per instance storage: slots and fields. Fields are a lower-level abstraction used to implement slots; they represent memory image allocated for implementing each object. The base-level Tiny CLOS programs never know the fields exist.

In the specific example of the history class, the extension works by allocating an extra *field* for each object. So, for example, `point` objects have two slots `x` and `y`; but they have three fields, for holding the `x` and `y` slots and the slot access log. The implementation of `history-class` in Tiny CLOS looks like:

```
(defclass history-class (metaobject-for-class)
  (variable history-index) ; the index of the field that will
                           ; store the access log for instances
                           ; of the class

  ; allocate an extra field and remember its index
  (defmethod compute-fields ()
    (<- self history-index (<- self allocate-field))
    (<- super compute-fields))

  (defmethod read-slot (object slot-name)
    (let* ((index (<- self history-index)))
      (<- self set-field object index
          '((get ,slot-name)
            ,@(<- self get-field object index))))
    (<- super read-slot object slot-name))

  (defmethod set-slot ...))
```

Fields are accessed through the methods `get-field` and `set-field`, and each field is specified by its index instead of its name. Fields have a more primitive naming mechanism in terms of indices.

Again, this solution solves the specific problems mentioned in the previous section. Implementation level conflation is avoided because (1) no

"hidden"slot for the accesses log is added, and (2) the access log stored in an object is retrieved through the lower-level methods `get-field` and `set-field`, which do not invoke `read-slot` recursively. Since the Tiny CLOS MOP provides two distinct handles, slots and fields, for implemented functionality and implementing functionality, programmers can select an appropriate view by switching the handles.

Unfortunately, this solution has significant problems of its own which make it unsuitable as a general solution. First, this solution loses advantages of metacircularity. Because the Tiny CLOS MOP is not metacircular in terms of slots, programmers have to learn lower-level abstractions and writing efficient meta-level programs is difficult. Second, this solution is only effective in the presence of a single extension to slot functionality. If, for example, someone wanted an additional extension (i.e. to store the objects in a persistent database [47]) there would still be conflation. This is because in such a situation there needs to be (at least) three views. The view of persistent objects with a slot access history, built on top of the view of persistent objects, built on top of ordinary objects. But the Tiny CLOS MOP provides only two levels, so there will still be some conflation. To avoid implementation level conflation in n levels of implementation, the MOP must provide support for n different views.

5.3 The Meta Helix Architecture

The common idea underlying the two unsatisfactory solutions is to distinguish the implemented and implementing functionality by using a different "handle" for each. In the first proposed solution, the objects are the handle to the implemented functionality, and the class metaobject is the handle to the implementing functionality. In the second proposed solution, slots are the handle to the implemented functionality, while fields are the handle to the implementing functionality.

But the problem with both of these solutions is that the benefits of metacircularity is destroyed by the fact that the two handles are distinct abstractions. The idea of our proposed solution is to address its problem by providing two handles, but to retain the benefits of metacircularity by the

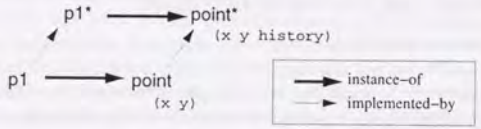

Figure 5.1: Two handles with the same abstraction

use of the same abstraction for the two handles. For example, in the case of the `history-class` extension, our solution has two class metaobjects, `point` and `point*`, to represent different implementation levels (Figure 5.1). The class `point` corresponds to the class in the extended language, which keeps slot access histories, while the class `point*` corresponds to the class in the non-extended language, which is used to implement the extended one.

Note that the two handles indicate the same entity. For example, the class `point` and the class `point*` corresponds to the same class. They are just handles to distinct views of the same entity as slots and fields are handles to distinct views of per-instance storage in the Tiny CLOS MOP. `point` is the handle to the implemented view and `point*` is the handle to the implementing view.

Although the two handles, such as `point` and `point*`, are often very similar, the relationship between the handles does not fit the usual `subclass-of` relationship. It cannot be either the `instance-of` or `base-meta` relationships. To capture the relationship between the two handles, we introduce an `implemented-by` relationship. In Figure 5.1, the class `point` is implemented-by the class `point*`, and the object `p1` is implemented-by the object `p1*`. The objects `p1` and `p1*` are instances-of `point` and `point*`, respectively.

If the implemented-by relationship is used, the problems by implementation level conflation can be easily solved. First, the programmer who want to ask what slots a class has can obtain an appropriate result:

```
> (class-slots (find-class 'point))
(X Y)
> (class-slots (implemented-by (find-class 'point)))
(HISTORY X Y)
```

Second, the method `read-slot` becomes more simple and easy to read. No ad-hoc techniques for avoiding infinite recursion are needed any more:

```
(defmethod read-slot (object slot-name)
  (let ((object* (implemented-by object)))
    (<- object* history
        `((get ,slot-name) ,@(<- object* history)))
    (<- super read-slot object slot-name)))
```

Note that `read-slot` is the method for not the class metaobject for `point*` but the class metaobject for `point`. The behavior of the object `p1`, which has slots with history, is controlled by the class metaobject for `point`.

Our choice of the name meta-helix for this architecture is best seen when thinking in terms of the relation between handles that the different solutions use. As shown in Figure 5.2, in the pure metacircular approach, the implementation loops directly back onto itself—leading to conflation. In the Tiny CLOS approach, the implementation maps between two distinct functionalities—leading to added complexity. In the meta-helical approach,

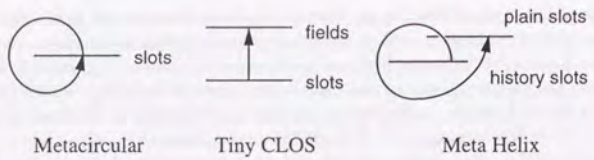

Figure 5.2: The implementation relation between interfaces

the implementation spirals between two distinct handles of (nearly) identical functionality—preserving what is good about the metacircular approach, while still reifying the distinction that prevents conflation of implementation levels.

Similarly, the meta helix works when there are more than two implementation levels. So, for example, in the case of the persistent history class mentioned in the previous section, we can create the three levels that are needed to maintain separate views, and relate them using implemented-by relationships. This is shown in Figure 5.3, which illustrates the helical nature of this architecture.

While distinguishing implementation levels, the meta helix architecture preserves the benefits of metacircularity because the meta helix architecture is a super set of the metacircular architecture. Except the implemented-by relationship, the meta helix architecture is quite identical to the metacircular architecture. So, for example, unlike the Tiny CLOS MOP, programmers do not need to learn new abstractions such as fields to implement an extended concept of slots. Also, writing an efficient meta-level program is still easy.

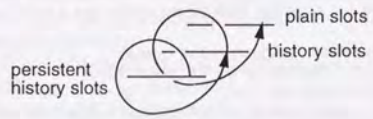

Figure 5.3: The Meta Helix supports n implementation levels.

5.4 Implementing the Meta Helix

The OpenC++ MOP is based on the meta helix architecture. This section first shows this fact, especially focusing on how the OpenC++ reifies

the implemented-by relationship. The way of realizing the implemented-by relationship is a main issue for implementing the meta helix architecture. Then this section shows how the implemented-by relationship should be reified for the CLOS MOP. Through this example, we present that the meta helix architecture is applicable to not only the OpenC++ MOP but also other kinds of MOPs.

The OpenC++ MOP is Meta-Helical

In the OpenC++ MOP, the implemented and implementing levels are naturally separated; the implemented level is the original program before translation and the implementing level is the resulting program of the translation. To implement the meta helix architecture, therefore, we should be just able to distinguish classes in the original program from the classes in the resulting program, and relate the corresponding classes by the implemented-by relationship.

The implemented-by relationship in the OpenC++ MOP is reified with an aliasing technique, which we used to avoid implementation level conflation caused by the macro `define` in Section 5.1. To do this, the OpenC++ MOP includes the following rule to be meta-helical:

- Any class appearing in the original program must be renamed in the resulting program of the translation.

