Open C++ Programmer's Guide for Version 2*

Shigeru Chiba
Xerox PARC and University of Tokyo
chiba@parc.xerox.com chiba@is.s.u-tokyo.ac.jp

1 Introduction

The goal of the OpenC++ project is to make the C++ language extensible. The project started in 1992 at the University of Tokyo in Japan. The first version was released in 1993 as one of the early C++ MOPs (metaobject protocols) and has been used as a research platform at many sites, including University of Newcastle upon Tyne in UK, LAAS in Toulouse, France, Universidade Federal do RGS in Brazil, and so on. After that, the project moved to Xerox Palo Alto Research Center (PARC) in US and joined the Open Implementation Group in 1994. OpenC++ met compile-time MOPs there and OpenC++ Version 2 was developed as the result.

The OpenC++ language enables programmers to extend C++ so that they can use language features that are not available in regular C++. These language features include distribution, persistence, and fault-tolerance. Although they are available in regular C++ with less integrated syntax, for example, OpenC++ enables programmers to use the member function call syntax when accessing a remote object.

Moreover, OpenC++ makes it possible to customize an optimization scheme for a particular class. The OpenC++ compiler manipulates a program at the source code level for optimizing the execution performance. Programmers can customize that program manipulation on a particular class. This benefit got feasible in the version 2 because its MOP is a compile-time MOP.

*Originally appeared as Technical Report SPL-96-024, Xerox PARC, 1996.

This document shows detailed specifications of OpenC++ Version 2. It consists of three parts. First, we give a brief tutorial of programming in OpenC++. It will help the readers get overview of the language. Then, we mention the base-level specifications of OpenC++. The difference from regular C++ is shown here. Last, we present the meta-level specifications, that is, the OpenC++ MOP. It is an interface to deal with the base-level program and customize the language.

2 Tutorial

OpenC++ is an extensible language based on C++. The extended features of OpenC++ are specified by a meta-level program given at compile time. For distinction, programs written in OpenC++ are called base-level programs. If no meta-level program is given, OpenC++ is identical to regular C++.

The meta-level program extends OpenC++ through the interface called the OpenC++ MOP. The OpenC++ compiler consists of three stages: preprocessor, source-to-source translator from OpenC++ to C++, and the back-end C++ compiler. The OpenC++ MOP is an interface to control the translator at the second stage. It allows to specify how an extended feature of OpenC++ is translated into regular C++ code.

An extended feature of OpenC++ is supplied as an *add-on* software for the compiler. The add-on software consists of not only the meta-level program but also runtime support code. The runtime support code provides classes and functions used by the base-level program translated into C++. The base-level program in OpenC++ is first translated into C++ according to the meta-level program. Then it is linked with the runtime support code to be executable code. This flow is illustrated by Figure 1.

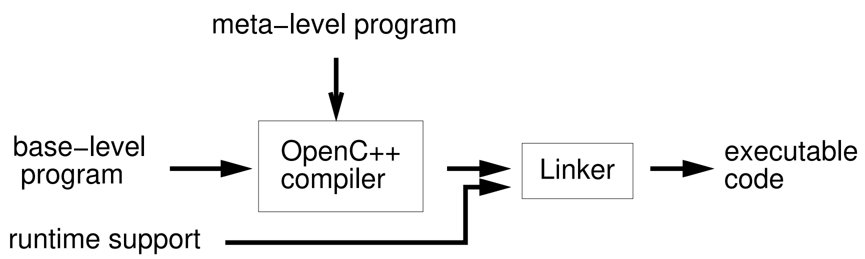

Figure 1: The OpenC++ Compiler

The meta-level program is written in C++, accurately in OpenC++ because OpenC++ is a self-reflective language. It defines new metaobjects to control source-to-source translation. The metaobjects are the meta-level representation of the base-level program and they perform the translation. Details of the metaobjects are specified by the OpenC++ MOP. In the followings, we go through a few examples so that we illustrate how the OpenC++ MOP is used to implement language extensions.

2.1 Verbose Objects

A MOP version of "hello world" is verbose objects, which print a message for every member function call. We choose them as our first example.

The MOP programming in OpenC++ is done through three steps: (1) decide what the base-level program should look like, (2) figure out what it should be translated into and what runtime support code is needed, and (3) write a meta-level program to perform the translation and also write the runtime support code. We implement the verbose objects through these steps.

2.1.1 What the base-level program should look like

In the verbose objects example, we want to keep the base-level program looking the same as much as possible. The only change should be to put an annotation that specifies which class of objects print a message for every member function call. Suppose that we want to make a class `Person` verbose. The base-level program should be something like:

```
// person.cc
#include <stdio.h>

metaclass Person : VerboseClass;     // metaclass declaration
class Person {
public:
    Person(int age);
    int Age() { return age; }
    int BirthdayComes() { return ++age; }
private:
    int age;
};

main()
{
    Person billy(24);
    printf("age %d\n", billy.Age());
```

```
    printf("age %d\n", billy.BirthdayComes());
}
```

Note that the metaclass declaration in the first line is the only difference from regular C++ code. It specifies that Person objects print a message for every member function call.

2.1.2 What the base-level program should be translated

In order to make the program above work as we expect, member function calls on Person objects must be appropriately translated to print a message. For example, the two expressions:

```
billy.Age()
billy.BirthdayComes()
```

must be translated respectively into:

```
(puts("Age()"), billy.Age())
(puts("BirthdayComes()"), billy.BirthdayComes())
```

Note that the value of the comma expression (x , y) is y. So the resulting values of the substituted expressions are the same as those of the original ones.

2.1.3 Write a meta-level program

Now, we write a meta-level program. What we should do is to translate only member function calls on Person objects in the way shown above. We can easily do that if we use the MOP.

In OpenC++, classes are objects as in Smalltalk. We call them class metaobjects when we refer to their meta-level representation. A unique feature of OpenC++ is that a class metaobject translates expressions involving the class at compile time. For example, the class metaobject for Person translates a member function call billy.Age() since billy is a Person object.

By default, class metaobjects are identity functions; they do not change the program. So, to implement our translation, we define a new metaclass — a new class for class metaobjects — and use it to make the class metaobject for Person.

The metaclass for a class is specified by the metaclass declaration at the base level. For example, recall that the base-level program person.cc contains this:

```
metaclass Person : VerboseClass;      // metaclass declaration
```

This declaration specifies that the class metaobject for `Person` is an instance of `VerboseClass`.

A new metaclass must be a subclass of the default metaclass `Class`. Here is the definition of our new metaclass `VerboseClass`:

```
// verbose.cc
#include "mop.h"

class VerboseClass : public Class {
public:
    VerboseClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*,
                               Ptree*, Ptree*);
};

Ptree* VerboseClass::TranslateMemberCall(Environment* env,
        Ptree* object, Ptree* op, Ptree* member, Ptree* arglist)
{
    return Ptree::Make("(puts(\"%p()\"), %p)",
                       member,
                       Class::TranslateMemberCall(env, object, op,
                                                  member, arglist));
}
```

The metaclass `VerboseClass` is just a regular C++ class. It inherits from `Class` and overrides one member function. `TranslateMemberCall()` takes an expression such as `billy.Age()` and returns the translated one. Both the given expression and the translated one are represented in the form of parse tree. `Ptree` is the data type for that representation.

Since the class metaobject for `Person` is responsible only for the translation involving the class `Person`, `TranslateMemberCall()` does not have to care about other classes. It just constructs a comma expression:

`(puts("` *member-name*`"),` *member-call*`)`

from the original expression. `Ptree::Make()` is a convenience function to construct a new parse tree. `%p` is replaced with the following argument.

We do not need many concepts to write a meta-level program. As we saw above, the key concepts are only three. Here, we summarize these key concepts:

class metaobject: The representation of a class at the meta level.

metaclass: A class whose instances are class metaobjects.

metaclass `Class`: The default metaclass. It is named because its instances are class metaobjects.

2.1.4 Compile, debug, and run

We first compile the meta-level program and extend the OpenC++ compiler, which is used to compile the base-level program. Because OpenC++ is a reflective language, the meta-level program is compiled by the OpenC++ compiler itself. Then, the compiled code is linked with the original compiler (`opencxx.a`)[1] and a new extended compiler is produced. Let's name the extended compiler `myocc`:

```
% occ -- -g -o myocc opencxx.a verbose.cc
```

The options following `--`, such as `-g`, are passed to the back-end C++ compiler. `verbose.cc` is compiled with the `-g` option and linked with `opencxx.a`. The produced file is `myocc` specified by the `-o` option. Unless the `-c` option is given, the OpenC++ compiler produces an executable file.

Next, we compile the base-level program `person.cc` with the extended compiler `myocc`:

```
% myocc -- -g -o person person.cc
```

Now, we got an executable file `person`. It prints member function names if they are executed:

```
% person
Age()
age 24
BirthdayComes()
age 25
%
```

The OpenC++ MOP provides a few functions for debugging. First, programmers may use `Display()` on `Ptree` objects to debug a compiler. This function prints the parse tree represented by the `Ptree` object. For example, if the debugger is `gdb`, programmers may print the parse tree pointed to by a variable `object` in this way:

[1] In the current version, the OpenC++ compiler cannot dynamically load meta-level programs.

```
% gdb myocc
     :
(gdb) print object->Display()
billy
$1 = void
(gdb)
```

Similarly, the OpenC++ compiler accepts the `-s` option to print the whole parse tree of the given program. The parse tree is printed in the form of nested list:

```
% myocc -s person.cc
[typedef [char] [* __gnuc_va_list] ;]
     :
[metaclass Person : VerboseClass [] ;]
[[[class Person [] [{ [
    [public :]
    [[] [Person ( [[[int] [i]]] )] [{ [
        [[age = i] ;]
    ] }]]
    [[int] [Age ( [] )] [{ [
        [return age ;]
    ] }]]
    [[int] [BirthdayComes ( [] )] [{ [
        [return [++ age] ;]
    ] }]]
    [private :]
    [[int] [age] ;]
] }]]] ;]
[[] [main ( [] )] [{ [
    [[Person] [billy ( [24] )] ;]
    [[printf [( ["age %d\n" , [billy . Age [( [] )]]] )]] ;]
    [[printf [( ["age %d\n" , [billy . BirthdayComes ...
] }]]
%
```

This option makes the compiler just invoke the preprocessor and prints the parse tree of the preprocessed program. `[]` denotes a nested list. The compiler does not perform translation or compilation.

