

Chapter 13

Reflection

This chapter describes reflection in the Java and C# programming languages. Reflection permits a running program to inspect and manipulate the classes, methods, fields and so on that make up the program. For instance, the program may obtain a list of all methods in a class, or all those methods whose names begin with the string "test". A method description `mo` thus obtained may be called using a reflective method call such as `mo.invoke(...)`. Similarly, a class description obtained this way may be used to create an object of the class.

13.1 What files are provided for this chapter

A number of example programs (in Java and C#) are used to illustrate runtime reflection:

Java example	C# example	Contents
<code>rtcg/Reflect0.java</code>	<code>rtcg/Reflect0.cs</code>	get types reflectively
<code>rtcg/Reflect1.java</code>	<code>rtcg/Reflect1.cs</code>	call method reflectively
<code>rtcg/Reflect2.java</code>	<code>rtcg/Reflect2.cs</code>	call methods reflectively
<code>rtcg/Reflect3.java</code>	<code>rtcg/Reflect3.cs</code>	measure reflective call speed

The example `Reflect0.java` (and `Reflect0.cs`) shows that for every type, and in particular, for every class, there is an object representing that type. The object has class `java.lang.Class` in Java (and class `System.Type` in C#).

The example `Reflect1.java` (and `Reflect1.cs`) shows how the object `co` corresponding to a class can be used to obtain an object `mo` representing a public method from that class:

```
import java.lang.reflect.*;                                // Method
```

```

class Reflect1 {
    public static void main(String[] args)
        throws NoSuchMethodException, IllegalAccessException,
            InvocationTargetException {
        Class<Reflect1> co = Reflect1.class;           // Get Reflect1 class
        Method mo = co.getMethod("Foo", new Class[] {}); // Get Foo() method
        mo.invoke(null, new Object[] { });           // Call it
    }

    public static void Foo() {
        System.out.println("Foo was called");
    }
}

```

The `mo` object has class `Method` in Java (and class `MethodInfo` in C#); see also Figure 13.1. Objects representing fields and constructors can be obtained similarly. The example also shows how the object `mo` can be used to call (invoke) the method `Foo` represented by `mo`. In general, the arguments to a method called by reflection must be passed in an array of `Objects`, so values of primitive type must be boxed (wrapped as objects), and an array must be allocated to hold the argument values. Similarly, the result of the method is returned as an object, which must then be unboxed (unwrapped) if it is a primitive type value.

The example `Reflect2.java` (and `Reflect2.cs`) shows one way to find methods that satisfy particular criteria. In this case all methods of a class are obtained, and every method that is static and whose name begins with the string "Test" is invoked with an empty argument list (this will cause a runtime error if the method does require arguments):

```

Method[] mos = co.getMethods();
System.out.println("These static methods are available:");
for (int i=0; i<mos.length; i++)
    if (Modifier.isStatic(mos[i].getModifiers()))
        System.out.println(mos[i].getName());
System.out.println();
System.out.println("Calling static methods whose names start with Test:");
for (int i=0; i<mos.length; i++)
    if (Modifier.isStatic(mos[i].getModifiers())
        && mos[i].getName().indexOf("Test") == 0)
        mos[i].invoke(null, new Object[] {});

```

13.2 Reflection mechanisms in Java and C#

The Java and C# reflection mechanisms are remarkably similar, as shown in Figure 13.1. In relation to reflection on generic types and methods the mechanisms differ considerably, though, because the CLI/.NET runtime system underlying C# supports generic types at runtime, and the Java Virtual Machine does not; see also Section 9.5.

	Java	C#
Class description	<code>java.lang.Class</code>	<code>System.Type</code>
Class object for class <code>C</code>	<code>C.class</code>	<code>typeof(C)</code>
One method in class <code>co</code>	<code>co.getMethod(...)</code>	<code>co.GetMethod(...)</code>
Public methods in <code>co</code>	<code>co.getMethods()</code>	<code>co.GetMethods()</code>
One field in class <code>co</code>	<code>co.getField(...)</code>	<code>co.GetField(...)</code>
Public fields in <code>co</code>	<code>co.getFields()</code>	<code>co.GetFields(...)</code>
One constructor in <code>co</code>	<code>co.getConstructor(...)</code>	<code>co.GetConstructor(...)</code>
Public constructors in <code>co</code>	<code>co.getConstructors()</code>	<code>co.GetConstructors()</code>
Reflection API	<code>java.lang.reflect</code>	<code>System.Reflection</code>
Method description	<code>Method</code>	<code>MethodInfo</code>
Field description	<code>Field</code>	<code>FieldInfo</code>
Constructor description	<code>Constructor</code>	<code>ConstructorInfo</code>
Call method	<code>mo.invoke(...)</code>	<code>mo.Invoke(...)</code>
Get field's value	<code>fo.get(...)</code>	<code>fo.GetValue(...)</code>
Set field's value	<code>fo.set(...)</code>	<code>fo.SetValue(...)</code>
Call constructor	<code>cco.newInstance(...)</code>	<code>cco.Invoke(...)</code>
Is generic type definition	<code>co is ParameterizedType</code>	<code>mo.IsGenericTypeDefinition</code>
Is generic type instance	(not possible)	<code>co.IsGenericType</code>
Is a type parameter	<code>co is TypeVariable</code>	<code>co.IsGenericParameter</code>
Get type parameters	<code>co.getTypeParameters()</code>	<code>co.GetGenericArguments()</code>
Get type arguments	<code>po.getActualTypeArguments()</code>	<code>co.GetGenericArguments()</code>
Create type instance	(not possible)	<code>co.MakeGenericType(...)</code>
Is generic mth definition	(not possible)	<code>co.IsGenericMethodDefinition</code>
Is generic mth instance	(not possible)	<code>mo.IsGenericMethod</code>
Get type parameters	<code>mo.getTypeParameters()</code>	<code>mo.GetGenericArguments()</code>
Get type arguments	(not possible)	<code>mo.GetGenericArguments()</code>
Create method instance	(not possible)	<code>mo.MakeGenericMethod(...)</code>

Figure 13.1: Java and C# reflection mechanisms. Here `co` represents a class or type, `mo` a method, `fo` a field, and `cco` a constructor.

Reflective method calls and field accesses provide extra flexibility by allowing a program to perform some 'introspection'. For instance, one can write a

general test framework that uses reflection to call those methods of a given class `C` whose names begin with the string `"Test"`; see `rtcg/Reflect2.java`. This is how the unit testing frameworks `jUnit` and `nUnit` are implemented.