This rule refies the implemented-by relationship. For example, in the example of history classes, the class `point` should be renamed `point*`. This aliasing gives programmers two handles to the implemented and implementing functionality. One handle is the class `point` for the implemented functionality, and the other is the class `point*` for the implementing functionality. By switching the two class names, programmers can avoid implementation level conflation. First, they are not confused by the introspective part of the MOP any more:[3]

```
> (class-of 'point)
(X Y)
> (class-of 'point*)
(X Y HISTORY)
```

Also, the method `read-slot` is implemented without confusion in an intuitive way (This implementation uses nested backquotes for emphasizing the similarity to other versions of implementations. Although nested backquotes might make the program look complex, the complexity does not result from the meta helix.):

[3]For making the argument clearer, we use Lisp-style syntax to show an OpenC++ program.

```
(defmethod read-slot (env object slot-name)
  `(begin (<- ,object history
              `((get ,',slot-name) ,@(<- ,object history)))
          ,(<- super read-slot env object slot-name)))
```

For example, this method translates an expression:

```
(<- p1 x)
```

into this expression:

```
(begin (<- p1 history
           `((get x) ,@(<- p1 history)))
       (<- p1 x))
```

Note that the type of `p1` is not the class `point` but the class `point*` after the translation. During the translation, all occurrences of the class name `point` are replaced with `point*`. Therefore, after the translation, the expression `(<- p1 history)` is not recursively processed by the class metaobject for `point`. Rather, it is processed by the class metaobject for `point*`, which is a distinct metaobject and would be the default one. If programmers want to recursively process, they can declare that the type of `p1` is still the class `point`.

The OpenC++ MOP also supports multiple implementation levels. The persistent history slots shown in Figure 5.3 can be implemented by just specifying a non-default metaclass `persistent-class` for `point*`. If the metaclass is not the default one, all the expressions involved with the class `point*` are recursively translated by the class metaobject for `point*`. This process is repeated until the metaclass becomes the default one.

A Meta-Helical Version of the CLOS MOP

The meta helix is applicable to other kinds MOPs such as the CLOS MOP. In the OpenC++ MOP, the implemented and implementing functionality are naturally separated into the original program and the translated program. In the CLOS MOP, however, the implemented and implementing functionality coexist in the same runtime environment. So we need a different technique to reify the implemented-by relationship.

We present that delegation works for reifying the implemented-by relationship for the CLOS MOP. In a meta-helical version of the CLOS MOP, a class metaobject is responsible for defining the class that will implement it. The class metaobject for `point` will produce a class `point*` equivalent to the definition:

```
(defclass point* ()
  (variable x y history))
```


As in the OpenC++ MOP, the class `point*` is the handle to the implementing functionality. The class metaobject for `point` delegates most of its work to the class metaobject for `point*`. For example, when an instance of `point` is created, the class `point` asks the class `point*` to create a `point*` object. The primitive function `implemented-by` on a `point` object returns the `point*` object. The slot access primitive `read-slot` is then specialized by the class metaobject for `point`. Its implementation is:

```
(defmethod read-slot (object slot-name)
  (let ((object* (implemented-by object)))
    (<- object* history
        `((get ,slot-name) ,@(<- object* history)))
    (<- super read-slot object slot-name)))
```

The method `read-slot` first gets the implementing object `object*` for that object and then updates its `history` slot. The method finally invokes `read-slot` supplied by the super class, which delegates the actual work of implementation to the class metaobject for `point*`.

5.5 Summary

This chapter presented a new analysis of a problem that arises in existing metacircular systems. This analysis shows how pure metacircularity causes confusion when there are not clearly distinguished views of the implemented and implementing functionality. We call this problem implementation level conflation. This chapter first shows an example of the confusion in Lisp macros, which is a simple metacircular system. Then it shows a more serious example with the CLOS MOP. The confusion makes troubles for both the users of the language extension and the implementor of the extension.

Because of implementation level conflation, we did not adopt the pure metacircular architecture for the OpenC++ MOP. Although implementation level conflation does not reduce the capability of the MOP for processing programs, it severely impacts the elegance of the design of the MOP. The elegance is a significant matter because making it really easy for progarmmers to prodcess programs is one of the design goals of metaobject protocols in general.

To avoid implementation level conflation while keeping benefits of metacircularity, we have developed a improved metacircular architecture, named the meta helix, for the OpenC++ MOP. It addresses the problem by providing two distinct handles to the implemented and implementing functionality while keeping benefits of metacircularity by using the same abstraction for the two handles. Programmers can enjoy metacircularity and, if needed, switch the handles to distinguish the implementing level from the implemented level. The meta helix is not only for the OpenC++ MOP but also

other kinds of MOPs. To show this, this chapter also presented a meta-helical version of the CLOS MOP.

Chapter 6

Libraries in OpenC++

Because of the ability for context-sensitive and non-local processing, the OpenC++ MOP makes it possible to include useful control/data abstractions in a library and, if it is already possible in regular C++, to implement the abstractions more efficiently. This chapter illustrates examples of libraries that the OpenC++ MOP enables but regular C++ does not.

The first two examples show abstractions that the OpenC++ MOP enables, and the next example presents that we can write a class library for metaclasses and make it easier to write similar metaclasses. Then, we mention that the OpenC++ MOP is also effective for meta-level programming. A few abstractions provided by the OpenC++ MOP for meta-level programmers are implemented by the MOP itself. Finally, we show examples of abstractions that the OpenC++ MOP can make efficient.

6.1 Named Object Library

The OpenC++ MOP makes it possible to implement the named object library that we presented in Chapter 2. With this library, the users can get the class name of an object at runtime. For example, the users may write something like this:

```
class Complex : public NamedObject {
public:
    double r, i;
};

void f(Complex* x)
{
    cout << "x is " << x->ClassName();
}
```

If invoked, a function `f()` displays "`x is Complex`".

To implement this library, the developer needs to write two kinds of programs: a base-level program and a meta-level program. The base-level

program defines a library class `NamedObject` and it is linked with the user program. The meta-level program defines a metaclass for processing the user program. We first show the base-level program:

```
metaclass NamedObject : NamedObjectClass;
class NamedObject {};
```

The base-level program defines a library class `NamedObject`. Also, the program declares that the metaclass of `NamedObject` is `NamedObjectClass`. Note that the class `NamedObject` does not include the member function `ClassName()`. It is automatically inserted to `NamedObject` and its subclasses by the meta-level program. For example, the subclass `Complex` defined by the library user is translated by the meta-level program into:

```
class Complex : public NamedObject {
public:
    double r, i;
    virtual char* ClassName() { return "Complex"; }
};
```

The meta-level program defines the metaclass `NamedObjectClass`, which performs the translation mentioned above. Since a subclass of `NamedObject` inherits the metaclass from `NamedObject`, the metaclass `NamedObjectClass` controls the translation on the class `Complex` as well although there is no explicit metaclass declaration. The definition of `NamedObjectClass` is as follows:

```
class NamedObjectClass : public Class {
public:
    NamedObjectClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateBody(Environment* env, Ptree* body){
        Ptree* mf = Ptree::qMake(
            "public:"
            "virtual char* ClassName() {"
            "  return \"`Name()`\";"
            "}\n");
        return Ptree::Append(body, mf);
    }
};
```

This metaclass overrides the member function `TranslateBody()`, which is invoked to translate members included in a class declaration. The overridden member function just constructs a `Ptree` metaobject that represents the member function `ClassName()`, and appends it to the other members. `Ptree::qMake()` is a member function provided by the OpenC++ MOP. It constructs a `Ptree` metaobject according to the given format. Unlike a similar function `Ptree::Make()`, it computes an expression appearing in the format if the expression is surrounded by back-quotes (`). In the example above, `Name()` is computed at compile time and the resulting value is

embedded in the constructed `Ptree` metaobject. `Name()` returns the name of the class metaobject. The difference between `qMake()` and `Make()` is analogous to the difference between the back-quote notation and the quote notation in Lisp (As for the back-quote notation, see Appendix A). We show the implementation of `qMake()` later in this chapter since its implementation needs the OpenC++ MOP; The member function `qMake()` cannot be implemented in regular C++.

To use this library, programmers first need to compile the meta-level program by the OpenC++ compiler, and then they have to link the compiled code with the original OpenC++ compiler. Suppose that the file name of the meta-level program is `nameclass.cc`:

```
% occ -- -o myocc opencxx.a nameclass.cc
```

The resulting executable module `myocc` is an extended OpenC++ compiler with which programmers can use the metaclass `NamedObjectClass`. The linked archive `opencxx.a` is an archive including the original OpenC++ compiler `occ`.

The library users compile their program by `myocc`. For example, to compile a source file `complex.cc`, the users may say something like this:

```
% myocc complex.cc
```

This command compiles `complex.cc` by the extended OpenC++ compiler and produces an executable module.

6.2 Distributed Object Library

Distributed objects are another example of data abstractions that regular C++ cannot include in a library. This section illustrates how the OpenC++ MOP works for including this abstraction in a library. This example needs more member functions to be inserted in library users' classes by a metaclass. Hence, through this example, we illustrate how to use various metaobjects like a `TypeInfo` metaobject for meta-level programming.