2.2 Syntax Extension for Verbose Objects

In the verbose object extension above, the base-level programmers have to write the `metaclass` declaration. The extension will be much easier to use if it provides easy syntax to declare verbose objects. Suppose that the base-level programmers may write something like this:

```
// person.cc
verbose class Person {
public:
    Person(int age);
    int Age() { return age; }
    int BirthdayComes() { return ++age; }
private:
    int age;
};
```

Note that the class declaration begins with a new keyword `verbose` but there is no `metaclass` declaration in the code above.

This sort of syntax extension is easy to implement with the OpenC++ MOP. To make the new keyword `verbose` available, the meta-level program must call `Class::RegisterMetaclass()` during the initialization phase of the compiler. So we add a `static` member function `Initialize()` to the class `VerboseClass`. It is automatically invoked at beginning by the MOP.

```
// verbose.cc
class VerboseClass : public Class {
public:
    VerboseClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*,
                               Ptree*, Ptree*);
    static BOOL Initialize();
};

BOOL VerboseClass::Initialize()
{
    RegisterMetaclass("verbose", "VerboseClass");
    return Class::Initialize();
}
```

`RegisterMetaclass()` defines a new keyword `verbose`. If a class declaration begins with that keyword, then the compiler recognizes that the metaclass is `VerboseClass`. This is all that we need for the syntax extension. Now the new compiler accepts the `verbose` keyword.

2.3 Matrix Library

The next example is a matrix library. It shows how the OpenC++ MOP works to specialize an optimization scheme for a particular class. The `Matrix` class is a popular example in C++ to show the usage of operator overloading. On the other hand, it is also famous that the typical implementation of the `Matrix` class is not efficient in practice. Let's think about how this statement is executed:

```
a = b + c - d;
```

The variables a, b, c, and d are Matrix objects. The statement is executed by invoking the operator functions +, -, and =. But the best execution is to inline the operator functions in advance to replace the statement:

```
for(int i = 0; i < N; ++i)
    a.element[i] = b.element[i] + c.element[i] - d.element[i];
```

C++'s inline specifier does not do this kind of smart inlining. It simply extracts a function definition but it does not fuse multiple extracted functions into efficient code as shown above. Expecting that the C++ compiler automatically performs the fusion is not realistic.

We use the OpenC++ MOP to implement this smart inlining specialized for the Matrix class. Again, we follow the three steps of the OpenC++ programming.

2.3.1 What the base-level program should look like

The objective of the matrix library is to provide the matrix data type as it is a built-in type. So the base-level programmers should be able to write:

```
Matrix a, b, c;
double k;
    :
a = a * a + b - k * c;
```

Note that the last line includes both a vector product a * a and a scalar product k * c.

2.3.2 What the base-level program should be translated

We've already discussed this step. The expressions involving Matrix objects are inlined as we showed above. We do not inline the expressions if they include more than one vector products. The gain by the inlining is relatively zero against two vector products.

Unlike the verbose objects example, we need runtime support code in this example. It is the class definition of Matrix. Note that the base-level programmers do not define Matrix by themselves. Matrix must be supplied as part of the compiler add-on for matrix arithmetics.

2.3.3 Write a meta-level program

To implement the inlining, we define a new metaclass `MatrixClass`. It is a metaclass only for `Matrix`. `MatrixClass` overrides a member function `TranslateAssign()`:

```
// matrixclass.cc

Ptree* MatrixClass::TranslateAssign(Environment* env,
                        Ptree* object, Ptree* op, Ptree* expr)
{
    if(we can inline on the expression)
        return generate optimized code
    else
        return Class::TranslateAssign(env, object, op, expr);
}
```

This member function translates an assignment expression. `object` specifies the L-value expression, `op` specifies the assignment operator such as `=` and `+=`, and `expr` specifies the assigned expression. If the inlining is not applicable, this function invokes `TranslateAssign()` of the base class. Otherwise, it parses the given `expr` and generate optimized code.

Since `expr` is already a parse tree, what this function has to do is to traverse the tree and sort terms in the expression. It is defined as a recursive function that performs pattern matching for each sub-expression. Note that each operator makes a sub-expression. So an expression such as `a + b - c` is represented by a parser tree:

```
[[a + b] - c]
```

The OpenC++ MOP provides a convenience function `Ptree::Match()` for pattern matching. So the tree traverse is described as follows:

```
static BOOL ParseTerms(Environment* env, Ptree* expr, int k)
{
    Ptree* lexpr;
    Ptree* rexpr;

    if(expr->IsLeaf()){     // if expr is a variable
        termTable[numOfTerms].expr = expr;
        termTable[numOfTerms].k = k;
        ++numOfTerms;
        return TRUE;
    }
    else if(Ptree::Match(expr, "[%? + %?]", &lexpr, &rexpr))
        return ParseTerms(env, lexpr, k)
```

```
                && ParseTerms(env, rexpr, k);
    else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
        return ParseTerms(env, lexpr, k)
                && ParseTerms(env, rexpr, -k);
    else if(Ptree::Match(expr, "[( %? )]", &lexpr))
        return ParseTerms(env, lexpr, k);
    else if(Ptree::Match(expr, "[- %?]", &rexpr))
        return ParseTerms(env, rexpr, -k);
    else
        return FALSE;
}
```

This function recursively traverses the given parse tree `expr` and stores the variables in `expr` into an array `termTable`. It also stores the flag (`+` or `-`) of the variable into the array. The returned value is `TRUE` if the sorting is successfully done.

After `ParseTerms()` is successfully executed, each term in the expression is stored in the array `termTable`. The rest of the work is to construct an inlined code from that array:

```
static Ptree* DoOptimize0(Ptree* object)
{
    Ptree* index = Ptree::GenSym();
    return Ptree::MakeStatement(
        "for(int %p = 0; %p < %s * %s; ++%p)\"
        "    %p.element[%p] = %p;",
        index, index, SIZE, SIZE, index,
        object, index, MakeInlineExpr(index));
}
```

`Ptree::GenSym()` returns a symbol name that has not been used. It is used as a loop variable. `Ptree::MakeStatement()` is a similar function to `Ptree::Make()`. It constructs a parse tree representing a statement instead of an expression. `MakeInlineExpr()` looks at the array and produces an inlined expression:

```
static Ptree* MakeInlineExpr(Ptree* index_var)
{
    int i;
    Ptree* expr;
    Ptree* inline_expr = nil;

    for(i = numOfTerms - 1; i >= 0; --i){
        char op;
        if(termTable[i].k > 0)
            op = '+';
```

```
        else
            op = '-';

        expr = Ptree::Make("%c %p.element[%p]",
                           op, termTable[i].expr, index_var);
        inline_expr = Ptree::Cons(expr, inline_expr);
    }

    return inline_expr;
}
```

The complete program of this example is `matrixclass.cc`, which is distributed together with the OpenC++ compiler. See that program for more details. It deals with the scalar and vector products as well as simple `+` and `-` operators.

2.3.4 Write runtime support code

Writing the runtime support code is straightforward. The class `Matrix` is defined in regular C++ except the `metaclass` declaration:

```
// matrix.h
const N = 3;

metaclass Matrix : MatrixClass;
class Matrix {
public:
    Matrix(double);
    Matrix& operator = (Matrix&);
        :
    double element[N * N];
};

Matrix& operator + (Matrix&, Matrix&);
Matrix& operator - (Matrix&, Matrix&);
Matrix& operator * (Matrix&, Matrix&);
Matrix& operator * (double, Matrix&);
```

Note that the class `Matrix` is a complete C++ class. It still works if the `metaclass` declaration is erased. For more details, see the sample program `matrix.cc`. They must be compiled by the OpenC++ compiler.

2.4 Syntax Extension for the Matrix Library

Initializer

We can also implement syntax sugar for the matrix library. First of all, we enable the following style of initialization:

```
Matrix r = { 0.5, -0.86, 0, 0.86, 0.5, 0, 0, 0, 1 };
```

This notation is analogous to initialization of arrays. In regular C++, however, an object cannot take an aggregate as its initial value. So we translate the statement shown above by `MatrixClass` into this correct C++ code:

```
double tmp[] = { 0.5, -0.86, 0, 0.86, 0.5, 0, 0, 0, 1 };
Matrix r = tmp;
```

To do this translation, `MatrixClass` must override a member function `TranslateInitializer()`:

```
// matrixclass.cc

Ptree* MatrixClass::TranslateInitializer(Environment* env,
            Ptree* init, PtreeArray& before, PtreeArray& after)
{
    Ptree* sep = Ptree::First(init);
    Ptree* expr = Ptree::Second(init);
    if(sep->Eq('=') && expr->WhatIs() == PtreeBraceId){
        Ptree* tmp = Ptree::GenSym();
        before.Append(Ptree::Make("double %p[] = %p;\n",
                                  tmp, expr));
        return Ptree::Make("= %p", tmp);
    }
    else
        return Class::TranslateInitializer(env, init, before,
                                           after);
}
```

This member function translates the initializer of a `Matrix` object. For example, it receives, as the argument `init`, the initializer `= { 0.5, ... }` following `tmp[]`. If the initializer is an aggregate, this member function translates it as we mentioned above. The temporary array is stored in `before`. The `Ptree` objects stored in `before` are inserted before the variable declaration after the translation.