Basically, the added flexibility of reflection derives from its ability to postpone some program decisions (which methods to call, and so on) till execution time.

Reflection has some drawbacks, too:

- types are checked only at runtime, so some compile-time type safety is lost;
- reflective method calls, field accesses, etc. are much slower than ordinary compiled method calls, field accesses, etc.

Reflective method calls are inefficient because of the required wrapping of arguments as `Object` arrays, and because access checks and so on must be performed at runtime. In a highly optimizing virtual machine Sun's HotSpot JVM 1.6.0, a reflective method call is typically slower than a static or virtual method call by a factor of approximately 13. Moreover, if the class containing the method is not public, additional runtime checks for method accessibility make the reflective call slower by a further factor of 5. The slowdown caused by reflection seems to be even more dramatic in Microsoft's CLI/.NET implementation (version 3.5), where a reflective call to a static method is 195–250 times slower than a normal call.

In C# the argument wrapping, result unwrapping, and access checks costs can be reduced by turning a `MethodInfo` object into a C# delegate. Calling the delegate is more than 30 times faster than a reflective method call, because method lookup, type checks and access checks are performed once and for all when the delegate is created, not at every call to it. Java does not provide an out-of-the-box mechanism with the same efficiency, but a similar effect can be obtained by defining a suitable interface and using runtime code generation as shown in Section 14.8.

13.3 History and literature

A reflection mechanism has been available in Lisp since 1960 in the form of *fexprs*. Brian Cantwell Smith introduced a 'reflective tower' as the concept of an infinite tower of Lisp interpreters, one interpreter executing the code of the interpreter below it [119].

Friedman and Wand [46] gave a simpler, more concrete version of Smith's ideas. Furthermore, they introduced a distinction between reification and re-

flection: 'We will use the term *reification* to mean the conversion of an interpreter component into an object which the program can manipulate. One can think of this transformation as converting a program (*i.e.* the expression being evaluated) into data. We will use the term *reflection* to mean the operation of taking a program-manipulable value and installing it as a component in the interpreter. This is thus a transformation from data to program.' As can be seen, these definitions do not agree entirely with the meaning of reflection in Java and C#. Friedman and Wand's notion reification seems similar to reflection in Java and C#, whereas their notion of reflection seems related to runtime code generation, the subject of the next chapter.

Danvy and Malmkjær presented a simpler reflective tower [34].

Chapter 14

Runtime code generation

Program specialization generates code that is specialized to particular values of one or more input parameters. Runtime code generation may be used to perform program specialization at runtime. Properly used, this can lead to considerable speedups.

Both Java/JVM and C#/CLI support runtime code generation. In C#/CLI runtime code generation is supported through the .Net Framework (see Section 14.4). In Java/JVM one can use third-party libraries to generate bytecode at runtime and use a Java class loader to dynamically load this code. In both cases the generated code can be executed via reflection (Chapter 13), or more efficiently via delegate calls in C# and interface method calls in Java.

14.1 What files are provided for this chapter

Java example	C# example	Contents
<code>rtcg/Power.java</code>	<code>rtcg/Power.cs</code>	specialize power function as source
	<code>rtcg/RTCG1D.cs</code>	generate argumentless method
<code>rtcg/RTCG2.java</code>	<code>rtcg/RTCG2D.cs</code>	generate method with argument
<code>rtcg/RTCG3.java</code>	<code>rtcg/RTCG3D.cs</code>	generate methods with loops
<code>rtcg/RTCG4.java</code>	<code>rtcg/RTCG4D.cs</code>	generate specialized power function
	<code>rtcg/RTCG5D.cs</code>	generate fast polynomial evaluator
	<code>rtcg/RTCG6D.cs</code>	generate fast expression evaluator
<code>rtcg/RTCG7.java</code>	<code>rtcg/RTCG7D.cs</code>	measure code generation time
<code>rtcg/RTCG8.java</code>	<code>rtcg/RTCG8D.cs</code>	generate sparse matrix multiplier

In addition there are some supporting Java interfaces: `rtcg/IMyInterface.java`, `rtcg/Int2Int.java`, and `rtcg/ISparseMult.java`.

A large example of runtime code generation in C# can be found in the .Net Framework implementation of regular expression matching. The source code can be studied in Microsoft's 'shared source' release of CLI [88], in file `sscli/fix/src/regex/system/text/regexexpressions/regexcompiler.cs`.

14.2 Program specialization

Consider a Java method `Power(n, x)` which computes x^n , that is, x raised to the n 'th power:

```
public static int Power(int n, int x) {
    int p;
    p = 1;
    while (n > 0) {
        if (n % 2 == 0)
            { x = x * x; n = n / 2; }
        else
            { p = p * x; n = n - 1; }
    }
    return p;
}
```

The two branches of the `if` statement in `Power` rely on these equalities, for n even ($n = 2m$) and n odd ($n = 2m + 1$):

$$\begin{aligned} x^{2m} &= (x^2)^m \\ x^{2m+1} &= x^{2m} \cdot x \end{aligned}$$

Assume that the value of parameter n is known, whereas the value of x is not. Then one can perform the tests in `while` and `if`, and one can perform other computations that depend on n only. However, one cannot perform any computations that depend on x . Instead one could generate code that will perform those computations at a later point, when x is known. That code can be packaged as a method `Power_5(int x)` which will compute x^5 when called on x . Specializing, or partially evaluating, `Power` for n being 5 might generate this specialized method:

```
public int Power_5(int x) {
    int p;
    p = 1;
    p = p * x;
    x = x * x;
    x = x * x;
```

```

    p = p * x;
    return p;
}

```

One can quite easily write a method `PowerTextGen(int n)`, which for a given `n` will generate a version `Power_n` of `Power` that is specialized for that value of `n`. Such a method is called a *generating extension* for `Power`, and could be written like this:

```

public static void PowerTextGen(int n) {
    System.out.println("public int Power_" + n + "(int x) {");
    System.out.println("    int p;");
    System.out.println("    p = 1;");
    while (n > 0) {
        if (n % 2 == 0) {
            System.out.println("    x = x * x;");
            n = n / 2;
        } else {
            System.out.println("    p = p * x;");
            n = n - 1;
        }
    }
    System.out.println("    return p;");
    System.out.println("}");
}

```

In fact, calling `PowerTextGen(5)` will produce exactly the method `Power_5` shown above.