Developing a library with the OpenC++ MOP follows three steps: (1) determine what a user program should look like, (2) figure out what the user program should be translated into and what runtime library is needed to run the translated program, and (3) write a meta-level program to perform the translation and also write the necessary runtime library. We present the implementation of the distributed object library in the order of the three steps.

What the user program should look like

The distributed object library helps the users write a program with distributed objects. The users should be able to define a distributed object as easy as they define a non-distributed object.

For example, the users write something like this code:

```
metaclass Rectangle : DistributionClass;
class Rectangle {
public:
    Rectangle(int l, int h) { length = l; height = h; }
    int Stretch(int l, int h) {
        length += l; height += h; return length * height;
    }
            :
    int length, height;
};
```

Note that a class `Rectangle` turns to a class for distributed objects by just putting a metaclass declaration.

Once putting the declaration, the user can create a `Rectangle` object on a server machine and access it from a client machine without concern of the location of the object:

```
// server side
main()
{
    Rectangle* r = new Rectangle(3, 4);
    Export(r, "rect", Rectangle);
    ServerLoop();
}
```

This program first creates a `Rectangle` object and then exports the object for clients. The (macro) function `Export()` is a library function for exporting a distributed object with a global name. The first argument is the exported object, the second argument is the global name, and the third argument is the type name of the exported object. In the example above, a `Rectangle` object `r` is exported with a global name `"rect"`. The function `ServerLoop()` is another library function, which starts waiting for requests from a client. The server program has to call this library function after all distributed objects are ready.

After `ServerLoop()` is called, the client program can freely access the distributed object on the server machine:

```
// client side
main()
{
    StartupClient("calvin");
    Rectangle* obj = Import("rect", Rectangle);
    cout << "new size: " << obj->Stretch(1, 3);
}
```


The client program first calls a library function `StartupClient()`, which connects the client to the server machine specified by the argument. In the example above, the client connects to the server machine named "`calvin`". Then the client program imports a distributed object from the server by a library (macro) function `Import()`. Once the distributed object is imported, the client program can deal with the object in the same way that it deals with ordinary objects.

What the user program should be translated into

To run the user program shown above, the meta-level program of the library needs to translate the user program and inserts marshalling and unmarshalling code. The marshalling code converts function arguments into a byte stream so that lower-level functions can handle and send them to a remote machine. The unmarshalling code performs the reverse conversion.

First, the member function call appearing in the client program should be translated so that marshalling code is inserted. The underlined code in the original client program:

```
// client side
main()
{
    StartupClient("calvin");
    Rectangle* obj = Import("rect", Rectangle);
    cout << "new size: " << obj->Stretch(1, 3);
}
```

should be translated into this:

```
// client side
main()
{
    int i;
    StartupClient("calvin");
    Rectangle* obj = Import("rect", Rectangle);
    cout << "new size: "
         << (i = 0,*(int*)&mBuf[i]=1,i+=sizeof(int),
             *(int*)&mBuf[i]=3,i+=sizeof(int),
             CallRemote(i,obj,2));
}
```

The substituted code shown by the underline copies two integer arguments `1` and `3` into an array of characters `mBuf`. This copying involves type conversion.[1] The variable `i` means that the size of the copied arguments. Then, a library function `CallRemote()` is called for sending the arguments stored

[1] The type conversion shown above assumes that all machines are based on the same architecture. For real systems, it should absorb difference between architectures, such as little endian and big endian.

in mBuf to the server machine. CallRemote() deals with the arguments in mBuf just as a simple byte stream. Since the implementation of the marshalling code depends on the signature of Stretch(), it should be inserted by the meta-level program.

At the server side, unmarshalling code needs to be inserted. The user program should be translated into the following program:

```
class Rectangle {
public:
    Rectangle(int l, int h) { length = l; height = h; }
    int Stretch(int l, int h) {
        length += l; height += h; return length * height;
    }
            :
    int length, height;
    void Dispatch(int*, void*, int);
};

void Rectangle::Dispatch(int* buf, void* obj, int member)
{
    switch(member){
        :
    case 2 :    // if Stretch() is called
      {
        int s = 0;
        int p1 = *(int*)&buf[s];
        s += sizeof(int);
        int p2 = *(int*)&buf[s];
        s += sizeof(int);
        *(int*)buffer = ((Rectangle*)obj)->Stretch(p1, p2);
      }
      break;
        :
    };
}
```

After the translation, a member function Dispatch() is appended to the class Rectangle. Dispatch() is used to invoke a member function for a distributed object when a client program calls the member function. It receives a byte stream (buf), a pointer to the object (obj), and an integer indicating the called member function (member). It unmarshals the byte stream to function arguments according to the value of member, and invokes the called member function with the unmarshalled arguments. Note that the unmarshalling code also depends on the signature of the called function and hence it needs to be produced by the meta-level program.

Write a runtime library

After determining what a user program should be translated into, the library developer writes a runtime library that the translated user program uses to run. It is an ordinary library written in regular C++ and includes such functions as Export(), Import(), ServerLoop(), CallRemote(), and so on. We show the implementation of the runtime library in Appendix C.1. The readers who are interested in details may see it.

Write a meta-level program

The rest of the work that the developer has to do is to write a meta-level program for the translation mentioned above. The meta-level program defines the metaclass DistributionClass, which overrides member functions TranslateSelf() and TranslateMemberCall() inherited from the default metaclass Class. TranslateSelf() controls the translation of a class definition and TranslateMemberCall() controls the translation of a member function call expression. Since the whole meta-level program is about a hundred lines, we show it in Appendix C.1 and here present just highlights of the program.

The role of the overridden TranslateSelf() is to produce a member function Dispatch() and inserts it into the user program. This is the implementation of TranslateSelf():

```
Ptree* DistributionClass::TranslateSelf(Environment* env)
{
    Ptree* name;
    TypeInfo t;
    int i;
    Ptree* code = nil;
    for(i = 0; (name = NthMemberName(i)) != nil; ++i){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId
           && whatis != PtreeDestructorId
           && LookupMemberType(env, name, t)
           && t.WhatIs() == FunctionType)
        {
            code = AppendDecoder(code, name, i, t);
        }
    }

    AppendAfterToplevel(Ptree::qMake(
        "void `Name()`::Dispatch(int* buf, void* object,"
                                 "int member){\n"
            "switch(member){\n `code` }}"));

    return Class::TranslateSelf(env);
}
```

This member function calls AppendDecoder() for every member function

that the translated class has. In the `while` loop, each member is retrieved by `NthMemberName()` and the type of the member is examined to determine whether the member is a member function or a data member. The member type is represented by a `TypeInfo` metaobject `t` returned by `LookupMemberType()`. If the member is neither a data member, a constructor, or a destructor, then the member function `AppendDecoder()` is called for the member. `AppendDecoder()` produces a `case` block and appends it to `code`. For example, `AppendDecoder()` produces the following code for the `Stretch()` member function:

```
case 2 :
  {
    int s = 0;
    int p1 = *(int*)&buf[s];
    s += sizeof(int);
    int p2 = *(int*)&buf[s];
    s += sizeof(int);
    *(int*)buf = ((Rectangle*)obj)->Stretch(p1, p2);
  }
  break;
```

The code produced by `AppendDecoder()` is included by the implementation of `Dispatch()`, which is eventually inserted by `AppendAfterToplevel()` just after the translated class definition.

`TypeInfo` metaobjects are also used in `AppendDecoder()`. For example, the following `for` loop is part of `AppendDecoder()`:

```
TypeInfo atype;
for(i = 0; t.NthArgument(i, atype); ++i){
    Ptree* argtype = atype.MakePtree();
    code = Ptree::Snoc(code, Ptree::qMake(
                "`argtype` p`i` = *(`argtype`*)&buf[s];\n"
                "s += sizeof(`argtype`);\n"));
}
```

This `for` loop uses `TypeInfo` metaobjects to produce the code for retrieving arguments from a network message stored in `buf`. The variable `t` is the `TypeInfo` metaobject for the type of the processed member function. Note that the type of each argument is obtained by calling `NthArgument()` for this metaobject. `MakePtree()` is another important member function of `TypeInfo`. It converts the `TypeInfo` metaobject to a `Ptree` metaobject that represents the type name. In the code shown above, `MakePtree()` is used to obtain the type name of each argument.