The `forall` statement

The second syntax sugar we show is a new kind of loop statement. For example, the programmer may write:

```
Matrix m;
    :
m.forall(e){ e = 0.0; };      // ; is always necessary
```

`e` is bound to each element during the loop. The programmer may write any statements between `{` and `}`. The loop statement above assigns `0.0` to all the elements of the matrix `m`. This new loop statement should be translated into this:

```
for(int i = 0; i < N; ++i){
    double& e = m.element[i];
    e = 0.0;
}
```

The OpenC++ MOP allows programmers to implement a new kind of statement such as `forall`. Because the new kind of statement is regarded as an expression in grammar, programmers can write it at any place an expression appears. However, they have to put a semicolon (`;`) at the end of the statement.

To implement this statement, first we have to register a new keyword `forall`:

```
// matrixclass.cc

BOOL MatrixClass::Initialize()
{
    RegisterNewWhileStatement("forall");
    return Class::Initialize();
}
```

`Initialize()` is a member function automatically invoked at the beginning of compilation.

We also have to define what the `forall` statement is translated into. `MatrixClass` overrides a member function `TranslateUserStatement()`:

```
Ptree* MatrixClass::TranslateUserStatement(Environment* env,
         Ptree* object, Ptree* op, Ptree* keyword, Ptree* rest)
{
    Ptree *tmp, *body, *index;
```

```
    Ptree::Match(rest, "[([%?]) %?]", &tmp, &body);
    index = Ptree::GenSym();
    return Ptree::MakeStatement(
        "for(int %p = 0; %p < %s * %s; ++%p){\n"
        "    double& %p = %p%p element[%p];\n"
        "    %p }\n",
        index, index, SIZE, SIZE, index,
        tmp, object, op, index, TranslateStatement(env, body));
}
```

The `forall` statement is parsed so that `object`, `op`, and `keyword` are bound to `m . forall`, respectively. `rest` is bound to the rest of code `(e){ e = 0.0; }`. `TranslateUserStatement()` uses those arguments to construct the substituted code. Note that it calls `MakeStatement()` instead of `Make()`. This is because the constructed code is not an expression but a statement. `TranslateStatement()` is called to recursively translate the `body` part of the `forall` statement.

2.5 Before-Method

CLOS provides a useful mechanism called before- and after- methods. They are special methods that are automatically executed before or after the primary method is executed.

2.5.1 What the base-level program should look like

We implement before-methods in OpenC++. For simplicity, if the name of a member function is `before_f()`, then our implementation regards this member function as the before-method for the member function `f()`. We don't introduce any syntax extension. For example,

```
metaclass Queue : BeforeClass;
class Queue {
public:
    Queue(){ i = 0; }
    void Put(int);
    void before_Put();
    int Peek();

private:
    int buffer[SIZE];
    int i;
};
```

`Put()` has a before-method `before_Put()` whereas `Peek()` does not since `before_Peek()` is not defined in the class `Queue`.

The before-method is automatically executed when the primary method is called. If the programmer say:

```
Queue q;
   :
q.Put(3);
int k = q.Peek();
```

The execution of `q.Put(3)` is preceded by that of the before-method `q.before_Put()`. Since `Peek()` does not have a before-method, the execution of `q.Peek()` is not preceded by any other function.

2.5.2 What the base-level program should be translated

In this extension, the class declaration does not require any change. Only member function calls need to be translated. For example,

```
q.Put(3)
```

should be translated into:

```
((tmp = &q)->before_Put(), tmp->Put(3))
```

This expression first stores the address of `q` in a temporary variable `tmp` and then calls `before_Put()` and `Put()`. The address of `q` should be stored in the temporary variable to avoid evaluating `q` more than once. Also, the temporary variable must be declared in advance.

2.5.3 Write a meta-level program

The metaclass `BeforeClass` overrides `TranslateMemberCall()` to implement the translation mentioned above. The complete program of `BeforeClass` is `before.cc` in the distribution package. Here, we explain some key topics in the program.

First of all, we have to decide whether there is a before-method for a given member function. `BeforeFunction()` does this work:

```
Ptree* BeforeClass::BeforeFunction(Ptree* name)
{
    Ptree* before = Ptree::Make("before_%p", name);
    if(IsMember(before) != 0)
        return before;
    else
        return nil;
}
```

In the first line, this produces the name of the before-method by `Ptree::Make()`. Then it calls `IsMember()` supplied by `Class`. `IsMember()` returns non-zero if the class has a member that matches the given name.

The next issue is a temporary variable. We have to appropriately insert a variable declaration to use a temporary variable. The name of the temporary variable is obtained by calling `Ptree::GenSym()`. The difficulty is how to share the temporary variable among member function calls. To do this, we record the temporary variable in the environment. We can define a subclass of `Environment::ClientData` and record the object in an environment. Let's define the subclass:

```
class TempVarName : public Environment::ClientData {
public:
    TempVarName(Ptree* p) { varname = p; }
    Ptree* varname;
};
```

A `TempVarName` object includes a single member `varname`, which is the name of a temporary variable. We record this object in the environment when we first declare the temporary variable. Then we use this object to check whether a temporary variable has been already declared.

```
Ptree* class_name = Name();
TempVarName* tmpvar
    = (TempVarName*)env->LookupClientData(this, class_name);
if(tmpvar != nil)
    varname = tmpvar->varname;
else{
    varname = Ptree::GenSym();
    tmpvar = new TempVarName(varname);
    Ptree* decl = Ptree::MakeStatement("%p* %p;",
                                       class_name, varname);
    env->InsertDeclaration(decl, this, class_name, tmpvar);
}

return Ptree::Make("((%p=%c%p)->%p(), %p->%p%p)",
                   varname, (op->Eq('.') ? '&' : ' '), object,
                   before_func, varname, member, arglist);
```

This is the core part of `TranslateMemberCall()` supplied by `BeforeClass`. It first looks for a `TempVarName` object by calling `LookupClientData()` with two search keys. If it is not found, a variable declaration `decl` is produced by `Make()` and it is inserted into the translated program by `InsertDeclaration()`. `InsertDeclaration()` also records a `TempVarName` object for future reference.

2.6 Wrapper Function

A wrapper function is useful to implement language extensions such as concurrency. A wrapper function is generated by the compiler and it intercepts the call of the original "wrapped" function. For example, the wrapper function may perform synchronization before executing the original function. The original function is not invoked unless the wrapper function explicitly calls it.

2.6.1 What the base-level program should be translated

We show a metaclass `WrapperClass` that generates wrapper functions. If `WrapperClass` is specified, it generates wrapper functions for the member functions of the class. And it translates the program so that the wrapper functions are invoked instead of the wrapped member functions. For example, suppose that the program is something like this:

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

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

void f()
{
    Point p;
    p.Move(3, 5);    // call Move()
}
```

The compiler generates a wrapper function `wrapper_Move()` for `Move()`. The call of `Move()` in `f()` is substituted by the call of the wrapper function. For simplicity, we make the wrapper function just invoke the wrapped function `Move()` without doing anything else. The translated program should be this:

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

```
public:
    void wrapper_Move(int, int);
};

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

void Point::wrapper_Move(int p1, int p2)  // generated wrapper
{
    // should do something here in a real example
    Move(p1, p2);
}

void f()
{
    Point p;
    p.wrapper_Move(3, 5);  // call the wrapper
}
```

2.6.2 Write a meta-level program

`WrapperClass` has to do three things: (1) to insert member declarations for wrapper functions, (2) to generate the definitions of the wrapper functions, and (3) to substitute the wrapper functions for the member functions in member call expressions. `WrapperClass` overrides `TranslateBody()` for (1), `TranslateMemberFunctionBody()` for (2), and `TranslateMemberCall()` for (3).

First, we show `TranslateBody()`. Its work is to translate the body of a class declaration. It looks at a member of the class and, if the member is a function, it inserts the declaration of the wrapper function. To get a member name, we use `NthMemberName()`:

```
Ptree* WrapperClass::TranslateBody(Environment* env, Ptree* body)
{
    Ptree* decl = Ptree::Make("public:\n");
    Ptree* name;
    TypeInfo t;
    int i = 1;
    while((name = NthMemberName(i++)) != nil){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId
           && whatis != PtreeDestructorId)
            if(LookupMemberType(env, name, t))
                if(t.WhatIs() == FunctionType){
                    Ptree* mem = t.MakePtree(WrapperName(name));
```

```
                    decl = Ptree::Make("%p %p;\n",
                                       decl, mem);
                }
    }

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

Ptree* WrapperClass::WrapperName(Ptree* name)
{
    return Ptree::Make("wrapper_%p", name);
}
```

In the `while` loop, we first check that the member is not a constructor or a destructor. If it is a constructor or a destructor, the parse tree indicated by `name` is `LeafClassNameId` or `PtreeDestructorId`. Then we get the type of the member by calling `LookupMemberType()`. If the member is a function, the type is `FunctionType`.

After we make sure that the member is a function, we generates the declaration of the wrapper function. We use `MakePtree()` to generate it. This member function of `TypeInfo` makes a parse tree that represents the type name. Suppose that a `TypeInfo t` is the pointer type to integer. Then `t.MakePtree()` returns:

```
int*
```

We can also give a variable name to `MakePtree()`. For example:

```
t.MakePtree(Ptree::Make("i"))
```

returns:

```
int* i
```

Similarly, if `t` is a function type, we can give a function name and get a parse tree that represents the function declaration. For example:

```
t.MakePtree(Ptree::Make("foo"))
```

returns if `t` is the function type that takes two integer arguments and returns a pointer to a character:

```
char* foo(int, int)
```

If `t` is the pointer type to that function, the returned parse tree is:

```
char* (*foo)(int, int)
```

Next, we mention `TranslateMemberFunctionBody()`. It generates the definition of the wrapper function if the member function is not a constructor or a destructor. The actual generation is done by `MakeWrapper()`:

```
void WrapperClass::MakeWrapper(Environment* env, Ptree* name,
                               Ptree* wrapper_name, TypeInfo& t,
                               BOOL inlined)
{
    TypeInfo atype;
    TypeInfo rtype;
    int i;
    Ptree* wrapper;
    Ptree* arglist = nil;

    for(i = 1; t.NthArgument(i, atype); ++i){
        Ptree* arg = atype.MakePtree(Ptree::Make("p%d", i));
        if(i == 1)
            arglist = arg;
        else
            arglist = Ptree::Make("%p,%p", arglist, arg);
    }

    t.Dereference(rtype);
    Ptree* body = WrapperBody(env, name, wrapper_name, i-1, t);
    Ptree* head = Ptree::Make("%p::%p(%p)",
                              Name(), wrapper_name, arglist);
    if(rtype.WhatIs() != UndefType)
        head = rtype.MakePtree(head);

    if(inlined)
        wrapper = Ptree::Make("inline %p{%p}\n",
                              head, body));
    else
        wrapper = Ptree::Make("%p{%p}\n", head, body));

    AppendAfterToplevel(wrapper);
}
```

By the first `for` loop, this member function constructs the argument list `arglist`. The name of the arguments are `p1`, `p2`, `p3`, and so on. To obtain the argument type, we call `NthArgument()` of `TypeInfo`. Then we call `MakePtree()` to construct each argument declaration.

The return type of the function is attached by calling `MakePtree()` on the return type `rtype`, which is obtained by calling `Dereference()` on the function type `t`. The argument passed to `MakePtree()` is the rest of the function header such as `X::f(int p1, char p2)`.

The constructed definition of the wrapper function is finally inserted into the translated program by `AppendAfterToplevel()`. It inserts the constructed parse tree just after the (translated) definition of the original member function. This avoids unnecessary duplicated copies of the wrapper functions. If we generate the wrapper functions when `TranslateBody()` processes the class declaration, the wrapper functions would be duplicated every time the class declaration is included by a different source file.

Finally, we show `TranslateMemberCall()`, which substitutes the wrapper function for the member function originally called. Its definition is simple. It calls the member function of the base class `Class` with the wrapper function's name instead of the original member's name:

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

2.6.3 Subclass of `WrapperClass`

The complete program of `WrapperClass` is found in `wrapper.cc`, which is distributed together with the OpenC++ compiler. Although the wrapper functions generated by `WrapperClass` do not perform anything except calling the original member function, we can define a subclass of `WrapperClass` to generate the wrapper functions that we need. (Note that, to make the subclass effective, we also have to edit the metaclass declaration so that the compiler uses the subclass.)

For example, suppose that we need a wrapper function that perform synchronization before calling the original member function. This sort of wrapper function is typical in concurrent programming. To implement this, we just define a subclass `SyncClass` and override `WrapperBody()`:

```
#include "wrapper.h"
class SyncClass : public WrapperClass {
public:
    SyncClass(Ptree* d, Ptree* m) : WrapperClass(d, m) {}
    Ptree* WrapperBody(Environment*, Ptree*, Ptree*, int,
                       TypeInfo&);
};
```

`WrapperBody()` is a `virtual` function and it is called by `MakeWrapper()` to construct the function body of the wrapper function. `WrapperBody()` supplied by `WrapperClass` returns a `return` statement such as:

`return` *original-function*`(p1, p2, ...);`

So we define `WrapperBody()` supplied by `SyncClass` as shown below:

```
Ptree* SyncClass::WrapperBody(Environment* env, Ptree* name,
                              Ptree* wrapper_name, int num_of_args,
                              TypeInfo& ftype)
{
    Ptree* ret = WrapperClass::WrapperBody(env, name, wrapper_name,
                                           num_of_args, ftype);
    return Ptree::Make("synchronize(); %p", ret);
}
```

This inserts `synchronize();` before the `return` statement.

As we see above, carefully designed metaclasses can be reused as the base class of another metaclass. Such metaclasses, that is, metaclass libraries, make it easier to write other metaclasses. Indeed, `MatrixClass` in the matrix example should be re-implemented so that other metaclasses such as `ComplexClass` can share the code for inlining with `MatrixClass`.

3 Base-Level Language (OpenC++)

This section addresses the language specification of OpenC++. OpenC++ is identical to C++ except two extensions. To connect a base-level program and a meta-level program, OpenC++ introduces a new kind of declaration into C++. Also, new extended syntax is available in OpenC++ if the syntax is defined by the meta-level program.

3.1 Base-level Connection to the MOP

OpenC++ provides a new syntax for metaclass declaration. This declaration form is the only connection between the base level and the meta level. Although the default metaclass is `Class`, programmers can change it by using this declaration form:

- `metaclass` *class-name* `:` *metaclass-name* [`(` *meta-arguments* `)`] `;`[2]

  This declares the metaclass for a class. It must appear before the class is defined. *meta-arguments* is a sequence of identifiers, type names, literals, and C++ expressions surrounded by `()`. The elements must be separated by commas. The identifiers appearing in *meta-arguments* do not have to be declared in advance. What should be placed at *meta-arguments* is specified by the metaclass.

The code shown below is an example of metaclass declaration:

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

The metaclass for `Point` is `PersistentClass`.

3.2 Syntax Extensions

The extended syntax described here is effective if programmers define it by the MOP. By default, it causes a syntax error. To make it available, programmers must register a new keyword, which is used in one of the following forms:

[2] [ ] means an optional field.

- Type Modifier *keyword* [(*function-arguments*)]

  The keyword registered to lead a type modifier may appear in front of type names, the `new` operator, or class declarations. For example, these statements are valid:

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

  Here, `distribute` and `remote` are registered keywords.

- Access Specifier *keyword* [(*function-arguments*)] :

  Programmers may define a keyword as a member-access specifier. It appears at the same place that the built-in access specifier such as `public` can appears. For example, if `after` is a registered keyword, then programmers may write:

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

- While-style Statement
  pointer `->` *keyword* (*expression*){ *statements* }
  object . *keyword* (*expression*){ *statements* }
  class-name::*keyword* (*expression*){ *statements* }

  A registered keyword may lead something like the `while` statement. In the grammar, that is not a statement but an expression. It can appear at any place where C++ expressions appear. *expression* is any C++ expression. It may be empty or separated by commas like function-call arguments. Here is an example of the while-style statement:

  ```
  Matrix m2;
  m2.forall(e){
      e = 0;
  };
  ```

  Note the last semicolon "`;`". It is needed because the `forall` statement is an "expressions" in the grammar.

  A registered keyword can also lead other styles of statements.

- For-style Statement
  pointer `->` *keyword* `(` *expr* `;` *expr* `;` *expr* `){` *statements* `}`
  object `.` *keyword* `(` *expr* `;` *expr* `;` *expr* `){` *statements* `}`
  class-name`::`*keyword* `(` *expr* `;` *expr* `;` *expr* `){` *statements* `}`

  The for-style statement takes three expressions like the `for` statement. Except that, it is the same as the while-style statement.

- Closure Statement
  pointer `->` *keyword* `(` *arg-declaration-list* `){` *statements* `}`
  object `.` *keyword* `(` *arg-declaration-list* `){` *statements* `}`
  class-name`::`*keyword* `(` *arg-declaration-list* `){` *statements* `}`

  The closure statement takes an argument declaration list instead of an expression. That is the only difference from the while-style statement. For example, programmers may write something like this:

  ```
  ButtonWidget b;
  b.press(int x, int y){
     printf("pressed at (%d, %d)\n", x, y);
  };
  ```

3.3 Loosened Grammar

Besides extended syntax, OpenC++'s grammar is somewhat loosened as compared with C++'s grammar. For example, the next code is semantically wrong in C++:

```
Point p = { 1, 3, 5 };
```

The C++ compiler will report that `p` cannot be initialized by "`{ 1, 3, 5 }`". Such an aggregate can be used only to initialize an array. The OpenC++ compiler simply accepts such a semantically-wrong code. It ignores semantical correctness expecting that the code will be translated into valid C++ code.

4 Metaobject Protocol (MOP)

At the meta level, the (base-level) programs are represented by objects of a few predefined classes (and their subclasses that programmers define). These objects are called *metaobjects* because they are *meta* representation of the programs. Source-to-source translation from OpenC++ to C++ is implemented by manipulating those metaobjects.

This section shows details of such metaobjects. They reflect various aspects of programs that are not accessible in C++. Although most of metaobjects provide means of introspection, some metaobjects represent a behavioral aspect of the program and enables to control source-to-source translation of the program. Here is the list of metaobjects:

- `Ptree` metaobjects:
  They represent a parse tree of the program. The parse tree is implemented as a nested-linked list.

- `TypeInfo` metaobjects:
  They represent types that appear in the program. The types include derived types such as pointer types and reference types as well as built-in types and class types.

- `Environment` metaobjects:
  They represent bindings between names and types. Since this MOP is a compile-time MOP, the runtime values bound to names are not available at the meta level.

- `Class` metaobjects:
  As well as they represent class definitions, they control source-to-source translation of the program. Programmers may define subclasses of `Class` in order to tailor the translation.

Distinguishing `TypeInfo` metaobjects and `Class` metaobjects might look like wrong design. But this distinction is needed to handle derived types. `TypeInfo` metaobjects were introduced to deal with derived types and fundamental types by using the same kind of metaobjects.

4.1 Representation of Program Text

Program text is accessible at the meta level in the form of parse tree. The parse tree is represented by a `Ptree` metaobject. It is implemented as a

Table 1: `static` member functions on `Ptree`

- `Ptree* First(Ptree* lst)` returns the first element of `lst`.

- `Ptree* Rest(Ptree* lst)` returns the rest of `lst` except the first element, that is, the *cdr* field of `lst`.

- `Ptree* Second(Ptree* lst)` returns the second element of `lst`.

- `Ptree* Third(Ptree* lst)` returns the third element of `lst`.

- `Ptree* Nth(Ptree* lst, int n)` returns the `n`-th element of `lst`. `Nth(lst, 1)` is equivalent to `First(lst)`.

- `Ptree* Last(Ptree* lst)` returns the last cons cell, which is a list containing only the last element of `lst`.

- `Ptree* ListTail(Ptree* lst, int k)` returns a sublist of `lst` obtained by omitting the first `k` elements. `ListTail(lst, 1)` is equivalent to `Rest(lst)`.

- `int Length(Ptree* lst)` returns the number of the elements of `lst`.

- `Ptree* Cons(Ptree* a, Ptree* b)` returns a cons cell whose *car* field is `a` and whose *cdr* is `b`.

- `Ptree* List(Ptree* e1, Ptree* e2, ...)` returns a list whose elements are `e1`, `e2`, ... `List()` returns a null list $[\![\ ]\!]$, which is denoted by `nil` or `NIL`.

- `Ptree* Append(Ptree* lst1, Ptree* lst2)` concatenates `lst1` and `lst2`. It returns the resulting list.

- `Ptree* CopyList(Ptree* lst)` returns a new list whose elements are the same as `lst`'s.

- `BOOL Eq(Ptree* lst, char x)`
- `BOOL Eq(Ptree* lst, char* x)`
- `BOOL Eq(Ptree* lst, Ptree* x)` returns `TRUE` if `lst` and `x` are equal.

nested linked-list of lexical tokens — the S expressions in the Lisp terminology. For example, this piece of code:

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

is parsed into:

```
⟦⟦int⟧ ⟦a = ⟦b + ⟦c * 2⟧⟧⟧ ;⟧
```

Here, ⟦ ⟧ denotes a linked list. Note that operators such as `=` and `+` make sublists. The sublists and their elements (that is, lexical tokens such as `a` and `=`) are also represented by `Ptree` metaobjects.

4.1.1 Basic Operations

To manipulate linked lists, the MOP provides some `static` member functions on `Ptree`, which are familiar to Lisp programmers. Table 1 shows those `static` member functions. In addition, the following member functions are available on `Ptree` metaobjects:

- `BOOL IsLeaf()`
This returns `TRUE` if the metaobject indicates a lexical token.

- `void Display()`
This prints the metaobject on the console for debugging. Sublists are surrounded by `[` and `]`.

- `int Write(FILE* fptr)`
This writes the metaobject to the file specified by `fptr`. Unlike `Display()`, sublists are not surrounded by `[` and `]`. This member function returns the number of written lines.

- `PtreeId WhatIs()`
This returns an `enum` constant that corresponds to the syntactical meaning of the code that the metaobject represents. For example, the metaobject represents a class name, this member function returns `LeafClassNameId`. If it represents a `if` statement, `PtreeIfStatementId` is returned. The returned constants are listed in Table 2.

The parse tree is basically a long list of the lexical tokens that appear in the program although some of them are grouped into sublists. The order of

Table 2: `enum` constants returned by `Ptree::WhatIs()`

- Toplevel declarations
  `PtreeDeclarationId`, `PtreeFunctionId`, `PtreeTemplateDeclId`, `PtreeMetaclassId`, `PtreeTypedefId`, `PtreeLinkageId`

- Statements
  `PtreeExprStatementId`, `PtreeLabelId`, `PtreeCaseLabelId`, `PtreeIfStatementId`, `PtreeSwitchStatementId`, `PtreeWhileStatementId`, `PtreeDoStatementId`, `PtreeForStatementId`, `PtreeBreakStatementId`, `PtreeContinueStatementId`, `PtreeReturnStatementId`, `PtreeGotoStatementId`

- Expressions
  `PtreeInfixExprId`, `PtreeCondExprId` (conditional), `PtreeCastId`, `PtreeUnaryExprId`, `PtreePostfixExprId`, `PtreeMemberAccessExprId`, `PtreeSizeofExprId`, `PtreeAssignExprId`, `PtreeSizeofTypeId`, `PtreePtrToMemExprId` (pointer to member, `->*` or `.*`), `PtreeNewId`, `PtreeDeleteId`, `PtreeFstyleCastId` (function-style cast), `PtreeUserStatementId`, `PtreeStaticUserStatementId`, `PtreeActualArgsId`

- Class Declarations
  `PtreeEnumId`, `PtreeClassId`

- Groups
  `PtreeParenId` (parenthesis), `PtreeBracketId`, `PtreeBraceId`, `PtreeAngleId`,

- Leaf Nodes
  `LeafTypeNameId`, `LeafClassNameId`, `LeafPointerId`, `LeafReferenceId`, `LeafConstValueId`, `LeafNameId`, `LeafReservedId`, `LeafThisId`, `LeafReservedModifierId`,

- Others
  `PtreeTemplateNameId`, `PtreeQClassNameId` (qualified class name), `PtreeQnameId` (qualified name), `PtreePtrToMemberId` (pointer to member, `::*`), `PtreeCvQualifierId` (`const` or `volatile`), `PtreeDestructorId`, `PtreeOperatorFuncId`, `PtreeAccessCtrlId`, `PtreeUserKeywordId`

the elements of that list is the same as the order in which the lexical tokens appear. But if some fields such as the type field are omitted in the program, then `nil` lists ⟦⟧ are inserted at that place. For example, if the return type of a function declaration is omitted as follows:

```
main(int argc, char** argv){ }
```

then a `nil` list is inserted at the head of the list:

```
⟦⟦⟧ main ⟦( ⟦⟦int⟧ ⟦argc⟧⟧ , ⟦⟦char⟧ ⟦* * argv⟧⟧⟧ )⟧ ⟦{ ⟦⟧ }⟧⟧
```

Since the function body is also omitted, a `nil` list is inserted between `{` and `}`.

4.1.2 Construction

Programmers can make `Ptree` metaobjects. Because the MOP provides a conservative garbage collector, they don't need to care about deallocation of the metaobjects. The next `static` member functions on `Ptree` are used to make a `Ptree` metaobjects.

• `Ptree* Make(char* format, [Ptree* sublist, ...])`
This makes a `Ptree` metaobject specified the `format` as `printf()` does. The `format` is a null-terminated string. All occurrences of `%c` (character), `%d` (integer), `%s` (character string), and `%p` (`Ptree`) in the `format` are replaced with the values following the `format`. `%%` in the `format` is replaced with `%`.

• `Ptree* MakeStatement(char* format, [Ptree* sublist, ...])`
This is identical to `Make()` except the generated `Ptree` metaobject represents not an expression but a statement.

• `Ptree* GenSym()`
This generates a unique symbol name (aka identifier) and returns it. The returned symbol name is used as the name of a temporary variable, for example.

The `Ptree` metaobject returned by `Make()` and `MakeStatement()` is not a real parse tree.[3] It is just a unparsed chunk of characters. Although programmers can use `Ptree` metaobjects generated by `Make()` as they use

[3] At least, for the time being.

other Ptree metaobjects, the structure of those metaobjects does not reflect the code they represent.

Using Make(), programmers can easily generate any piece of code to substitute for part of the original source code. For example, suppose array_name is ⟦xpos⟧ and offset is 3. The following function call:

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

makes a Ptree metaobject that represents:

```
xpos[3]
```

%p simply expand a given Ptree metaobject as a character string. Thus programmers may write something like:

```
Ptree::Make("char* GetName(){ return \"%p\"; }",
            array_name);
```

Note that a double quote " must be escaped by a backslash \ in a C++ string. \"%p\" makes a string literal. The function call above generates the code below:

```
char* GetName(){ return "xpos"; }
```

Although Make() and MakeStatement() follow the old printf() style, programmers can also use a more convenient style similar to Lisp's backquote notation. For example,

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

The expression above can be rewritten using qMake() as follows:

```
Ptree::qMake("`array_name`[`offset`]")
```

Note that the "backqouted" C++ expressions array_name and offset are directly embedded in the C++ string. Their occurrence are replaced with the value of the expression. This replacement cannot be implemented in regular C++. It is implemented by the metaclass for Ptree.

• Ptree* qMake(char* text)
This makes a Ptree metaobject that represents the text. Any C++ expression surrouned by backquotes ` can appear in text. Its occurence is replaced with the value denoted by the expression. The type of the expression must be Ptree*, int, or char*.

• Ptree* qMakeStatement(char* text)
This is identical to qMake() except the generated Ptree metaobject represents not an expression but a statement.

4.1.3 Pattern Matching

The MOP provides a `static` member function on `Ptree` metaobjects for pattern matching.

• `BOOL Match(Ptree* list, char* pattern,` [`Ptree** sublist,` ...])
This compares the `pattern` and `list`. If they match, this function returns `TRUE` and binds the `sublist`s to appropriate sublists of the `list`, as specified by the `pattern`. Note that the type of `sublist` is pointer to `Ptree*`.

For example, the function `Match()` is used as follows:

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

The pattern `[%? + %?]` matches a linked list that consists of three elements if the second one is `+`. If an expression `expr` matches the pattern, `lexpr` gets bound to the first element of `expr` and `rexpr` gets bound to the third element.

The `pattern` is a null-terminated string. Since `Match()` does not understand the C++ grammar, lexical tokens appearing in the pattern must be separated by a white space. For example, a pattern `a+b` is regarded as a single token. The pattern is constructed by these rules:

1. A word (characters terminated by a white space) is a pattern that matches a lexical token.
2. `%[`, `%]`, and `%%` are patterns that match `[`, `]`, and `%`.
3. `[]` is a pattern that matches a null list (`nil`).
4. `[`*pat1 pat2* ... `]` is a pattern that matches a list of *pat1*, *pat2*, ...
5. `%*` is a pattern that matches any token or list.
6. `%?` is a pattern that matches any token or list. The matched token or list is bound to `sublist`.
7. `%_` is a pattern that matches the rest of the list (the `cdr` part).
8. `%r` is a pattern that matches the rest of the list. The matched list is bound to `sublist`.

4.1.4 Reifying Program Text

If a `Ptree` metaobject represents a literal such as an integer constant and a string literal, we can obtain the value denoted by the literal.

• `BOOL Reify(unsigned int& value)`
This returns `TRUE` if the metaobject represents an integer constant. The denoted value is stored in `value`. Note that the denoted value is always a positive number because a negative number such as `-4` generates two destinct tokens such as `-` and `4`.

• `BOOL Reify(char*& string)`
This returns `TRUE` if the metaobject represents a string literal. A string literal is a sequence of character surrounded by double quotes `"`. The denoted null-terminated string is stored in `string`. The denoted string does not include the double quotes at the both ends. Also, the escape sequences are not expanded.

4.1.5 Convenience Classes `PtreeIter` and `PtreeArray`

The MOP provides a few convenience classes to help programmers to deal with `Ptree` objects. One of them is the class `PtreeIter`. It is useful to perform iteration on a list of `Ptree` objects. Suppose that `expr` is a list:

```
PtreeIter next(expr);
Ptree* p;
while((p = next()) != nil)
    compute on p ;
```

Each element of `expr` is bound to `p` one at a time. The operator `()` on `PtreeIter` objects returns the next element. This is the same as calling a member function `Pop()`. If the reader likes the `for`-loop style, she can also say:

```
for(PtreeIter i = expr; !i.Empty(); i++)
    compute on *i ;
```

Although this interface is slightly slow, it distinguishes the end of the list and a `nil` element.

Another class `PtreeArray` is for an unbounded array of `Ptree` objects. It is used as follows (suppose that `expr` is a `Ptree` object):

```
PtreeArray a;             // allocate an array
a.Append(expr);           // append expr to the end of the array
Ptree* p = a[0];          // get the first element
Ptree* p2 = a.Ref(0);     // same as a[0]
int n = a.Number();       // get the number of elements
Ptree* lst = a.All();     // get a list of all the elements
```

4.2 Representation of Types

`TypeInfo` metaobjects represent types. Because C++ deals with derived types such as pointer types and array types, `Class` metaobjects are not used for primary representation of types. `TypeInfo` metaobjects do not treat `typedef`ed types as independent types. They are treated just as aliases of the original types.

The followings are member functions on `TypeInfo` metaobjects:

- `TypeInfoId WhatIs()`

This returns an `enum` constant that corresponds to the kind of the type: `BuiltInType`, `ClassType` (including `class`, `struct`, and `union`), `EnumType`, `TemplateType`, `PointerType`, `ReferenceType`, `PointerToMemberType`, `ArrayType`, `FunctionType`, `TemplateFunctionType`, or `UndefType` (the type is unknown).

- `Ptree* FullTypeName()`

This returns the full name of the type if the type is a built-in type, a class type, an `enum` type, or a template class type. Otherwise, this returns `nil`. For example, if the type is a nested class `Y` defined within a class `X`, this returns `X::Y`.

- `uint IsBuiltInType()`

This returns a bit field that represents details of the built-in type. If the type is not a built-in type, it simply returns `0` (`FALSE`). To test the bit field, these masks are available: `CharType`, `IntType`, `ShortType`, `LongType`, `SignedType`, `UnsignedType`, `FloatType`, `DoubleType`, and `VoidType`. For example, *bit_field* `& LongType` is `TRUE` if the type is `long`, `unsigned long`, or `signed long`.

- `Class* ClassMetaobject()`

This returns a `Class` metaobject that represent the type. If the type is not a `class` type, it simply returns `nil`.

- `void Dereference(TypeInfo& t)`

This returns the dereferenced type in `t`. For example, if the type is `int**`, then `Dereference()` on the type returns a `TypeInfo` metaobject for `int*`. If a function type is dereferenced, `t` becomes its return type. If dereferencing is not possible, the `Undef` type is returned in `t`.

- `void Dereference()`
This is identical to `Dereference(TypeInfo&)` except that the `TypeInfo` metaobject itself is changed to represent the dereferenced type.

- `BOOL NthArgument(int nth, TypeInfo& t)`
If the type is `FunctionType`, this returns the type of the `nth` argument at `t`. If the type is not `FunctionType` or the `nth` argument does not exist, this function returns `FALSE`. If the `nth` argument is ... (ellipses), then the returned type is `UndefType`.

- `Ptree* MakePtree(Ptree* var_name = nil)`
This makes a `Ptree` metaobject that represents the type name. For example, if the type is pointer to integer, this returns ⟦`int*` *var_name*⟧. `var_name` may be `nil`.

4.3 Representation of Environment

`Environment` metaobjects represent bindings between names and types. If the name denotes a variable, it is bound to the type of that variable. Otherwise, if the name denotes a type, it is bound to the type itself. Programmers can look up names by these member functions on `Environment` metaobjects:

- `BOOL Lookup(Ptree* name, BOOL& is_type_name, TypeInfo& t)`
This looks up the given `name` into the environment and returns `TRUE` if found. The type of `name` is returned at `t`. If the name is a type name, `is_type_name` is changed to `TRUE`. If it is a variable name, `is_type_name` is `FALSE`.

- `BOOL Lookup(Ptree* name, TypeInfo& t)`
This is an alias of `Lookup(Ptree*, BOOL&, TypeInfo&)` shown above.

- `Class* LookupClassMetaobject(Ptree* name)`
This looks up the given `name` and returns a `Class` metaobject of the type. The `name` may be a variable name or a type name. If the `name` is not found, this function returns `nil`.

`Environment` metaobjects are also useful to store client data. Programmers can record client data in any `Environment` metaobject and look it up later. The client data must be a subclass of `Environment::ClientData`. The following member functions are for manipulating the client data:

- `BOOL AddClientData(Class* metaobject, Ptree* key, Environment::ClientData* data)`
This records `data` in the environment. `metaobject` and `key` are used to retrieve `data` later. If another data is recorded with the same pair of `metaobject` and `key`, this returns `FALSE`.

- `Environment::ClientData* LookupClientData(Class* metaobject, Ptree* key)`
This returns the client data recorded with `metaobject` and `key`. If the data is not found, this returns `nil`.

- `BOOL DeleteClientData(Environment::ClientData* data)`
This deallocates `data` recorded in the environment. If `data` is not found, this returns `FALSE`. (Note that `data` is automatically deallocated when the environment is deallocated.)

`Environment` metaobjects are also used to insert declarations in the translated program. These are member functions for that purpose:

- `void InsertDeclaration(Ptree* decl)`
This inserts the given code `decl` at the beginning of the function body that is currently processed.

- `BOOL InsertDeclaration(Ptree* decl, Class* metaobject, Ptree* key, Environment::ClientData* data)`
This inserts the given code `decl` at the beginning of the function body. The difference from the function above is that it also records the client data in the outermost environment of the function body. It guarantees that the recorded data last while the function body is translated. Note that `AddClientData()` records the client data in the immediate environment (aka the innermost block). The client data is not visible out of that environment.

To insert code in the translated program, also see `InsertBeforeToplevel()` and `AppendAfterToplevel()` in Section 4.4.7.

4.4 Class Metaobjects

Class metaobjects play the key role of the MOP. They represent class definitions and also control source-to-source translation of the program. Their default class is `Class`, but programmers may define a subclass of `Class` to control the source-code translation.

The class of a metaobject is selected by the `metaclass` declaration at the base level. If the metaclass for `Point` is `PersistentClass` at the base level, then the class metaobject for `Point` is an instance of `PersistentClass`. This semantics is natural because a metaclass means the class of a class.

4.4.1 Selecting a Metaclass

Base-level programmers may specify a metaclass in a way other than the `metaclass` declaration. The exact algorithm to select a metaclass (that is, the class of a class metaobject) is as shown below:

1. The metaclass specified by the `metaclass` declaration.

2. The metaclass specified by the keyword attached to the class declaration if exists (See `RegisterMetaclass()` in Section 4.4.6 for more details).

3. Or else, the metaclass for the base classes. If the metaclass of each base class is different, an error is caused.

4. Otherwise, the default `Class` is selected.

If both the metaclass declaration and the keyword exist and they specify different metaclasses, then an error is caused.

4.4.2 Constructor

Class metaobjects may receive meta arguments from the base level when they are initialized. The constructor is responsible to deal with the meta arguments. By default, the meta arguments are simply ignored. Here is the prototype of the constructor:

• `Class(Ptree* definition, Ptree* meta_args)`
This constructor initializes the data members. `definition` is the whole piece of code of the class declaration. If meta arguments are not given, `meta_args` is `nil`.

4.4.3 Protocol for Introspection

Since a class metaobject is the meta representation of a class, programmers can access details of the class definition through the class metaobject. The followings are member functions on class metaobjects. The subclasses of `Class` cannot override them.

- `Ptree* Name()`

This returns the name field of the class declaration.

- `Ptree* BaseClasses()`

This returns the base-classes field of the class declaration. For example, if the class declaration is:

```
class C : public A, private B { ... };
```

Then, `BaseClasses()` returns:

```
⟦: ⟦public A⟧ , ⟦private B⟧⟧
```

- `Ptree* Members()`

This returns the body of the class declaration. It is a list of member declarations. It does not include `{` and `}`.

- `Ptree* Definition()`

This returns the whole of the class declaration. It is the `Ptree` metaobject passed to the constructor as `definition`.

- `char* MetaclassName()`

This returns the name of the metaclass.

- `Class* NthBaseClass(Environment* env, int n)`

This returns the `n`-th base class.

- `Ptree* NthMemberName(int n)`

This returns the name of the `n`-th member (including data members and member functions). It returns `nil` if `n`-th member does not exist. `n` must be more than zero.

- `int IsMember(Ptree* name)`

This returns `0` (`FALSE`) if `name` is not a member of the class. Otherwise, it returns where the member is. `IsMember(NthMemberName(`n`))` is equal to n.

- `BOOL LookupMemberType(Environment* env, Ptree* name, TypeInfo& t)`

This looks up the type of the member specified by `name`. The found type is stored in `t`. The function returns `FALSE` if the member is not found.

4.4.4 Protocol for Translation

Class metaobjects controls source-to-source translation of the program. Expressions involving a class are translated from OpenC++ to C++ by a member function on the class metaobject.[4] Programmers may define a subclass of `Class` to override such a member function to tailor the translation.

The effective class metaobject that is actually responsible for the translation is the *static* type of the object involved by the expression. For example, suppose:

```
class Point { public: int x, y; };
class ColoredPoint : public Point { public: int color; };
        :
Point* p = new ColoredPoint;
```

Then, an expression for data member read, `p->x`, is translated by the class metaobject for `Point` because the variable `p` is a pointer to not `ColoredPoint` but `Point`. Although this might seem wrong design, it is a right way since only static type analysis is available for compile-time MOPs.

The `virtual` member functions on `Class` shown below control source-to-source translation on each kind of expression. They take an environment and an expression, then returns a translated expression. All of them are overridable.

- `Ptree* TranslateClassName(Environment* env, Ptree* keyword, Ptree* name)`

This translates the `name` of the class. All occurences of the class name in the program are replaced with the returned name. `keyword` specifies a user-defined keyword (type modifier) attached to the class name. If none, `keyword` is `nil`.

[4] In the current version, the translated code is not recursively translated again. So the metaobjects have to translate code from OpenC++ to C++ rather than from OpenC++ to (less-extended) OpenC++. This limitation will be fixed in future.

— *Default implementation by* `Class`
This does not change the name but it just returns `name`. If `keyword` is not `nil`, it causes an error.

• `Ptree* TranslateSelf(Environment* env)`
This translates the declaration of the class. The declaration is not passed as an argument because it is available by `Definition()`.

— *Default implementation by* `Class`
This follows a layered subprotocol. It calls `TranslateClassName()`, `TranslateBaseClasses()`, and `TranslateBody()` so that class names in the declaration are correctly renamed.

• `Ptree* TranslateBaseClasses(Environment* env, Ptree* bases)`
This translates the base-classes field of class declaration.

— *Default implementation by* `Class`
This calls `TranslateClassName()` on each base class. A subclass of `Class` can call this with a modified `bases`. For example, it may append another base class to `bases`.

• `Ptree* TranslateBody(Environment* env, Ptree* body)`
This translates the body of class declaration. `body` has been already translated except user-defined access specifiers so that all the class names are renamed. This function can only append member functions written in C++ to `body`. The class names appearing in the appended member functions must be already renamed. This function also has to properly process user-defined access specifiers.

— *Default implementation by* `Class`
This returns `body`.

• `Ptree* TranslateMemberFunctionBody(Environment* env,`
`Ptree* name, BOOL inlined, Ptree* body)`
This translates the body of a member function. `name` is the name of the member function. `inlined` is `TRUE` if the member function is inlined. `body` is statements surrounded by `{}`. The braces are not included by `body`. `env` contains the member name and the argument names.

— *Default implementation by* `Class`
This translates `body` by calling `TranslateStatement()` on each statement.

• `Ptree* TranslateInitializer(Environment* env, Ptree* expr, PtreeArray& before, PtreeArray& after)`

This translates a variable initializer `expr`, which would be "⟦`=` *expression*⟧" or "⟦`(` ⟦*expression*⟧ `)`⟧". The two forms correspond to C++'s two different notations for initialization. For example:

```
complex z(2.3, 4.0);
complex z = 0.0;
```

The `Ptree` objects stored in `before` and `after` are inserted before and after the declaration including the variable initializer. This is useful to implement, for example, a translation from

```
complex z = {2.3, 4.0};
```

into:

```
static double tmp[] = {2.3, 4.0};
complex z(tmp);
```

To store a `Ptree` object in an `PtreeArray` object, `Append(Ptree* list)` on `PtreeArray` is available.

— *Default implementation by* `Class`

This translates `expr` by calling `TranslateExpression()` on the initializing parameter (the second element of `expr`) and returns the result.

• `Ptree* TranslateAssign(Environment* env, Ptree* object, Ptree* assign_op, Ptree* expr)`

This translates an assignment expression such as `=` and `+=`. `object` is an instance of the class, which `expr` is assigned to. `assign_op` is an assignment operator. `object` and `expr` have not been translated yet.

— *Default implementation by* `Class`

This calls `TranslateExpression()` on `object` and `expr` and returns the translated expression.

• `Ptree* TranslateSubscript(Environment* env, Ptree* object, Ptree* index)`

This translates a subscript expression (array access). `object` is an instance of the class, which the operator `[]` denoted by `index` is applied to. `index` is a list "⟦`[` *expression* `]`⟧". `object` and `expr` have not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and `index` and returns the translated expression.

- `Ptree* TranslateFunctionCall(Environment* env, Ptree* object, Ptree* args)`

This translates a function call expression on `object`. Note that it is not for translating a member function call. It is invoked to translate an application of the the `()` operator. `object` is an instance of the class. `object` and `args` have not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and `args` and returns the translated expression.

- `Ptree* TranslatePostfix(Environment* env, Ptree* object, Ptree* post_op)`

This translates a postfix increment or decrement expression (`++` or `--`). `object` is an instance of the class, which the operator `post_op` is applied to. `object` has not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and returns the translated expression.

- `Ptree* TranslateUnary(Environment* env, Ptree* unary_op, Ptree* object)`

This translates a unary expression. `unary_op` is the operator, which are either `*`, `&`, `+`, `-`, `!`, `~`, `++`, or `--`. `sizeof` is not included. `object` is an instance of the class, which the operator is applied to. `object` has not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and returns the translated expression.

- `Ptree* TranslateBinary(Environment* env, Ptree* lexpr, Ptree* binary_op, Ptree* rexpr)`

This translates a binary expression. `binary_op` is the operator such as `*`, `+`, `<<`, `==`, `|`, `&&`, and `,` (comma). `lexpr` and `rexpr` are the left-side expression and the right-side expression. They have not been translated yet. The type of `rexpr` is the class.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `lexpr` and `rexpr` and returns the translated expression.

• `Ptree* TranslateNew(Environment* env, Ptree* header,`
`Ptree* new_op, Ptree* placement,`
`Ptree* type_name, Ptree* arglist)`
This translates a `new` expression. `header` is a user-defined keyword (type modifier), `::` (if the expression is `::new`), or `nil`. `new_op` denotes the lexical token `new`. `type_name` may include an array size surrounded by `[]`. `arglist` is arguments to the constructor. It includes parentheses `()`. `placement`, `tname`, and `arglist` have not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `placement` and `arglist`, and `TranslateNewType()` on `type_name`. Then it returns the translated expression.

• `Ptree* TranslateMemberRead(Environment* env, Ptree* object,`
`Ptree* op, Ptree* member)`
This translates a member read expression on the `object`. The operator `op` is `.` (dot) or `->`. `member` specifies the member name. `object` has not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on the `object` and returns the translated expression.

• `Ptree* TranslateMemberRead(Environment* env, Ptree* member)`
This translates a member read expression on the `this` object.

— *Default implementation by* `Class`
This returns `member`.

• `Ptree* TranslateMemberWrite(Environment* env, Ptree* object,`
`Ptree* op, Ptree* member, Ptree* assign_op, Ptree* expr)`
This translates a member write expression on the `object`. The operator `op` is `.` (dot) or `->`. `member` specifies the member name. `assign_op` is an assign operator such as `=` and `+=`. `expr` specifies the right-hand expression of the assign operator. `object` and `expr` have not been translated yet.

— *Default implementation by* `Class`

This calls `TranslateExpression()` on `object` and `expr` and returns the translated expression.

• `Ptree* TranslateMemberWrite(Environment* env, Ptree* member, Ptree* assign_op, Ptree* expr)`
This translates a member write expression on the `this` object. `member` specifies the member name. `assign_op` is an assign operator such as `=` and `+=`. `expr` specifies the right-hand expression of the assign operator. `expr` has not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `expr` and returns the translated expression.

• `Ptree* TranslateMemberCall(Environment* env, Ptree* object, Ptree* op, Ptree* member, Ptree* arglist)`
This translates a member function call on the `object`. The operator `op` is `.` (dot) or `->`. `member` specifies the member name. `arglist` is arguments to the function. It includes parentheses `()`. `object` and `arglist` have not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object`, and `TranslateArgumentList()` on `arglist`. Then it returns the translated expression.

• `Ptree* TranslateMemberCall(Environment* env, Ptree* member, Ptree* arglist)`
This translates a member function call on the `this` object. `member` specifies the member name. `arglist` is arguments to the function. It includes parentheses `()`. `arglist` has not been translated yet.

— *Default implementation by* `Class`
This calls `TranslateArgumentList()` on `arglist` and returns the translated expression.

• `Ptree* TranslateUserStatement(Environment* env, Ptree* object, Ptree* op, Ptree* keyword, Ptree* rest)`
This translates a user-defined statement, which is a while-style, for-style, or closure statement. The first three elements of the statement are specified by `object`, `op`, and `keyword`. The rest of the statement, the `()` part and the `[]` part, is specified by `rest`.

— *Default implementation by* `Class`
This causes an error and returns `nil`.

• `Ptree* TranslateStaticUserStatement(Environment* env,`
`Ptree* keyword, Ptree* rest)`
This translates a user-defined statement beginning with a class name. This is named after that the syntax is similar to one for `static` member function calls. The meaning of the arguments is the same as that of `TranslateUserStatement()`.

The MOP does not allow programmers to customize array access or pointer operations. Suppose that `p` is a pointer to a class `A`. Then the class metaobject for `A` cannot translate expressions such as `*p` or `p[3]`. This design decision is based on C++'s one. For example, C++'s operator overloading on `[]` does not change the meaning of array access. It changes the meaning of the operator `[]` applied to not an array of objects but an object.

If the MOP allows programmers to customize array access and pointer operations, they could implement an inconsistent extension. For example, they want to translate an expression `p[2]` into `p->get(2)`, where `p` is a pointer to a class `X`. Then, what should this expression `*(p + 2)` be translated into? Should the MOP regard it as an array access or a pointer dereference? Because C++ provides strong pointer arithmetic, designing an interface to consistently customize array access and pointer operations is difficult.

4.4.5 Protocol for Initialization and Finalization

The MOP provides functions to initialize and finalize class metaobjects.

• `static BOOL Initialize()`
This is called only once on each metaclass right after the compiler starts. It returns `TRUE` if the initialization succeeds. The subclasses of `Class` may define their own `Initialize()` but they have to call their base classes' `Initialize()`. This function is not overridable.

— *Default implementation by* `Class`
This does nothing except returning `TRUE`.

• `virtual Ptree* Finalize()`
This is called on each class metaobject after all the translation is finished.

The returned `Ptree` object is inserted at the end of the translated source file.

— *Default implementation by* `Class`
This returns `nil`.

4.4.6 Protocol for Registering Keywords

To make user-defined keywords available at the base level, programmers must register the keywords by the `static` member functions on `Class` shown below. Those member functions are called within `Initialize()` in Section 4.4.5.

• `void RegisterNewModifier(char* keyword)`
This registers `keyword` as a new type modifier.

• `void RegisterNewAccessSpecifier(char* keyword)`
This registers `keyword` as a new access specifier.

• `void RegisterNewWhileStatement(char* keyword)`
This registers `keyword` as a new `while` statement.

• `void RegisterNewForStatement(char* keyword)`
This registers `keyword` as a new `for` statement.

• `void RegisterNewClosureStatement(char* keyword)`
This registers `keyword` as a new closure statement.

• `void RegisterMetaclass(char* keyword, char* metaclass)`
This registers `keyword` as a new type modifier and associates it with `metaclass`. If this keyword appears in front of a class declaration, then `metaclass` is selected for the declared class. See also Section 4.4.1.

4.4.7 Service Functions

These are also `static` member functions on `Class`:

• `Ptree* TranslateNewType(Environment* env, Ptree* type_name)`
This translates the `type_name` field of `new` expressions.

• `Ptree* TranslateArgumentList(Environment* env, Ptree* arglist)`

This translates an argument list.

• `Ptree* TranslateIndex(Environment* env, Ptree* index)`
This translates the `index` field of subscript expressions.

• `Ptree* TranslateExpression(Environment* env, Ptree* expr)`
This translates an expression.

• `Ptree* TranslateExpression(Environment* env,`
`Ptree* expr, TypeInfo& t)`
This translates an expression and stores its type in `t`.

• `Ptree* TranslateStatement(Environment* env, Ptree* expr)`
This translates a statement, including a block statement surrounded by `{}`.

The next two member functions insert given code into the translated source code. As for insertion, also see `InsertDeclaration()` in Section 4.3.

• `void InsertBeforeToplevel(Ptree* list)`
This inserts `list` before the toplevel declaration, such as function definitions, that are currently translated.

• `void AppendAfterToplevel(Ptree* list)`
This appends `list` after the toplevel declaration, such as function definitions, that are currently translated.

4.4.8 Template Class

Metaclasses for template classes must be `TemplateClass` or its subclass programmers define. `TemplateClass` is a subclass of `Class` and it defines the following member functions for introspection:

• `Ptree* TemplateDefinition()`
This returns the whole definition of the template, including the keyword `template` and the template arguments.

• `Ptree* TemplateArguments()`
This returns the template argument.

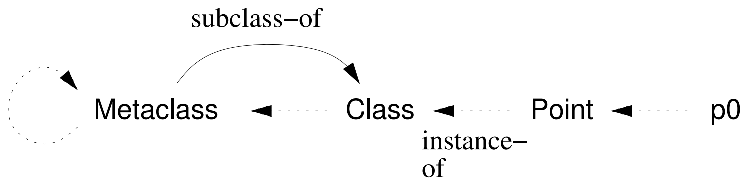

Figure 2: Instance-of Relationship

4.4.9 Metaclass for `Class`

Since OpenC++ is a self-reflective language, the meta-level programs are also in OpenC++. They must be compiled by the OpenC++ compiler. Because of this self-reflection, metaclasses also have their metaclasses. The metaclass for `Class` and its subclasses must be `Metaclass`. However, programmers do not have to explicitly declare the metaclass for their metaclasses because the subclasses of `Class` inherit the metaclass from `Class`.

`Metaclass` makes it easy to define a subclass of `Class`. It automatically inserts the definition of `MetaclassName()` of that subclass and also generates house-keeping code internally used by the compiler.

Since `Metaclass` is a subclass of `Class`, its metaclass is `Metaclass` itself. This relationship is illustrated in Figure 2.

4.5 Error Message

The following functions reports an error that occurs during the source-to-source translation.

- `void ErrorMessage(char* message, Ptree* what = nil, Ptree* where = nil)`

This displays an error message "*message* `"`*what*`" in` *where*".

- `void WarningMessage(char* message, Ptree* what = nil, Ptree* where = nil)`

This displays a waning message "*message* `"`*what*`" in` *where*".

4.6 C++ Preprocessing

The OpenC++ programs are first preprocessed by the C++ preprocessor. During processing, the macro `__opencxx` is predefined.

5 Command Reference

NAME

`occ` — the Open C++ compiler

SYNOPSIS

`occ [-l] [-s] [-v] [-c] [-E] [-I`*include_directory*`]`
`[-D`*name*`[=`*def*`]] [--` *C++ compiler options*`]` *source_file*

DESCRIPTION

`occ` compiles an OpenC++ program into an object file. It first invokes the C++ preprocessor and generates a `.occ` file, then translates it into a `.ii` file according to meta-level code. The `.ii` file is compiled by the back-end C++ compiler, and finally an `a.out` file is produced. If `occ` is run with the `-c` option, it generates a `.o` file but suppresses linking.

OPTIONS

`-D` Define a macro *name* as *def*.

`-E` Don't run the back-end C++ compiler. Stop after generating a `.ii` file.

`-I` Add a *directory* to the search path of the `#include` directive.

`-c` Suppress linking and produce a `.o` file.

`-l` Print the list of loaded metaclasses.

`-s` Print the whole parse tree of the given source program. Don't perform translation or compilation. If no source file is given, `occ` reads from the standard input.

`-v` Specify the verbose mode.

`--` Following options are interpreted as options for the back-end C++ compiler. For example, if you type

```
occ -I.. -- -g foo.c
```

Then the `-g` option is passed to the C++ compiler.

FILES

`file.cc`	source file.
`file.occ`	output file after C++ preprocessing.
`file.ii`	output file after translation.

`file.o`	object file.
`opencxx.a`	library to link with meta-level code.

NOTES

- When the C++ processor runs, the macro `__opencxx` is predefined.
- The current version of the compiler cannot dynamically load meta-level programs on demand.

COPYRIGHT

AUTHOR

Shigeru Chiba, Xerox PARC and The University of Tokyo.

Contents

1 Introduction **1**

2 Tutorial **2**
2.1 Verbose Objects 3
2.1.1 What the base-level program should look like 3
2.1.2 What the base-level program should be translated . . 4
2.1.3 Write a meta-level program 4
2.1.4 Compile, debug, and run 6
2.2 Syntax Extension for Verbose Objects 7
2.3 Matrix Library 8
2.3.1 What the base-level program should look like 9
2.3.2 What the base-level program should be translated . . 9
2.3.3 Write a meta-level program 10
2.3.4 Write runtime support code 12
2.4 Syntax Extension for the Matrix Library 13
2.5 Before-Method 15
2.5.1 What the base-level program should look like 15
2.5.2 What the base-level program should be translated . . 16
2.5.3 Write a meta-level program 16
2.6 Wrapper Function 18
2.6.1 What the base-level program should be translated . . 18
2.6.2 Write a meta-level program 19
2.6.3 Subclass of `WrapperClass` 22

3 Base-Level Language (OpenC++) **24**
3.1 Base-level Connection to the MOP 24
3.2 Syntax Extensions 24
3.3 Loosened Grammar 26

4 Metaobject Protocol (MOP) **27**
4.1 Representation of Program Text 27
4.1.1 Basic Operations 29
4.1.2 Construction 31
4.1.3 Pattern Matching 33
4.1.4 Reifying Program Text 34
4.1.5 Convenience Classes `PtreeIter` and `PtreeArray` . . . 34
4.2 Representation of Types 35
4.3 Representation of Environment 36

4.4 Class Metaobjects 38
4.4.1 Selecting a Metaclass 38
4.4.2 Constructor 38
4.4.3 Protocol for Introspection 39
4.4.4 Protocol for Translation 40
4.4.5 Protocol for Initialization and Finalization 46
4.4.6 Protocol for Registering Keywords 47
4.4.7 Service Functions 47
4.4.8 Template Class 48
4.4.9 Metaclass for `Class` 49
4.5 Error Message 49
4.6 C++ Preprocessing 49

5 Command Reference 50