Note the close structural similarity between `Power` and `PowerTextGen`. It arises because `PowerTextGen` is obtained from `Power` by classifying those parts of `Power` depending only on `n` as *static* (to be executed early), and those parts depending on `x` as *dynamic* (to be executed late). The `PowerTextGen` method simply executes the static parts and generates code for the dynamic parts. The classification of program code and variables into static and dynamic may be performed by a so-called *binding-time analysis*. Binding-time analysis is an important step in automatic *program specialization*, also known as *partial evaluation* [63, 93].

However, generating a specialized program in the form of Java or C# source code is suitable only when the value of `n` is known well in advance of runtime. Namely, the generated specialized program must be compiled, which is quite resource demanding. This precludes program specialization at runtime, when it is more likely that the value of variables such as `n` are available for program specialization.

14.3 Quasiquote and two-level languages

Before we embark on runtime bytecode generation in section 14.4, we shall first see how easy runtime code generation can be when the language has adequate facilities for it.

14.3.1 The Scheme programming language

In the dynamically typed functional language Scheme [70, 127], it is particularly easy to express program generation, thanks to two factors: It is easy to write Scheme program fragments in abstract syntax within Scheme itself, and the so-called quasiquote mechanism is ideal for writing programs that generate other programs.

The Scheme syntax is particularly simple. Consider the function f that computes x times 3 plus 7, which would be written like this in Standard ML:

```
fun f x = x * 3 + 7
```

would look like this in Scheme:

```
(define (f x) (+ (* x 3) 7))
```

Thus the Scheme keyword `define` introduces a definition, f is the name of the defined function, x is its parameter, and `(+ (* x 3) 7)` is the function body. Any program in Scheme is written as a list of nested lists, each list having a keyword or operator as its first element. Thus the function declaration is a list with three elements, where the second element is a list `(f x)` of the names of the function and its parameters. The third element of the function declaration is the function body, which in this case is the arithmetic expression `(+ (* x 3) 7)`. In Scheme, all expressions, including arithmetic expressions, are written in prefix notation, with the operator before the operands.

To define the function f , start a Scheme system such as `scm` [59] and type in the function definition. The function gets defined inside the Scheme system, and the value `#<unspecified>` is returned:

```
> (define (f x) (+ (* x 3) 7))
#<unspecified>
```

The user's entry is written after the prompt (`>`), and the system's response is written on the next line. To apply function f to the argument 10, write a function application, which is a list with the function as operator and 10 as sole argument:

```
> (f 10)
37
```

In Scheme, a list of constants such as 11 22 33 is written '(11 22 33) where the quote operator (') denotes a constant. The define keyword is used for functions as well as variables, so we bind variable cs to the list 11 22 33 as follows:

```
> (define cs '(11 22 33))
#<unspecified>
```

The car function returns the head, or first element, of a list:

```
> (car cs)
11
```

The cdr function returns the tail, or all elements but the first one:

```
> (cdr cs)
(22 33)
```

The null? function tests whether a list is empty; the symbol #f obviously means false:

```
> (null? cs)
#f
```

The list function is used to construct a list from general expressions:

```
> (list (car (cdr cs)) '11 (+ 3 30))
(22 11 33)
```

Scheme's conditional expression (if e1 e2 e3) corresponds to if e1 then e2 else e3 in Standard ML, and to e1 ? e2 : e3 in Java or C#.

14.3.2 Recursive functions in Scheme

The Scheme analog of the Java Power function shown at the beginning of section 14.2 can be defined like this, using an auxiliary function sqr to compute x^2 :

```
(define (sqr x) (* x x))
(define (powr n x)
  (if (> n 0)
      (if (eq? (remainder n 2) 0)
```

234 *Quasiquote and two-level languages*

```
(sqr (powr (/ n 2) x))
(* x (powr (- n 1) x))
)
1)
)
```

The remainder function computes the integer remainder from division, and the operators (>), (/) and (-) compute greater-than, division and subtraction. Then we can compute 2^{10} and 3^{97} like this:

```
> (powr 10 2)
1024
> (powr 97 3)
19088056323407827075424486287615602692670648963
```

Suppose we want to compute the value of a polynomial:

$$p(x) = cs[0] + cs[1] \cdot x + cs[2] \cdot x^2 + \dots + cs[n] \cdot x^n$$

for a given coefficient array *cs*. This can be done conveniently using Horner's rule:

$$p(x) = cs[0] + x \cdot (cs[1] + x \cdot (\dots (cs[n] + 0) \dots))$$

A recursive function to evaluate a polynomial with coefficient list *cs* at a given point *x* can be written as follows:

```
(define (poly cs x)
  (if (null? cs)
      0
      (+ (car cs) (* x (poly (cdr cs) x))))
) )
```

Let's compute $11 + 22x + 33x^2$ for $x = 10$:

```
> (poly cs 10)
3531
```

14.3.3 Quote and eval in Scheme

Note the subtle difference between $(+ 2 3)$, which is an expression whose value is 5, and $'(+ 2 3)$, which is a constant whose value is the three-element list containing the plus symbol (+), the constant 2, and the constant 3:

```
> (+ 2 3)
5
> '(+ 2 3)
(+ 2 3)
```

Function `eval` takes as argument a list that has the form of an expression, and evaluates that expression; in a sense, `eval` is an inverse of `quote`:

```
> (eval '(+ 2 3))
5
```

This works also if the expression-like list contains a free variable:

```
> (define myexpr '(+ (* x 3) 7))
#<unspecified>
> myexpr
(+ (* x 3) 7)
> (define x 10)
#<unspecified>
> (eval myexpr)
37
```

Using `eval` on a list that has the form of a function declaration, will define the function inside the Scheme system. The following defines `f` to be the function that adds 10 to its argument:

```
> (eval '(define (f x) (+ x 10)))
#<unspecified>
> (f 7)
17
```

Using a combination of the `list` function and the `quote` (`'`) operator one can write programs that construct new functions, and then define them using `eval`. This provides a way to do runtime code generation in Scheme.

14.3.4 Backquote and comma in Scheme

It quickly becomes rather cumbersome to use `list` and `quote` to construct new expressions and function declarations. Instead, one can use the *backquote* operator (```), also called *quasiquote*, and the *comma* operator (`,`), also called the *unquote* operator. The backquote operator (```) works the same as the quote operator (`'`), except that inside a backquoted expression, one can use the comma operator (`,`) to insert the value of a computed expression.