The metaclass `DistributionClass` also overrides `TranslateMemberCall()`, which translates a member function call expression. For example, it translates an expression in the user program:

```
obj->Stretch(1, 3)
```


into something like this expression:

```
int i;
  :
(i = 0,*(int*)&mBuf[i]=1,i+=sizeof(int),
 *(int*)&mBuf[i]=3,i+=sizeof(int), CallRemote(i,obj,2))
```

The implementation of `TranslateMemberCall()` is similar to the implementation of `AppendDecoder()`. It uses `TypeInfo` metaobjects and produces an expression that converts function arguments to a network message stored in `mBuf`:

```
Ptree* DistributionClass::TranslateMemberCall(
                Environment* env, Ptree* object,
                Ptree* op, Ptree* member, Ptree* arglist)
{
    TypeInfo ftype, atype;
    int id = IsMember(member);

    PtreeIter next(Ptree::Second(arglist));
    Ptree* code = nil;
    Ptree* tmp = Ptree::GenSym();

    env->InsertDeclaration(Ptree::qMake("int `tmp`;"));
    LookupMemberType(env, member, ftype);
    for(int i = 0; ftype.NthArgument(i, atype); ++i){
        Ptree* p = next();
        Ptree* tname = atype.MakePtree();
        code = Ptree::Snoc(code, Ptree::qMake(
              "*(`tname`*)&mBuf[`tmp`]"
              "= `TranslateExpression(env, p)`,"
              "`tmp` += sizeof(`tname`),"));
        next();                 // skip ,
    }

    return Ptree::qMake("(`tmp`=0,`code`"
                        "CallRemote(`tmp`, `object`, `id`))");
}
```

This member function first inserts a variable declaration in the processed program by calling `InsertDeclaration()` for `env`. This declares a temporary variable used in the translated expression. The name of the temporary variable is given by calling `Ptree::GenSym()`. Then `TranslateMemberCall()` looks up the type of the called member function and produces the code for converting function arguments to a network message. The produced code is finally connected with other code and returned as the result of the translation.

6.3 Wrapper Library

Since a similar kind of abstraction often requires similar code translation, programmers may write a library to help meta-level programming for the similar code translation. Such a library should be called *a metaclass library*. In this section, we present an example of metaclass libraries.

A wrapper function is a useful technique for implementing abstractions such as concurrent objects. It is a function that wraps another function in itself and, if invoked, simply calls the wrapped function. But it may also performs some computation before or after calling it. For example:

```
int f(int i) { return i + 1; }

int wrap_f(int i) {
    cout << "f() is called.\n"
    return f(i);
}
```

Here, `wrap_f()` is a wrapper function for `f()`. It prints a message and then calls the wrapped function `f()`.

A number of abstractions can be implemented by metaclasses that produce wrappers for all member functions of a class. Suppose that a metaclass `MyWrapperClass` does such a thing. This metaclass translates the user program shown below:

```
metaclass Point : MyWrapperClass;
class Point {
public:
    void Move(int, int);
    void rMove(int, int);
    int x, y;
};

void Point::Move(int new_x, int new_y) {
    x = new_x; y = new_y;
}

void Point::Move(int diff_x, int diff_y) {
    x += diff_x; y += diff_y;
}

void f(Point& p)
{
    p.Move(3, 5);    // call Move()
    p.rMove(-1, 2);  // call rMove()
}
```

into the following program including wrapper functions for `Move()` and `rMove()`:


```
class Point {
public:
    void Move(int, int);
    void rMove(int, int);
    int x, y;
public:
    void wrapper_Move(int, int);
    void wrapper_rMove(int, int);
};

void Point::Move(int new_x, int new_y) { ... }

// inserted wrapper function for Move()
void Point::wrapper_Move(int p1, int p2)
{
    cout << "Move() is called.\n";
    Move(p1, p2);
}

void Point::rMove(int diff_x, int diff_y) { ... }

// inserted wrapper function for rMove()
void Point::wrapper_rMove(int p1, int p2)
{
    cout << "rMove() is called.\n";
    rMove(p1, p2);
}

void f(Point& p)
{
    p.wrapper_Move(3, 5);    // call the wrapper
    p.wrapper_rMove(-1, 2); // call the wrapper
}
```

Note that all occurrences of member calls for `Point` objects are substituted by the call of the wrapper function. For example, the call of `Move()` in `f()` is substituted by the call of the wrapper function.

This kind of wrapper metaclass is found in implementations of many abstractions. The translation by those wrapper metaclasses are quite similar and the only difference is what the produced wrapper functions perform before or after calling the wrapped functions. For example, the wrappers shown above just print a message, but, if they instead perform synchronization before calling the wrapped function, then `Point` objects will be concurrent objects.

A meta-level program using a metaclass library

Since wrapper metaclasses like `MyWrapperClass` are quite similar to each other, we should write a base class of these metaclasses and provide it as a metaclass library. Library developers can make their wrapper classes inherit

from the base class and focus on what the wrapper functions perform before or after calling the wrapped function.

Let the name of the base class be `WrapperClass`. With the metaclass library, `MyWrapperClass` should be defined by the following simple code:

```
class MyWrapperClass : public WrapperClass {
public:
    MyWrapperClass(Ptree* d, Ptree* m) : WrapperClass(d, m){}
    Ptree* WrapperBody(Environment*, Ptree*, Ptree*, int,
                       TypeInfo&);
};

Ptree* WrapperBody(Environment* e, Ptree* name, Ptree* wrapper,
                   int nargs, TypeInfo& ftype)
{
    Ptree* body = Class::WrapperBody(e, name, wrapper, nargs,
                                     ftype);
    return Ptree::qMake(
              "cout << \"`name`() is called.\n\";`body`");
}
```

Note that the metaclass `MyWrapperClass` inherits from `WrapperClass` and overrides only a member function `WrapperBody()`, which produces the body of a wrapper function. The base class performs the rest of the translation, which is to insert the declarations of wrapper functions, to replace all occurrences of member calls with calls of the wrapper functions, and so forth.

A metaclass library

The metaclass `WrapperClass` provided by the metaclass library needs to execute three things: (1) to insert wrapper functions in the user program, (2) to substitute calls of wrapper functions for calls of the wrapped functions, and (3) to provide a member function that a subclass of `WrapperClass` can override for specifying the behavior of wrapper functions. We below present overviews of how the three things are implemented. The complete implementation of `WrapperClass` is found in Appendix C.2.

For (1), `WrapperClass` overrides two member functions. One is `TranslateBody()`, which controls the translation of a class definition. It is overridden to insert the declarations of wrapper functions in the class definition. For example, the definition of the class `Point` is translated into this code:

```
class Point {
public:
    void Move(int, int);
    int x, y;
public:
    void wrapper_Move(int, int);
};
```


The underlined code is the inserted declaration. To do this translation, `TranslateBody()` examines each member of the class and, if the member is a function, it inserts the declaration of the wrapper function for that member:

```
Ptree* WrapperClass::TranslateBody(Environment* env, Ptree* body)
{
    Ptree* decl = Ptree::qMake("public:\n");
    Ptree* name;
    TypeInfo t;
    int i = 0;
    while((name = NthMemberName(i++)) != nil){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId
           && whatis != PtreeDestructorId
           && LookupMemberType(env, name, t)
           && t.WhatIs() == FunctionType){
               Ptree* m = t.MakePtree(WrapperName(name));
               decl = Ptree::qMake("`decl` `m`;\n");
        }
    }

    return Ptree::Append(body, decl);
}
```

Here, the member function `WrapperName()` a member function of the class `WrapperClass`. It returns the name of the wrapper function for the given member function.

The other member function overridden by `WrapperClass` is `TranslateMemberFunctionBody()`, which translates the body of a member function. It is overridden to produce the definitions of wrapper functions. For example, `TranslateMemberFunctionBody()` processes the definition of a member function `Move()` and inserts the definition of the wrapper function `wrapper_Move()` right after `Move()`:

```
void Point::Move(int new_x, int new_y)
{
    x = new_x; y = new_y;
}

void Point::wrapper_Move(int p1, int p2)
{
    cout << "Move() is called.\n";
    Move(p1, p2);
}
```

`TranslateMemberFunctionBody()` constructs the definition of `wrapper_Move()` from the `TypeInfo` metaobject for the type of `Move()`. First, it derives argument types and the return type from that `TypeInfo` metaobject and converts those types to `Ptree` metaobjects by calling `MakePtree()`.

Then it assembles the converted Ptree metaobjects with the wrapper name and the body of the wrapper function, and constructs the complete definition of wrapper_Move(). The overall structure of TranslateMemberFunctionBody() is as follows:

```
Ptree* WrapperClass::TranslateMemberFunctionBody(...)
{
      :
  Ptree* arglist = argument list of the wrapper function
      :
  Ptree* body = WrapperBody(env, name, wrapper_name, i - 1, t);
  Ptree* head = Ptree::qMake(
                        "`Name()`::`wrapper_name`(`arglist`");
      :
  AppendAfterToplevel(Ptree::Make("`head`{`body`}\n"));
  return Class::TranslateMemberFunctionBody(...);
}
```

The produced definition of wrapper_Move() is inserted by calling AppendAfterToplevel(). Note that TranslateMemberFunctionBody() calls WrapperBody() to make a function body so that (3) a subclass of WrapperClass can override it and specify the behavior of wrapper functions. The default WrapperBody() supplied by WrapperClass returns an expression that just calls the wrapped function.

Finally, we show TranslateMemberCall(), which WrapperClass overrides for (2). It substitutes calls of wrapper functions for calls of the wrapped functions. To do this substitution, it just calls TranslateMemberCall() supplied by Class with the name of a wrapper function:

```
Ptree* WrapperClass::TranslateMemberCall(Environment* env,
                                         Ptree* member, Ptree* arglist)
{
    return Class::TranslateMemberCall(env, WrapperName(member),
                                      arglist);
}
```

Note that the second argument to TranslateMemberCall() supplied by Class is not the name of the wrapped function, that is, member, but the name of the wrapper function. Thus a member function call for a wrapper-class object:

```
p.Move(3, 5)
```

is translated into this expression:

```
p.wrapper_Move(3, 5)
```


6.4 Implementation of qMake()

The OpenC++ MOP provides a member function Ptree::qMake() (quoted make), which constructs a Ptree metaobject according to the given format. Although this member function is more convenient than a similar member function Make(), it cannot be implemented within regular C++, but requires meta-level programming to be implemented. Without meta-level programming, only Make() is available.

The implementation of qMake() is an example of meta-meta level programming in OpenC++. The OpenC++ MOP uses itself to implement metaobjects and their member functions such as qMake(), so that the OpenC++ MOP provides better abstractions and programming interface for programmers to write metaclasses. Through this example, we present that OpenC++ can naturally deal with meta-meta level programming and it actually uses meta-meta level programming for implementing abstractions that make it easier to write metaclasses.

What the user program should look like

Recall the usage of qMake(). If a variables tmp is a pointer to a Ptree metaobject representing a variable name xyz, and a variable i is an integer 3, then programmers may write something like this:

```
Ptree* exp = Ptree::qMake("int `tmp` = `i`;");
```

This program constructs a Ptree metaobject "int xyz = 3". The expressions surrounded by back-quotes are expanded when qMake() is invoked.

The program shown above can be rewritten into a program using Make():

```
Ptree* exp = Ptree::Make("int %p = %d", tmp,  i);
```

Unlike qMake(), Make() takes a format and some parameters, which are substituted for the occurrences of %p and %d in the format. This kind of programming interface is popular in C and C++, but it becomes error-prone as the number of parameters increases. Typical errors caused by this interface are to give a wrong number of parameters and to place parameters in a wrong order.

What the user program should be translated into

Function calls of qMake() are translated into a combination of several function calls. For example, this program:

```
Ptree* exp = Ptree::qMake("int `tmp` = `i`;");
```

is translated into:

```
Ptree* exp = (Ptree*)(PtreeHead()+"int "+tmp+" = "+i+";");
```

Note that the program after the translation is in regular C++. For example, the character string "tmp" becomes a variable name tmp after the translation. Since regular C++ cannot convert a character string to a variable name at runtime, the conversion should be done by the translation at compile time.

PtreeHead() returns a PtreeHead object, which is a stream object producing a Ptree metaobject. The + operator is used to input a character string, a Ptree metaobject, and so on, to the stream object. The inputted data are concatenated into a Ptree metaobject, and the concatenated Ptree metaobject is obtained by explicitly casting the PtreeHead object into the type Ptree*. The cast operator is overloaded to return the concatenated Ptree metaobject.

Write a meta-level program

Writing a metaclass for performing the translation mentioned above is quite straightforward. To do this translation, the metaclass QuoteClass just overrides a member function TranslateMemberCall():

```
Ptree* QuoteClass::TranslateMemberCall(Environment* env,
                                       Ptree* member, Ptree* args)
{
    Ptree* name = SimpleName(member);
    char* str;

    if(Ptree::Eq(name, "qMake")){
        Ptree* arg1 = Ptree::First(Ptree::Second(args));
        if(arg1->Reify(str) && str != nil)
            return ProcessBackQuote(FALSE, str);
        else
            ErrorMessage("bad argument for qMake()", arg1);
    }
    else
        return Class::TranslateMemberCall(env, member, args);
}
```

This member function translates a given function-call expression if the called function is qMake(), otherwise it delegates the translation to the member function supplied by the base class Class.

Since the argument member may be not a simple member name but a qualified name such as Ptree::qMake(), TranslateMemberCall() first calls SimpleName() to strip the class name and double colons off:

```
Ptree* QuoteClass::SimpleName(Ptree* qualified_name)
{
    if(qualified_name->IsLeaf())
        return qualified_name;
    else
        return Ptree::First(Ptree::Last(qualified_name));
}
```


After that, TranslateMemberCall() determines whether the member name is qMake, and if so, it converts the first argument from a Ptree metaobject to a character string. This conversion is done by calling Reify() for arg1. Then TranslateMemberCall() calls ProcessBackQuote() with the converted character string to translate the member-call expression. ProcessBackQuote() is a member function of QuoteClass.

6.5 Metaclass

Only the implementation of qMake() is not an example of use of meta-meta level programming. The default metaclass Class and its subclasses are also implemented with using meta-meta level programming, so that programmers can easily define a new metaclass. For this reason, the metaclass Class and its subclasses are also class metaobjects, and they are instances of the metaclass Metaclass. Figure 6.1 shows this instance-of relationship.

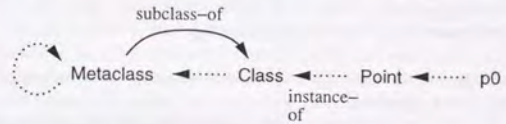

Figure 6.1: The instance-of relationship among metaclasses

Protocol without meta-meta level programming

The definitions of metaclasses seen so far have not explicitly showed all the protocol that the metaclasses must obey. They need to be translated by the metaclass Metaclass so that they satisfy all the protocol.

To obey all the protocol, (i) a new metaclass has to have a member function MetaclassName(), and (ii) a function that instantiates the metaclass must be registered. For example, a new metaclass MyClass should be translated into something more complex than what we have seen:

```
class MyClass : public Class {
public:
    MyClass(Ptree* d, Ptree* m) : Class(d, m) {}
        :
    char* MetaclassName() { return "MyClass"; }
```

```
};

static Class* CreateMyClass(Ptree* d, Ptree* m)
{
    return new MyClass(d, m);
}

static ListOfMetaclass myClassObject("MyClass", CreateMyClass,
                                     MyClass::Initialize());
```

After the translation, a member function `MetaclassName()`, a function `CreateMyClass()`, and an object `myClassObject` are inserted. `CreateMyClass()` is a function that instantiates the metaclass, and the object `myClassObject` is created at the beginning of runtime and registers `CreateMyClass()`. The registered function is used to implement a function that receives a class name in the form of character string and instantiates the specified class. This function is internally used by the OpenC++ compiler when instantiating a metaclass, because the `new` operator does not take a character string to specify the instantiated class:

```
class Point { ... };
Point* p0 = new Point;          // valid
char* c = "Point";
Point* p1 = new c;              // invalid
```

Note that `new Point` is a valid expression but `new c` is not since `c` is not a class name but a variable.

The behavior implemented by the inserted functions and variable cannot be inherited from the base class; it must be explicitly implemented by every new metaclass. This is because implementing the behavior needs the definition of the new metaclass as the example of named object that we have shown in Chapter 2.

Protocol with meta-meta level programming

Since the definition obeying all the protocol is complex and error prone, the actual OpenC++ MOP has meta-level programmers write a simpler definition of a metaclass, and automatically translates it into the regular definition that obeys all the protocol. To do this, the OpenC++ MOP provides a metaclass `Metaclass`, which is the metaclass of all metaclasses.