For example, let us define `e` to be the list `(+ 3 x)`, and then try to use `unquote` `e` both inside an ordinary quoted list and inside a backquoted list:

236 *Quasiquote and two-level languages*

```
> (define e '(* x 3))
#<unspecified>
> '(+ ,e 7)
(+ (unquote e) 7)
> '(+ ,e 7)
(+ (* x 3) 7)
```

The effect of using `unquote e` inside a backquoted list is to insert the value of variable `e` at that point in the list. This is precisely equivalent to an expression using the `list` function and quoted expressions:

```
> (list '+ e '7)
(+ (* x 3) 7)
```

In fact, the unquoted expression need not be a variable such as `e`, but can be any expression which may even contain further backquotes and commas. For instance, we can define a function `addsqrgen` which, given an argument `y`, returns a declaration of function `f` that takes an argument `x` and returns $x+y^2$:

```
> (define (addsqrgen y) `(define (f x) (+ x ,( * y y))))
#<unspecified>
```

Applying `addsqrgen` to `7` returns declaration of a function `f` that adds 49 to its argument:

```
> (addsqrgen 7)
(define (f x) (+ x 49))
```

If we use `eval` on the resulting list, the function `f` becomes defined and can be used subsequently:

```
> (eval (addsqrgen 7))
#<unspecified>
> (f 100)
149
```

14.3.5 Program generation with backquote and comma

Using Scheme's backquote and comma notation, one can easily write a generating extension for the functions `poly` and `powr` shown in section 14.3.2.

A generating extension for `poly` is a function that, given a list `cs` of coefficients for a polynomial, produces an expression with a free variable `x`. If this expression is evaluated with `x` bound to a number, then it computes the value of the polynomial for that value of `x`. A generating extension `polygen` for `poly` can be defined like this using backquote and comma:

```
(define (polygen cs)
  (if (null? cs)
      '0
      '(+ ,(car cs) (* x ,(polygen (cdr cs))))))
```

For instance, building the expression to compute polynomial $11 + 22x + 33x^2$ can be done as follows:

```
> (polygen '(11 22 33))
(+ 11 (* x (+ 22 (* x (+ 33 (* x 0)))))
```

The resulting Scheme list corresponds to the expression $11 + x \cdot (22 + x \cdot (33 + x \cdot 0))$, which has the same value as $11 + 22x + 33x^2$. We can bind x and evaluate the expression to see that this works:

```
> (define x 10)
#<unspecified>
> (eval (polygen '(11 22 33)))
3531
```

A generating extension for `powr` is a function that, given a value for n , produces an expression with a free variable x . If this expression is evaluated, then it computes x to the n 'th power.

```
(define (powrgen n)
  (if (> n 0)
      (if (eq? (remainder n 2) 0)
          '(sqr ,(powrgen (/ n 2)))
          '(* x ,(powrgen (- n 1))))
      '1)
  )
```

For example:

```
> (powrgen 97)
(* x (sqr (sqr (sqr (sqr (sqr (* x (sqr (* x 1))))))))))
```

This Scheme list corresponds to the expression $x \cdot ((((((x \cdot (x \cdot 1)^2)^2)^2)^2)^2)^2)$, which has the same value as x^{97} . We can use `eval`, `backquote` and `comma` to define a function `powreval` that computes x^{97} for any x , and then check that it works:

```
> (eval `(define (powreval x) ,(powrgen 97)))
#<unspecified>
> (powreval 3)
19088056323407827075424486287615602692670648963
```

14.3.6 Two-level languages and binding-times

The generating extensions shown above may seem very ingenious, but in fact backquotes and commas can be inserted in a fairly systematic way, by analysing the binding times of variables and expressions. What one obtains is a *two-level* language or *metaprogramming* language, in which one can write both static (or early) code and dynamic (or late) code. Recall from section 14.2 that *static* code will be executed as usual, whereas *dynamic* code will be generated rather than executed. Consider again the `poly` example:

```
(define (poly cs x)
  (if (null? cs)
      0
      (+ (car cs) (* x (poly (cdr cs) x))))
  ) )
```

As above, assume that parameter `cs` is static and that `x` is dynamic. We classify computations as static or dynamic, marking the dynamic code by underlining. For instance, the condition `(null? cs)` in the `if` expression can be static because `cs` is static. The second branch of the `if` expression must be dynamic because it depends on dynamic parameter `x`, whereas the `0` in first branch could be static, but is made dynamic since the second branch is:

```
(define (poly cs x)
  (if (null? cs)
      0
      (+ (car cs) (* x (poly (cdr cs) x)))
  ) )
```

The structure of the above annotated program is as follows. The static `if` expression has two dynamic branches. The second branch consists of a dynamic addition operator with a static first operand `(car cs)` and a dynamic second operand, which is a dynamic multiplication applied to dynamic operand `x` and a static call to function `poly`.

A static function call is one that will be ‘unfolded’, that is, will be replaced with whatever it computes. The result of a static function call is not necessarily static, that is, available as a value early; the function call may generate a new program fragment such as `(+ 33 (* x 0))` rather than a value such as `33`.

To obtain a generating extension from the annotated program, we put a backquote on dynamic code that appears in static context, and put a comma on static code that appears in a dynamic context. Also, we may delete parameter `x` because it is dynamic and does not change at all in the recursive calls:

```
(define (polygen cs)
  (if (null? cs)
      '0
      '(+ ,(car cs) (* x ,(polygen (cdr cs))))
  ) )
```

Subjecting the `powr` example to the same treatment, we find that everything is static except the branches of the inner `if`:

```
(define (powr n x)
  (if (> n 0)
      (if (eq? (remainder n 2) 0)
          (sqr (powr (/ n 2) x))
          (* x (powr (- n 1) x))
      )
      1
  )
)
```

and hence, using backquote and comma:

```
(define (powrgen n)
  (if (> n 0)
      (if (eq? (remainder n 2) 0)
          `(sqr ,(powrgen (/ n 2)))
          `(* x ,(powrgen (- n 1)))
      )
      '1)
  )
```

14.3.7 Two-level languages and quasiquote

DynJava is an extension of the Java programming language with facilities for program generation [101], essentially a two-level language. In DynJava, code to be generated is prefixed with backquote (```) exactly as in Scheme. Inside backquoted code one can use the dollar sign (`$`) to insert the value of variables; this is a restricted version of Scheme's comma or unquote operator. For instance, a generating extension for evaluation of polynomials can be written as follows in DynJava:

```
`{ double res = 0.0; }
for (int i=cs.length-1; i>=0; i--)
  `{ res = res * x + $cs[i]; }
`{ return res; }
```

First the backquoted statement ``{ double res = 0.0; }` causes a declaration of `res` to be generated, then the `for` loop is executed and generates an assign-

ments to `res` for each element of `cs`, and finally a return statement is generated.