Like other metaclasses, the metaclass `Metaclass` is a subclass of `Class`. It overrides `TranslateSelf()` and `TranslateBody()`:

```
Ptree* Metaclass::TranslateSelf(Environment* env)
{
    Ptree* name = Name();
    Ptree* tmpname = Ptree::GenSym();
```



```
    AppendAfterToplevel(Ptree::Make(
        "static Class* %p(Ptree* def, Ptree* marg){\n"
        "    return new %p(def, marg); }\n"
        "static ListOfMetaclass %p(\"%p\", %p,\n"
        "                          %p::Initialize());\n",
        tmpname, name,
        Ptree::GenSym(), name, tmpname, name));

    return Class::TranslateSelf(env);
}

Ptree* Metaclass::TranslateBody(Environment* env, Ptree* body)
{
    Ptree* mem = Ptree::Make("public: char* MetaclassName() {\n"
                             "    return \"%p\"; }",
                             Name());
    return Class::TranslateBody(env, Ptree::Append(body, mem));
}
```

The member function `TranslateSelf()` inserts a function and a variable for instantiating the metaclass, and `TranslateBody()` inserts a member function `MetaclassName()` in the declaration of the metaclass. For the reason of bootstrapping, `Metaclass` uses not `Ptree::qMake()` but `Ptree::Make()`.

Since a subclass inherits the metaclass from the base class, programmers do not need to explicitly write a metaclass declaration for new metaclasses. The OpenC++ compiler automatically selects `Metaclass` for the new metaclasses since they are subclasses of `Class`. No ad-hoc implementation is required; the OpenC++ MOP can naturally deal with this and programmers can enjoy a simpler protocol by this mechanism.

6.6 Vector Library

All the examples shown above are of abstractions that regular C++ cannot handle but the OpenC++ MOP can do. The OpenC++ MOP also makes some kinds of abstractions more efficient than in regular C++. From this section, we show a few examples of such abstractions.

The first example is the vector library. In Chapter 2, we showed that the template mechanism of C++ enables a vector abstraction for any type, but the implementation with the template mechanism was not as efficient as an ideal implementation. If the OpenC++ MOP is used, however, the vector abstraction is implemented more efficiently.

Vector library in regular C++

As we showed in Chapter 2, a vector abstraction is implemented in regular C++ by the following template:

```
template <class T> class Vector {
```

```
    T elements[SIZE];
public:
    Vector operator + (Vector& a, Vector& b) {
        Vector c;
        for(i = 0; i < SIZE; ++i)
            c.elements[i] = a.elements[i] + b.elements[i];

        return c;
    }
        :
};
```

This implementation is not efficient because it deals with successive operators like `v2 + v3 + v4` as separate function calls. For each operator, the operator function for `Vector` is called and the `for` loop is executed from `0` to `SIZE`, although successive `for` loops can be fused into a single efficient loop.

This problem is due to the limited ability of the template mechanism to supply code adapted for the library user code. The only adaptation that the template mechanism can do is type parameterization, and therefore, the vector abstraction needs to be implemented with very generic description — overloading primitive operators like `+`. No implementation techniques for a particular case can be included in the description.

Vector library in OpenC++

If the OpenC++ MOP is used, the vector abstraction can be implemented more efficiently. The library developer can define a special metaclass for the template class `Vector`, which translates successive operators into an efficient single loop instead of separate function calls. Doing this translation is straightforward; only one member function `TranslateAssign()` needs to be overridden:

```
#include "template.h"

class VectorClass : public TemplateClass {
public:
    VectorClass(Ptree* d, Ptree* m) : TemplateClass(d, m) {}
    Ptree* TranslateAssign(Environment*, Ptree*, Ptree*,
                           Ptree*);
    Ptree* Inline(Environment*, Ptree*, Ptree*);
};

Ptree* VectorClass::TranslateAssign(Environment* env,
                                    Ptree* object,
                                    Ptree* op, Ptree* expr)
{
    if(!object->IsLeaf() || !op->Eq('=')
       || expr->IsLeaf())        // e.g. a = b;
```



```
        return TemplateClass::TranslateAssign(env, object,
                                              op, expr);
    Ptree* index = Ptree::GenSym();
    return Ptree::qMakeStatement(
        "for(int `index` = 0;`index` < SIZE; ++`index`)\n"
        "  `object`.element[`index`]"
        "  =`Inline(env, expr, index)`;\n");
}
```

The member function `TranslateAssign()` translates an assignment expression such as `=` and `+=`. It translates an assignment expression into an efficient `for` loop if the operator is `=`. The right-side expression is passed to a member function `Inline()` and translated into an appropriate expression:

```
Ptree* VectorClass::Inline(Environment* env, Ptree* expr,
                           Ptree* index)
{
    Ptree* lexpr;
    Ptree* rexpr;

    if(expr->IsLeaf())
        return Ptree::qMake("`expr`.element[`index`]");
    else if(Ptree::Match(expr, "[%? + %?]", &lexpr, &rexpr))
        return Ptree::qMake("`Inline(env, lexpr, index)`"
                            "+ `Inline(env, rexpr, index)`");
    else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
        return Ptree::qMake("`Inline(env, lexpr, index)`"
                            "- `Inline(env, rexpr, index)`");
    else if(Ptree::Match(expr, "[( %? )]", &lexpr))
        return Ptree::qMake("(`Inline(env, lexpr, index)`)");
    else if(Ptree::Match(expr, "[- %?]", &lexpr))
        return Ptree::qMake("-`Inline(env, lexpr, index)`");
    else if(Ptree::Match(expr, "[%? * %?]", &lexpr, &rexpr))
        return Ptree::qMake(
                    "`lexpr` * `Inline(env, rexpr, index)`");
    else{
        ErrorMessage("invalid vector expression", expr);
        return nil;
    }
}
```

The member function `Inline()` tests whether the given expression matches a pattern and, if it matches, calls `Inline()` recursively to process the subexpressions. `Inline()` can deal with not only the `+` operator but also the `-` and `*` operators and parentheses `()`.

Experiments

The metaclass `VectorClass` translates an assignment expression on a vector into an efficient `for` loop. For example, this program:

Table 6.1: Execution performance of the vector library (μsec.)

	length 8				length 64			
# of vectors	1	2	3	4	1	2	3	4
Sun C++	0.5	1.5	3.0	4.4	3.3	10.1	20.0	30.1
GNU C++	0.3	1.7	3.1	4.5	6.9	21.7	36.2	51.1
OpenC++	0.3	0.9	1.3	1.7	6.9	6.5	9.8	13.1
Hand-coded	0.9	0.9	1.3	1.7	6.5	6.5	9.8	13.0

Average of 1,000,000 (size 8) or 300,000 (size 64) iterations.
SPARC Station 20/61, SunOS 4.1.3

```
Vector<double> v1, v2, v3, v4;
    :
v1 = v2 + v3 + v4;
```

is translated into this:

```
Vector<double> v1, v2, v3, v4;
    :
for(i = 0; i < SIZE; ++i)
    v1.elements[i] = v2.elements[i] + v3.elements[i]
                   + v4.element[i];
```

This translation drastically improves execution performance. To show the improvement, we ran a micro benchmark program and measured execution time of `Vector` expressions. The micro benchmark program computes the sum of various numbers of vectors of `double`. First, we ran the program with the vector library written in regular C++, then ran the same program with the vector library written in OpenC++. Since different compilers perform different optimization techniques, we used GNU C++ 2.7.2.1 (with option -O3) and Sun C++ 3.0.1 (with option -fast) for regular C++. The OpenC++ compiler uses GNU C++ for the backend compiler. Moreover, we ran a program that is equivalent to the micro benchmark but optimized by hand without the vector library. This hand-coded program uses arrays of `double` instead of objects and was compiled by GNU C++. When measuring the execution time, we changed the length of each vector between 8 and 64, and also the number of the vectors in the assigned expression from 1 to 4. When the number of the vectors is 1, the expression is `v0 = v1;` No addition is executed. All the benchmark programs used in this experiment are in Appendix C.3.

The results of the experiment are listed in Table 6.1. The vector library implemented with the OpenC++ MOP achieved as good performance as the hand-coded program. Although the OpenC++ program is slightly slower than the hand-coded program when the number of vectors is 1 and the vector length is 64, this fact is caused by the implementation difference in copying a vector. The hand-coded program copies a vector by explicitly copying each element, but the OpenC++ program copies it by using the default object copy mechanism, which is compiled into a call of `memcpy()`. The implementation strategy of copying a vector is also the reason that Sun C++ achieved the best performance when the number of vectors is 1 and the length of each vector is 64. The Sun C++ compiler inlines `memcpy()` when an object is copied.