There are a number of academic tools for program generation with the same goals as `DynJava`, notably `SafeGen` [57], `Genoupe` [39] and `Jumbo` [?] for Java, and `Metaphor` [98] for C#. None of these can be considered quite ready for industrial use.

In the remainder of this chapter we shall generate specialized programs not in source form, but *in bytecode form*, using runtime code generation.

14.4 Runtime code generation using C#

The C# language and the .Net Common Language Infrastructure (CLI) support runtime code generation through the namespace `System.Reflection.Emit`. It seems to be a conscious design choice in CLI to make runtime code generation fairly easy and efficient.

In Section 14.4.1 we outline how to generate, at runtime, a new method `MyMethod`, and how to call the resulting method. Then Sections 14.4.2 through 14.4.6 show several examples that generate the body of such a method.

14.4.1 CLI runtime code generation of dynamic method

This subsection shows how to generate a so-called dynamic method at runtime. Our goal is to generate a method `MyMethod`, as if it were declared like this:

```
public static double MyMethod(double x) {
    ... generated method body ...
}
```

Despite the name, the generation of a ‘dynamic method’ actually adds a new static (non-instance) method to an existing CLI module. To call the generated method, we need a delegate type compatible with it. For this can either use a specific non-generic delegate type such as `D2D` that describes a delegate that takes as argument a `double` and returns a `double`:

```
public delegate double D2D(double x);
```

Or, since C# 3.5, we can use a type instance of the generic delegate type `Func<...>`; in this particular case:

```
Func<double,double>
```

The steps required to generate and call the dynamic method are outlined in Figure 14.1; the full example code is in file `rtcg/RTCG2D.cs`. First an object of type `DynamicMethod` from namespace `System.Reflection.Emit` is created. In this example, it describes a static method `MyMethod` that has return type `double` and a single parameter of type `double`, and belongs to the CLI module that contains the `String` type (the latter is not important).

```
// (1) Create a DynamicMethod and obtain an ILGenerator from it:
DynamicMethod methodBuilder =
    new DynamicMethod("MyMethod",           // Method name
                     typeof(double),       // Return type
                     new Type[] { typeof(double) }, // Argument types
                     typeof(String).Module); // Containing module
ILGenerator ilg = methodBuilder.GetILGenerator();
// (2) ... use ilg to generate the body of MyMethod (see later) ...
// (3) Call the newly generated method:
D2D mymethod = (D2D)methodBuilder.CreateDelegate(typeof(D2D));
double res = mymethod(5);
```

Figure 14.1: (1) Create a dynamic method, (2) generate its body, and (3) call the method through a delegate. Step (2) is detailed in Section 14.4.2.

14.4.2 Generating and calling a simple method

For illustration, let us generate the body of a simple method `MyMethod` in `MyClass`:

```
public virtual double MyMethod(double x) {
    System.Console.WriteLine("MyClass.MyMethod() was called");
    return x + 2.1;
}
```

We do this by implementing step (2) in Figure 14.1 as follows, using the CIL bytecode generator `ilg` created in those figures to generate bytecode:

```
ilg.EmitWriteLine("MyMethod() was called");
ilg.Emit(OpCodes.Ldarg_0);
ilg.Emit(OpCodes.Ldc_R8, 2.1);
ilg.Emit(OpCodes.Add);
ilg.Emit(OpCodes.Ret);
```

The first line generates bytecode to print a message. The second line loads the value of the newly generated method's first parameter; this is argument number 0 because the method is static. The third line pushes the double constant

2.1, the fourth line adds the former two values, and the `Ret` instruction returns the result.

The full example is from file `rtcg/RTCG2D.cs`, in which the newly generated method is called in two ways:

- The generated method may be called by reflection on the `DynamicMethod` object:

```
res = (double)methodBuilder.Invoke(null, new object[] { 6 });
```

- The generated method may be wrapped as a delegate and called by a delegate call, as already shown in Figure 14.1:

```
D2D mm = (D2D)methodBuilder.CreateDelegate(typeof(D2D));
res = mm(7);
```

This is faster than a reflective call, because it avoids runtime access checks, and wrapping of arguments and unwrapping of results.

14.4.3 Speed of code generated at runtime

The example `rtcg/RTCG3D.cs` generates three different versions of a method containing a simple loop, to be executed one billion times:

```
public static void MyMethod(int x) {
    do {
        x--;
    } while (x != 0);
}
```

The purpose of this example is to compare the speed of bytecode generated at runtime with the speed of ordinary compiled C# code, and to see how bytecode style affects the speed. The results, summarized in Section 14.7, show that bytecode generated at runtime is as fast as ordinary compiled code.

A bytecode loop can be generated by defining a label `start` to mark a position in the CIL code stream, and subsequently using the label in a conditional branch instruction:

```
Label start = ilg.DefineLabel();
ilg.MarkLabel(start);           // start:
ilg.Emit(OpCodes.Ldarg_0);      // push x
ilg.Emit(OpCodes.Ldc_I4_1);     // push 1
ilg.Emit(OpCodes.Sub);         // subtract
```

```

ilg.Emit(OpCodes.Starg_S, 0);      // store (x-1) in x
ilg.Emit(OpCodes.Ldarg_0);        // push x
ilg.Emit(OpCodes.Brtrue, start);  // if non-zero, go to start
ilg.Emit(OpCodes.Ret);           // return

```

Above, the loop counter is kept in the method's argument 0, but the loop counter might be kept on the stack top instead:

```

Label start = ilg.DefineLabel();
ilg.Emit(OpCodes.Ldarg_0);        // push x
ilg.MarkLabel(start);            // start:
ilg.Emit(OpCodes.Ldc_I4_1);       // push 1
ilg.Emit(OpCodes.Sub);           // subtract 1
ilg.Emit(OpCodes.Dup);           // duplicate stack top
ilg.Emit(OpCodes.Brtrue, start);  // if non-zero go to start
ilg.Emit(OpCodes.Pop);           // pop x
ilg.Emit(OpCodes.Ret);           // return

```

This code is functionally equivalent to the previous version, but apparently this use of the CLI stack prevents the just-in-time code generator from generating machine code that uses the x86 registers efficiently; so this is more than 50 % slower in both the Microsoft and Mono implementations.