This experiment also shows that real C++ compilers do not perform the optimization that the metaclass `VectorClass` performs. Although an ideal compiler should automatically perform the optimization, it seems difficult for real compilers to do that within reasonable space and time. Our claim is that such an optimization should be done by a metaclass rather than a compiler's optimizer. Such an optimization is difficult for a general-purpose optimizer as our experiment showed, but on the other hand, it is not difficult for a metaclass because the metaclass is written by the library developer, who knows semantic information about the library code. We believe that a compiler's optimizer should focus on general optimizations and metaclasses should perform a special optimization that is effective only for a particular class. Although forcing end-programmers to write a metaclass is not realistic, we believe that it is acceptable that library programmers write metaclasses since they are usually experienced programmers and their code is reused by many end programmers.

6.7 The Standard Template Library

The Standard Template Library (STL) [44] is another example of abstractions that The OpenC++ MOP can implement more efficiently. STL is a C++ library included by the ANSI standard of C++. Despite of the high-level abstractions that STL provides, a program using STL is often slower than an equivalent program written without STL. The OpenC++ MOP contributes to avoid performance drawbacks caused by STL with keeping its high-level abstractions.

Brief Overview of STL

A unique feature of STL is that the library consists of independent components and the users can flexibly combine the components to obtain the functionality they need. The main components of STL are containers and generic algorithms. The containers are objects that store a collection of

other objects, and the generic algorithms are functions that process containers. Since type names are parameterized with using the template mechanism, STL users may use a generic algorithm with any kind of containers. They do not have to use a different version of the generic algorithm for a different kind of containers.

For example, STL users may write something like this:

```
list<double> a1 = ... ;
set<double> a2 = ... ;
n1 = count(a1.begin(), a1.end(), 3.14);
n2 = count(a2.begin(), a2.end(), 3.14);
```

This program computes the number of 3.14 stored in containers a1 and a2, respectively. a1 is a list container and a2 is a set container. count is a generic algorithm to determine the number of elements in a container that are equal to a given value. It takes pointers to the first and the last element in the container and the value that it counts the number of. Note that the same generic algorithm count is used for two different kinds of containers list and set. The single generic algorithm serves all kinds of containers.

The connectivity between containers and generic algorithms is enabled by another kind of STL component called iterators. In the program above, iterators are the values returned by a1.begin() and a1.end(). They are pointer-like components that all kinds of containers provide as common interface to access the elements. Generic algorithms use the iterators to traverse elements stored in a container. For example, iterators for list containers are defined as follows:

```
class iterator {
public:
    iterator(list<T>* p) { ptr = p; }
    list<T>* ptr;
    int eof() { return ptr == 0; }
    int operator != (iterator& a) { return ptr != a.ptr; }
    T operator * () { return ptr->value; }
    iterator& operator ++ (){ptr = ptr->next; return *this;}
    iterator operator ++ (int) {
        iterator prev = *this;
        ptr = ptr->next;
        return prev;
    }
};
```

Iterators are objects for which pointer operators such as * and ++ are overloaded. Generic algorithms use the iterators as if they are C++ pointers to arrays; for example, the next template function is an implementation of the generic count algorithm:

```
template <class I, class T>
int count(I first, I last, T value)
```



```
{
    int n = 0;
    while(first != last)
        if(*first++ == value)
            ++n;

    return n;
}
```

The template argument I is the type of iterators and T is the type of container elements. Note that the variables first and last are used as if they are pointers to an array of the type T.

Performance improvement by the OpenC++ MOP

Although iterators give great flexibility to STL, they also involve performance drawbacks if compared with an equivalent program written without iterators. Since a generic algorithm must indirectly accesses elements in a container through an iterator, its execution performance tends to be slower. If programmers give up generality of the generic algorithm and specialize the algorithm to work only for a particular kind of container, then the specialized algorithm will be more efficient because iterators are not needed any more. For example, the following function is a specialized count algorithm for counting elements only in a list<int> container:

```
int count(List<int>* first, List<int>* last, int value)
{
    int n = 0;
    while(first != last){
        if(first->value == value)
            ++n;

        first = first->next;
    }

    return n;
}
```

Note that now the variables first and last are not iterators but actual pointers to list<int> objects. Hence reading an element is done by the -> operator instead of the * operator. Also, the ++ operator is replaced with the expression first = first->next.

The OpenC++ MOP reduces the overheads by iterators. In OpenC++, the STL implementor can write a metaclass that specializes a generic algorithm for a particular kind of container and translates a program to use that specialized algorithm. Suppose that a program uses the generic count algorithm with a list<int> container. The metaclass converts iterators for list<int> into actual pointers to list<int> and it replaces calls of

the generic count algorithm with calls of a count algorithm specialized for list<int>. First, the definition of iterators for list containers:

```
class iterator { ... };
```

is translated from a class type into this pointer type:

```
typedef list<T>* iterator;
```

Then the template function shown below is derived as a specialized algorithm from the generic count algorithm. It is substituted for the generic count algorithm called with list containers:

```
template <class I, class T>
int count_for_list_int(I first, I last, T value)
{
    list<int>* tmp;
    int n = 0;
    while(first != last)
        if((tmp=first,first=first->next,tmp)->value == value)
            ++n;

    return n;
}
```

This template function supposes that I is bound to list<int>*.

The translation mentioned above is easily implemented with about 40 lines of meta-level program. The metaclass for this translation is effective not only for the generic count algorithm. It deals with combinations of list containers and any generic algorithm. For the complete definition of the metaclass, see in Appendix C.4. We also show a metaclass for set containers in the same place.

Experiment

To illustrate the performance improvement by the metaclasses, we measured the execution time of the generic count algorithm with/without the metaclasses. As for the containers, we used list<int> and set<int>. When the metaclasses were not used, the measured program was compiled by GNU C++ 2.7.2 with option -O3 and Sun C++ 4.1 with option -fast. When the metaclasses are used, the program was compiled by the OpenC++ compiler, which uses GNU C++ for the backend compiler. Also, we measured the execution time of hand-optimized count algorithms for list<int> and set<int>. These algorithms were compiled by GNU C++. All the programs used in this experiment are shown in Appendix C.4.

The results of this experiment is listed in Table 6.2. As for list containers, the generic count algorithm involves 73% (Sun C++) or 36% (GNU C++) overheads against the hand-optimized version, but with the OpenC++ MOP, the overheads are reduced to only 9%. As for set containers, the generic count algorithm involves 39% (Sun C++) or 21% (GNU C++) overheads. But the OpenC++ MOP reduces the overheads to 10%. These results show that generic algorithms of STL work for any kind of containers but this adaptability causes serious performance degradation. The OpenC++ MOP recovers this performance degradation from one half to one fourth while keeping the adaptability of STL.

Table 6.2: Execution performance of STL (msec.)

	list (ratio)	set (ratio)
Sun C++	57 (1.73)	860 (1.39)
GNU C++	45 (1.36)	750 (1.21)
OpenC++	36 (1.09)	680 (1.10)
Hand-coded	33 (1.00)	620 (1.00)

Average of 100 (list) or 10 (set) iterations.
SPARC Station 20/514, Solaris 5.3

6.8 The OOPACK Benchmark Test

The last example is the OOPACK benchmark test [49]. This benchmark test is a program for testing the ability of a C++ compiler for compiling a program written in object-oriented programming (OOP) as efficiently as a program in non-OOP. The program contains a suite of tests, each of which consists of two equivalent routines written in OOP and non-OOP. The OOP routines are written with higher-level abstractions, whereas the non-OOP routines are written in C style for efficiency. In other words, the non-OOP routines are hand-optimized versions of the corresponding OOP routines.

If the OOP routines are slower than the non-OOP routines, that performance degradation means costs to use higher-level abstractions implemented with objects under the compiler. A C++ compiler should be able to compile the OOP routines as efficient as the non-OOP routines since it can in principle transform the OOP routines into the non-OOP routines by inlining member functions and performing constant propagation and strength reduction.

Improvement with the OpenC++ MOP

Real C++ compilers have difficulty in compiling the OOP routines efficiently as we show later, but this inefficiency is fairly recovered by the OpenC++ MOP. Although the OpenC++ MOP cannot improve the execution speed without changing the benchmark program, it can extend the language syntax to allow the programmer to write more efficient OOP code than the OOP routines in the original benchmark program. The extended syntax is used for annotating compilation hints without directly describing lower-level details of the implementation. It does not affect the level of abstraction of the OOP routines.