Also, one might try to keep the loop counter in local variable 0 instead of argument 0. This makes no difference relative to the first version in the Microsoft implementation, but gives some speedup in the Mono 1.1.9 implementation provided option `optimize=all` is used.

14.4.4 Example: specializing the power function

For a potentially more useful case, consider again the `Power(n, x)` method from Section 14.2, which raises `x` to the `n`'th power. We define again a generating extension `PowerGen` for `Power`, but instead of generating Java or C# or Scheme source code, we shall generate bytecode, at runtime. This is implemented by file `rtcg/RTCG4D.cs`.

A call `PowerGen(n)` generates CIL code that will efficiently compute `x` to the `n`'th power, for any value of `x`. As in Sections 14.2 and 14.3.5, the bytecode generated by `PowerGen` contains no loops, no tests, and no computations on `n`; they have been performed during code generation. Even for moderate values of `n` (such as 16), the specialized code is therefore faster than the general code.

The `PowerGen` bytecode generator looks like this:

```

public static void PowerGen(ILGenerator ilg, int n) {
    ilg.DeclareLocal(typeof(int));    // p is local_0, x is arg_1
    ilg.Emit(OpCodes.Ldc_I4_1);

```

244 *Runtime code generation using C#*

```
    ilg.Emit(OpCodes.Stloc_0);           // p = 1;
while (n > 0) {
    if (n % 2 == 0) {
        ilg.Emit(OpCodes.Ldarg_1);     // x is arg_1
        ilg.Emit(OpCodes.Ldarg_1);
        ilg.Emit(OpCodes.Mul);
        ilg.Emit(OpCodes.Starg_S, 1);  // x = x * x
        n = n / 2;
    } else {
        ilg.Emit(OpCodes.Ldloc_0);
        ilg.Emit(OpCodes.Ldarg_1);
        ilg.Emit(OpCodes.Mul);
        ilg.Emit(OpCodes.Stloc_0);     // p = p * x;
        n = n - 1;
    }
}
ilg.Emit(OpCodes.Ldloc_0);
ilg.Emit(OpCodes.Ret);                // return p;
}
```

This generator is somewhat more verbose than `PowerTextGen` in Section 14.2 but has exactly the same structure. This shows there is nothing inherently mysterious about runtime bytecode generation. It is just a question of skipping the source code generation and the compiler, and going straight to the virtual machine's bytecode.

14.4.5 Example: compiled polynomial evaluation

Example `rtcg/RTCG5D.cs` shows the C# code to generate specialized evaluators for a polynomial with coefficients `cs`, using Horner's rule as in Section 14.3.5:

$$p(x) = cs[0] + x \cdot (cs[1] + x \cdot (\dots (cs[n] + 0) \dots))$$

Method `Poly(cs, x)` evaluates the polynomial at `x`:

```
public static double Poly(double[] cs, double x) {
    double res = 0.0;
    for (int i=cs.Length-1; i>=0; i--)
        res = res * x + cs[i];
    return res;
}
```

Method `PolyGen` generates, at runtime, a specialized version of `Poly` for a given coefficient array `cs`.

```

public static void PolyGen(ILGenerator ilg, double[] cs) { // x is arg_1
    ilg.Emit(OpCodes.Ldc_R8, 0.0);           // push res = 0.0 on stack
    for (int i=cs.Length-1; i>=0; i--) {
        ilg.Emit(OpCodes.Ldarg_1);           // load x
        ilg.Emit(OpCodes.Mul);               // compute res * x
        if (cs[i] != 0.0) {
            ilg.Emit(OpCodes.Ldc_R8, cs[i]); // load cs[i]
            ilg.Emit(OpCodes.Add);           // compute x * res + cs[i]
        }
    }
    ilg.Emit(OpCodes.Ret);                   // return res;
}

```

Again the generated code contains no loops, tests, or array indexing. The evaluation can be performed solely on the stack, accessing only the argument `x`, and pushing the coefficients as literal constants. The specialized code may be twice as fast as the general one even for rather polynomials of degree only 8, say, and may be much faster when there are many zero coefficients.

14.4.6 Example: compiled expression evaluation

Example `rtcg/RTCG6D.cs` presents a rudimentary abstract syntax for expressions in one variable, as may be found in a program to interactively draw graphs or compute zeroes of functions etc. The abstract syntax permits calls to static methods in class `System.Math`. An expression in abstract syntax (subclass of abstract class `Expr`) has two methods:

- method `double Eval(double x)` to compute the value of the expression as a function of `x`, essentially by interpreting the abstract syntax;
- method `void Gen(ILGenerator ilg)` to generate CIL code for the expression.

Even for small expressions the generated code is three times faster than the general interpretive evaluation by `Eval`.

14.5 JVM runtime code generation (gnu.bytecode)

Here we show the necessary setup for generating a Java class `MyClass` containing a method `MyMethod`, using the `gnu.bytecode` package [48]. First we use reflection to create a named public class `MyClass` with superclass `Object`, and make the class implement an interface that describes the method `MyMethod` we want to generate. The full details of this example are in file `rtcg/RTCG2.java`.

```
// (1) Create class and method:
ClassType co = new ClassType("MyClass");
co.setSuper("java.lang.Object");
co.setModifiers(Access.PUBLIC);
co.setInterfaces(new ClassType[] { new ClassType("IMyInterface") });
... (1*) generate a constructor in class MyClass ...
Method mo = co.addMethod("MyMethod");
mo.setSignature("(D)D");
mo.setModifiers(Access.PUBLIC);
mo.initCode();
CodeAttr jvmg = mo.getCode();
... (2) use jvmg to generate the body of method MyClass.MyMethod ...
// (3) Load class, create instance, and call generated method:
byte[] classFile = co.writeToArray();
Class ty = new ArrayClassLoader().loadClass("MyClass", classFile);
Object obj = ty.newInstance();
IMyInterface myObj = (IMyInterface)obj;
double res = myObj.MyMethod(5);
```

Figure 14.2: Setup for RTCG with Java and `gnu.bytecode`: (1) Create class and method; (2) generate method body; and (3) create class and call method. Step (1*) is detailed in Figure 14.3.

An argumentless constructor is added to class `MyClass` in step (1*) of Figure 14.2 by adding a method with the special name `<init>`. The constructor simply calls the superclass constructor, using an `invokespecial` instruction, as shown in Figure 14.3.

Then a method with given signature and access modifiers, represented by method object `mo`, is added to the class in step (2) of Figure 14.2, and a code generator `jvmg` is obtained for the method and is used to generate the method's body.

Once the constructor and the method have been generated, in step (3) of Figure 14.2 a representation of the class is written to a byte array and loaded into the JVM using a class loader. This produces a class reference `ty` that represents the new class `MyClass`.

Finally, to call the newly generated method, an instance `obj` of the class is created and cast to the interface describing the generated method, and then the method in the object `myObj` is called using an interface call `myObj.MyMethod(5)`.

14.6 JVM runtime code generation (BCEL)

The Bytecode Engineering Library BCEL [21] is another third-party Java library that can be used for runtime code generation. Here we outline the necessary setup for code generation with BCEL.

One must create a class generator `cg`, specifying superclass, access modifiers and a constant pool `cp` (Section 9.3) for the generated class, and an interface describing the generated method. Then one must generate a constructor for the class, and generate the method, as outlined in Figure 14.4.

An argumentless constructor is added to class `MyClass` (step 1 in Figure 14.4) by adding a method with the special name `<init>`. The constructor should just call the superclass constructor, as shown in Figure 14.5.

Step 2 in Figure 14.4 generates the body of the method. Then step 3 writes a representation `clazz` of the class to a byte array, loads it into the JVM using a class loader, creates an instance of the class and casts it to the interface. Then the generated method can be called by an interface call as in Section 14.5. Example files `rtcg/RTCG4B.java` and `rtcg/RTCG7B.java` contain other complete examples using the BCEL bytecode toolkit.

14.7 Speed of code and of code generation

Figures 14.6 and 14.7 compare the speed of the generated code, and the speed of code generation, for three JVM implementations, Microsoft's CLR, and the Mono implementation. Experiments were made (around 2004) under Linux,

```

Method initMethod =
    co.addMethod("<init>", new Type[] {}, Type.void_type, 0);
initMethod.setModifiers(Access.PUBLIC);
initMethod.initCode();
CodeAttr jvmg = initMethod.getCode();
Scope scope = initMethod.pushScope();
Variable thisVar = scope.addVariable(jvmg, co, "this");
jvmg.emitLoad(thisVar);
jvmg.emitInvokeSpecial(ClassType.make("java.lang.Object")
    .getMethod("<init>", new Type[] {}));
initMethod.popScope();
jvmg.emitReturn();

```

Figure 14.3: Generating a constructor at (1*) in Figure 14.2.

```

// (1) Create class and method:
ClassGen cg = new ClassGen("MyClass", "java.lang.Object",
    "<generated>",
    Constants.ACC_PUBLIC | Constants.ACC_SUPER,
    new String[] { "IMyInterface" });
ConstantPoolGen cp = cg.getConstantPool();
InstructionFactory factory = new InstructionFactory(cg);
... (1*) generate a constructor in class MyClass ...
InstructionList il = new InstructionList();
MethodGen mg = new MethodGen(Constants.ACC_PUBLIC,
    Type.DOUBLE, new Type[] { Type.DOUBLE },
    new String[] { "x" },
    "MyMethod", "MyClass", il, cp);
... (2) use il to generate the body of method MyClass.MyMethod ...
mg.setMaxStack();
cg.addMethod(mg.getMethod());
// (3) Load class, create instance, and call generated method:
JavaClass clazz = cg.getJavaClass();
byte[] classFile = clazz.getBytes();
Class ty = new ArrayClassLoader().loadClass("MyClass", classFile);
Object obj = ty.newInstance();
IMyInterface myObj = (IMyInterface)obj;
double res = myObj.MyMethod(5);

```

Figure 14.4: Setup for RTCG in Java with BCEL: (1) Create class and method; (2) generate method body; and (3) create class and call method. Step (1*) is detailed in Figure 14.4.

with Sun Hotspot 1.5.0, IBM J2RE 1.4.2 [37], Mono 1.1.9 with option `optimize=all` [95], gcc 3.3.5 and GNU Lightning 1.2 [84] under Linux, and Microsoft CLR 2.0 beta under Windows 2000 under VmWare 4.5.2.

Sun's Hotspot Client VM performs approximately 400 million iterations per second, and the IBM JVM performs nearly twice as many. In both cases this is as fast as bytecode compiled from Java source, and in case of the IBM JVM, even as fast as machine code compiled from C (optimized with `-O2`). This shows that bytecode generated at runtime carries no inherent speed penalty compared to code compiled from Java programs. The Sun Hotspot Server VM optimizes more aggressively and removes the entire loop because its results are never used. These measurements were made with example `rtcg/RTCG3D.cs` from Section 14.4.3 and example `rtcg/RTCG3.java`.

On the same platform, straight-line JVM bytecode can be generated at a rate of about a million bytecode instructions per second with Sun HotSpot Client VM and approximately half that with the Sun Hotspot Server JVM or the IBM JVM, when using the `gnu.bytecode` package [48]. Code generation with BCEL package [21] seems to be only half as fast as with `gnu.bytecode`. The code generation time includes the time taken by the just-in-time compiler to generate machine code. These measurements were made with `rtcg/RTCG7D.cs`, `rtcg/RTCG7.java` and `rtcg/RTCG7B.java`.

14.8 Efficient reflective method calls in Java

Reflective method calls in Java are slow because of the need to wrap primitive type arguments (`int` and so on) as objects and to unwrap primitive type results. In contrast to C#, a reflective method handle `mo`, which is an object of class `java.lang.reflect.Method`, cannot be wrapped as a delegate.

However, using runtime code generation one can obtain a similar effect, by creating from `mo` an object of a class that has a method of the correct compile-time type.