For example, the following code is the `Matrix` test in the benchmark. It computes multiplication of two matrices in OOP style:

```
void MatrixBenchmark::oop_style() const
{
    Matrix c(L, L, C);
    Matrix d(L, L, D);
    Matrix e(L, L, E);
    for(int i = 0; i < e.rows; i++)
        for(int j = 0; j < e.cols; j++){
            double sum = 0;
            for(int k=0; k<e.cols; k++)
                sum += c(i,k) * d(k,j);

            e(i,j) = sum;
        }
} .
```

The OpenC++ MOP extends the syntax to make a `foreach` statement available for this test and improve the execution performance with the new statement. The next is the code rewritten using the `foreach` statement:

```
void MatrixBenchmark::oop_style() const
{
    Matrix c(L, L, C);
    Matrix d(L, L, D);
    Matrix e(L, L, E);

    c.foreach(i){
        for(int j = 0; j < e.cols; ++j){
            double sum = 0;
            d.foreach(k){
                sum += c(k) * d(j);
            };

            e(i,j) = sum;
        }
    };
}
```


Here, `c.foreach(i)` means iterating the following block statement for each row of the matrix `c`. `i` is bound to the index of the row currently processed. In the block statement, `c(k)` indicates the `k`-th column in the row. Unlike the `for` statement, the `foreach` statement explicitly indicates that the loop is executed for traversing the rows of a matrix, so that the statement can be translated into optimized code for the traversing. In fact, the metaclass for `Matrix` translates the program above into this efficient one:

```
void MatrixBenchmark::oop_style() const
{
    Matrix c(L, L, C);
    Matrix d(L, L, D);
    Matrix e(L, L, E);

    for(int i=c.rows, t1=c.cols, t2=(c.rows-1)*c.cols;
        --i >= 0;
        t2 -= t1)
    {
        double const* t3 = &(c.Data())[t2];
        for(int j = 0; j < e.cols; ++j){
            double sum = 0;
            for(int k=d.rows, t4=d.cols, t5=(d.rows-1)*d.cols;
                --k >= 0;
                t5 -= t4)
            {
                double const* t6 = &(d.Data())[t5];
                sum += t3[k] * t6[j];
            };

            e(i,j) = sum;
        }
    };
}
```

Note that the `foreach` statements are translated into `for` statements.

The second test in the benchmark program is `Iterator`. It computes dot-product of two vectors implemented by arrays of `double`. The `Iterator` objects in this test is similar to STL's iterator, but they also contain the length of a vector and provide a member function `done()` to check whether there are no more elements:

```
void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    for(Iterator ai(A,N), bi(B,N);
        !ai.done();
        ai.next(), bi.next())
    {
        sum += ai.look() * bi.look();
    }
```

```
    IteratorResult = sum;
}
```

Here, A and B are arrays of double, and N is the length of the arrays.

The OpenC++ MOP provides a statement foreach for the class Iterator. With the foreach statement, the program above is rewritten into this:

```
void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    Iterator ai(A,N), bi(B,N);
    ai.foreach(v){
        sum += v * bi.look();
        bi.next();
    };

    IteratorResult = sum;
}
```

The foreach statement for Iterator provides a control abstraction similar to foreach for Matrix. It iterates the following block statement for each element of the vector. In the block statement, a specified variable (v in the program above) indicates the element currently processed. However, the foreach statement for Iterator is translated differently way from Matrix. This is oop_style() after the translation:

```
void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    Iterator ai(A,N), bi(B,N);
    for(int t7 = 0, t8 = ai.Limit(); t7 < t8; ++t7){
        const double& v = ai.Array(t7);
        sum += v * bi.look();
        bi.next();
    };

    IteratorResult = sum;
}
```

The length of the vector that ai points to is stored in a local variable before starting iteration. This eliminates accesses to ai when the loop-termination condition is checked.

The last test in the benchmark is Complex. This test computes a complex-valued "SAXPY" operation, and measures how efficiently a C++ compiler compiles a Complex object, which represents a complex number and is very popular in scientific computing:

```
void ComplexBenchmark::oop_style() const
{
    Complex factor(0.5, 0.86602540378443864676);
```



```
    for(int k = 0; k < N; k++)
        Y[k] = Y[k] + factor * X[k];
}
```

Here, X and Y are arrays of Complex objects.

The execution performance of this test can be improved by a similar technique that we showed for the vector library in Section 6.6. No extended syntax is needed. The program shown above is translated by a metaclass into this:

```
void ComplexBenchmark::oop_style() const
{
    double factor_re = 0.5;
    double factor_im = 0.86602540378443864676;
    for(int k = 0; k < N; k++){
        Y[k].re = Y[k].re+factor_re*X[k].re-factor_im*X[k].im;
        Y[k].im = Y[k].im+factor_re*X[k].im+factor_im*X[k].re;
    }
}
```

Note that a Complex object factor is broken down into two double variables, factor_re and factor_im. This leads a C++ compiler to allocate factor on registers rather than a stack frame, and eventually contributes to performance improvement. Without this translation, C++ compilers such as GNU C++ and Sun C++ do not allocate objects on registers even though allocating them is possible in principle.

Experiments

We measured execution time of the OOPACK benchmark test under different settings. We first ran the benchmark program compiled by Sun C++ 4.1 with -fast option, and then the program compiled by GNU C++ 2.7.2 with -O3 option. We also compiled the program by the OpenC++ compiler with metaclasses for the translation shown above, and measured the execution time of the compiled code. When compiling with those metaclasses, the benchmark program was rewritten to utilize the extended syntax. The OpenC++ compiler used GNU C++ 2.7.2 with option -O3 for the back-end compiler. All the programs used for the experiment are presented in Appendix C.5.

The results of the experiment are listed in Table 6.3. The OOPACK benchmark consists of four tests: Max, Matrix, Complex, and Iterator. We did not develop a metaclass for the Max test, which is for measuring how well a C++ compiler inlines a function. But for the remaining three tests, OpenC++ showed better performance than Sun C++ and GNU C++. If OpenC++ is used, the OOP routines involves only 10% to 20% overheads against the non-OOP routines written in C style. On the other hand, the OOP routines compiled by Sun C++ is twice or three times slower than the

Table 6.3: Execution time of the OOPACK benchmark (sec.)

		Sun C++	GNU C++	OpenC++
Max	(C-style)	8.1	9.0	9.0
	(OOP)	8.5	12.0	12.0
	Ratio	1.1	1.3	1.3
Matrix	(C-style)	10.9	9.8	9.8
	(OOP)	31.3	80.7	11.2
	Ratio	2.9	8.2	1.1
Complex	(C-style)	13.2	11.2	11.2
	(OOP)	23.3	18.2	13.1
	Ratio	1.8	1.6	1.2
Iterator	(C-style)	7.1	7.1	7.1
	(OOP)	15.2	8.2	8.1
	Ratio	2.1	1.2	1.1

50000 (Max), 500 (Matrix), 20000 (Complex), 50000 (Iterator) iterations.
SPARC Station 20/514, Solaris 5.3

non-OOP routines. The OOP routines compiled by GNU C++ also involves 60% or 20% overheads for `Complex` and `Iterator`, but the `Matrix` test is more than 8 times slower.

6.9 Summary

This chapter presented eight examples of libraries that the OpenC++ MOP enables. The first two examples, named objects and distributed objects, presented useful data abstractions that regular C++ cannot handle. Through the examples, we also illustrated how various metaobjects like `TypeInfo` are used in meta-level programming.

The next example, wrapper library, is an example of metaclass libraries. Implementing abstractions like named objects and distributed objects is facilitated if there is a metaclass library which provides typical meta code for metaclass writers. This example showed a metaclass `WrapperClass`, which helps programmers write a metaclass handling wrapper functions.

`qMake()` and `Metaclass` are examples of meta-meta level programming. The OpenC++ MOP naturally allows meta-meta level programming in order to make useful abstractions available at meta level as well as base level. `qMake()` is a meta-level function providing convenient interface for meta-

class writers. Its interface cannot be implemented without the OpenC++ MOP. `Metaclass` is a meta-metaclass for all metaclasses. It simplifies the protocol for writing a new metaclass.

The remaining three examples showed that the OpenC++ MOP can be used to improve execution performance of some kinds of abstractions. The OpenC++ MOP makes it possible to specialize library code for user code and improve execution performance. This specialization includes elimination of unnecessary indirection and encapsulation and it helps C++ compilers to generate more efficient object code. Furthermore, the OpenC++ MOP allows syntax extension, which is used for putting annotations for efficient compilation. In the example of the OOPACK benchmark, we showed that such syntax extensions actually improve execution performance while keeping the level of abstraction.