More precisely, assume we have a `Method` object `mo` that represents a method with signature $(R)(T_1, \dots, T_n)$. Then we want to dynamically create an object `oi` whose class implements a compiled interface `OI` which describes a method $R\ m(T_1, \dots, T_n)$ corresponding to `mo`. Moreover, the creation of `oi` must check, once and for all, that the type $R\ m(T_1, \dots, T_n)$ for `m` given by `OI` actually matches the type described by the `Method` object `mo`, and that all access restrictions — no access to private methods, and so on — are respected, so that this need not be checked at every invocation of `oi.m(...)`. Also, any primitive type arguments passed to `oi.m(...)` must be passed straight on to the method underlying `mo`, without any boxing or unboxing operations.

```

InstructionFactory factory = new InstructionFactory(cg);
InstructionList ilc = new InstructionList();
MethodGen mgc = new MethodGen(Constants.ACC_PUBLIC,
                             Type.VOID, new Type[] { }, new String[] { },
                             "<init>", "MyClass",
                             ilc, cp);
ilc.append(factory.createLoad(Type.OBJECT, 0));
ilc.append(factory.createInvoke("java.lang.Object", "<init>",
                               Type.VOID, new Type[] { },
                               Constants.INVOKESPECIAL));

ilc.append(new RETURN());
mgc.setMaxStack();
cg.addMethod(mgc.getMethod());

```

Figure 14.5: Generating a constructor at (1*) in Figure 14.4.

	Sun HotSpot		IBM JVM	MS CLR	Mono CLI	C gcc gnulig'n
	Client	Server				
Compiled loop (10 ⁶ /s)	392	∞	781	775	775	781
Generated loop (10 ⁶ /s)	392	∞	781	775	775	515
Code generation (10 ³ /s)	1000	450	500	700	350	>50000

Figure 14.6: Speed of simple loop, and of code generation; 1.6 GHz Pentium M.

	Sun HotSpot		IBM JVM	MS CLR	Mono CLI	C gcc gnulig'n
	Client	Server				
Compiled loop (10 ⁶ /s)	875	∞	1770	2220	1727	1818
Generated loop (10 ⁶ /s)	875	∞	2150	2220	1755	1818
Code generation (10 ³ /s)	1030	440	550	833	475	>50000

Figure 14.7: Speed of simple loop, and of code generation; 2.8 GHz Pentium 4.

In this way we obtain an object that is rather similar to a delegate, in Java. Not surprisingly, this implementation of delegate creation in Java is approximately 100 times slower than the built-in delegate creation in Microsoft's CLR. Experiments indicate that a Java 'delegate' created in this way must be called approximately 2,000 times before the cost of creating the delegate class and the delegate object has been recovered.

14.9 Applications of runtime code generation

General application areas of runtime code generation include:

- unrolling of recursion and loops, and inlining of constants, as in the `Power` example, the polynomials examples, or vector dot product computation;
- removal of interpretive overhead, as in the MS .Net Framework regular expression matcher;

Concrete example applications of runtime code generation:

- generation of efficient code for expressions or functions entered by a user at runtime, for instance in a function graph drawing program (`rtcg/RTCG6D.cs`);
- fast raytracing algorithms for a given scene;
- specialized sorting routines with inlining of the comparison predicates, or compiling different comparison predicates dependent on the type or order of elements being sorted; see the record comparison example in [74];
- training neural networks of a given network topology;
- the `bitblt` example from [74];
- fast customized serialization and deserialization methods for a given class [132];
- fast multiplication of sparse matrices (`rtcg/RTCG8D.cs` and `rtcg/RTCG8.java`).

14.10 History and literature

Program generators have been used for decades, mostly in the form of generators of programs in source text which are then compiled by an ordinary compiler. Recent examples include Velocity [135] and XDoclet [138], which

are widely used, for instance to generate skeleton Java classes from database schemas or data descriptions in XML.

Quasiquote was used by the linguist V.v.O. Quine around 1940, and introduced in the Lisp programming language before 1980. Alan Bawden [13] explains the use and history of quasiquote (backquote and comma) in Lisp and Scheme.

David Keppel, Susan J. Eggers and Robert R. Henry argue in a series of technical reports that runtime code generation is useful [74, 75]. The reports require some understanding of processor architecture. Later work in the same group led to the DyC system for runtime code generation in C [9, 52].

Bytecode generation tools for Java include `gnu.bytecode` [48], developed for Kawa, a JVM-based implementation of Scheme, and the Bytecode Engineering Library (BCEL) [21], formerly called `JavaClass`, which seems to be used in several projects.

Tools for runtime code generation in C, called 'C or tick-C, have been developed by Dawson R. Engler and others [43, 44]. Their portable Vcode system is fast; it generates code using approximately 10 instructions per generated instruction.

The staging of computations implied by runtime code generation is closely related to *staging transformations* [68]. Automatic staging and specialization of programs in the subject of *partial evaluation* and *program specialization*.

Partial evaluation was proposed by Futamura in 1970. It received much attention in Japan, Russia and Sweden in the 1970'es, and worldwide attention since the mid-1980'es. The monograph by Jones, Gomard and Sestoft [63] and the encyclopedia paper [93] provide an overview of this area, many examples, and references to the literature.

Indeed, partial evaluation provided the original inspiration for Nielson and Nielson's [99] detailed formal study of two-level functional languages. Danvy and Filinski used the concept of a two-level language in their study and improvement of the CPS transformation [33].

Antonio Cisternino and Andrew Kennedy have developed a library for C# that considerably simplifies runtime generation of CIL code [27, 7, 26].

Implementations of two-level languages, or metaprogramming languages, include Sheard and Taha's MetaML [130], Oiwa's DynJava [101], Calcagno and Taha's MetaOCaml [22], based on OCaml, and finally Draheim's Genoupe [39] and Neverov's Metaphor [98], both based on C#. Older related systems include Engler's tick-C, mentioned above, and Leone's Fabius [80], but these are defunct by now.

A challenge that has been addressed especially in recent years is how ensure that generated programs are always well-formed and well-typed, ideally by (type)checking only the generating program. For Scheme, which is dynami-

cally typed, this problem is usually ignored, but for statically typed languages such as ML, Java and C# this ought to be possible. This has been achieved in e.g. the metaprogramming language MetaOCaml, but it is rather more difficult for general program generators such as SafeGen [57].

A comparison of the efficiency and potential speedups of runtime code generation in several implementations of the Java/JVM and C#/CLR runtime environments can be found in [118]. That report also provides more examples, such as runtime code generation in an implementation of the US Federal Advanced Encryption Standard (AES).