

Programming Languages, F2005

Lecture 11, Wednesday 27 April 2005

- Reflection and runtime code generation in modern execution platforms
- Reflection in C# and Java
- Program generation and partial evaluation
- Runtime code-generation in C# and Java

Reflection in C#: manipulating classes, methods, etc at runtime (rtcg/Reflect0.cs)

Reflection means that a running program can manipulate information about its classes, methods, etc.

For every compiletime *type* T there is a runtime *value* `typeof(T)` of class `Type` representing it:

```
using System;
```

```
class Reflect0 {  
    public static void Main(String [] args) {  
        Type ty1 = typeof(int);  
        Console.WriteLine(ty1);  
        Console.WriteLine(typeof(long));  
        Console.WriteLine(typeof(Reflect0));  
        Console.WriteLine(typeof(String[][]));  
    }  
}
```

More powerful and protective runtime environments (JVM and MS CLR)

- More execution stages:
 - compilation to bytecode (`javac`, `csc`)
 - dynamic loading and linking of new code (classloader)
 - compilation of bytecode to machine code (JIT)
 - execution of machine code
- More information (metadata, types) in the bytecode to support dynamic loading and verification.
- Metadata permit *reflection*: a running program can inspect its own classes, methods, etc.
- JIT compilation of bytecode to optimized machine code supports portable *runtime code generation*.
Runtime code generation: a running program can create new classes, methods, etc.

Neither reflection nor runtime code generation is well supported in Pascal, C, C++, Fortran, and so on.

Reflection in C#: calling a method (rtcg/Reflect1.cs)

One can get a particular method (or other member) from a type, as a `MethodInfo` object.

One can call the method represented by the object:

```
using System; // For Type  
using System.Reflection; // For MethodInfo  
  
class Reflect1 {  
    public static void Main(String [] args) {  
        Type ty = typeof(Reflect1); // Get Reflect1 class  
        MethodInfo m = ty.GetMethod("Foo"); // Get Foo method  
        m.Invoke(null, new object[] { }); // Call it  
    }  
  
    public static void Foo() { Console.WriteLine("Foo was called"); }  
}
```

Compiletime safety is poor: the compiler cannot check that the method exists, is static, is accessible, etc.

A reflective method call is 30–40 times slower than a compiled call to a method.

Instead, one can create a delegate from the `MethodInfo` — example in file `rtcg/RTCG2D.cs`.

A delegate call is only 1.5–2.0 times slower than a compiled method call.

Some practical uses of reflection (rtcg/Reflect2.cs)

Example: Find and call all public static methods whose name starts with Test (similar to jUnit).

```
class Reflect2 {
    public static void Main(String [] args) {
        Type co = typeof(Reflect2);           // Get Reflect2 class
        MethodInfo[] mos = co.GetMethods();   // Get all public methods
        Console.WriteLine("These public static methods are available:");
        for (int i=0; i<mos.Length; i++)
            if (mos[i].IsStatic)
                Console.WriteLine(mos[i].Name);
        Console.WriteLine();
        Console.WriteLine("Calling public static methods whose names start with Test:");
        for (int i=0; i<mos.Length; i++)
            if (mos[i].IsStatic && mos[i].Name.IndexOf("Test") == 0)
                mos[i].Invoke(null, new Object[] {});
    }

    public static void Foo() { Console.WriteLine("Foo"); }
    public void NonStaticFoo() { Console.WriteLine("NonStaticFoo"); }
    static void NonPublicFoo() { Console.WriteLine("NonPublicFoo"); }
    public static void TestGoo() { Console.WriteLine("TestGoo"); }
    public static void TestFoo() { Console.WriteLine("TestFoo"); }
}
```

Generating specialized programs

Sometimes we have a general program and it would pay to generate specialized versions of it.

Example: The power function

Computing x^n , that is, x 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 function is based on these mathematical equivalences:

$$x^{2m} = (x^2)^m$$
$$x^{2m+1} = x^{2m} \cdot x$$

Reflection in Java: calling a method (rtcg/Reflect1.java)

Class Class in package java.lang corresponds to CLR's class Type.

Class Method in package java.lang.reflect corresponds to CLR's class MethodInfo, etc.

In general, the reflection machinery in CLR and Java nearly identical.

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

class Reflect1 {
    public static void main(String[] args)
        throws NoSuchMethodException, IllegalAccessException,
            InvocationTargetException {
        Class 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");
    }
}
```

Specializing the power function

Assume that we need to compute x^7 for many different values of x .

Then we can use a faster *specialized* method Power_7 that works only for $n = 7$ but for all x :

```
public int Power_7(int x) {
    int p;
    p = 1;
    p = p * x;
    x = x * x;
    p = p * x;
    x = x * x;
    p = p * x;
    return p;
}
```

Staging and partial evaluation

We have split the computation of $\text{Power}(x, 7)$ into two stages:

- First do all computations that depend on $n = 7$ only; the result is a new method Power_7.
- Then execute method Power_7 for as many different x as needed.

A lot more can be (and has been) said about this. See for instance

Jones, Gomard, Sestoft: *Partial evaluation and automatic program generation*, Prentice-Hall International 1993.

Generating a specialized power method (rtcg/Power.java)

```
public static void Power(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("}");
}
```

The drawback is that we need to invoke the Java (or C#) compiler to compile the generated function.

It may be better to generate bytecode, at runtime.

But, how fast is code generated at runtime?

No slower! Code generated at runtime for this loop is just as fast as code compiled from Java/C#:

```
do {
    n--;
} while (n != 0);
```

Speed of the generated code, and code generation speed:

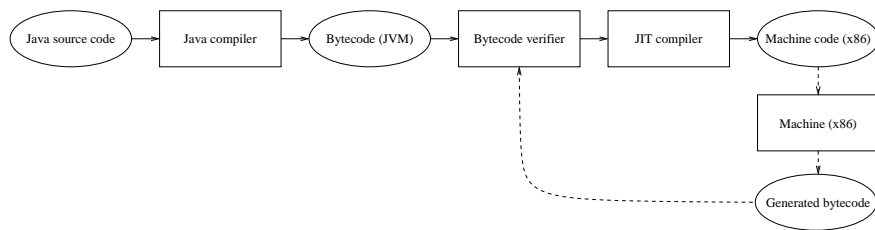
	Sun HotSpot JVM		IBM	MS	C
	Client	Server	JVM	CLR	gcc -O2
Compiled loop (million iter/sec)	243	∞	421	408	422
Generated loop (million iter/sec)	243	∞	421	408	N/A
Code generation (thousand instr/sec)	200	142	180	100	N/A

Sun HotSpot 1.4.0 and IBM JIT 1.3.1 for Linux, and MS .Net CLR 1.0 on Windows 2000; 850 MHz Pentium 3.

Stack operations upset MS CLR: using Load ;Dup instead of Load ;Load can make code 37 % slower!

Although the virtual machines are stack machines, the JIT generates traditional CPU register-based code.

Runtime code generation (RTCG) in Java Virtual Machine and Common Language Runtime



At runtime new bytecode can be generated and loaded.

The JVM or CLR just-in-time (JIT) compiler will compile it to efficient machine code.

The JIT compiler must be present in the JVM or CLR anyway.

Hence the overhead of this is much smaller than the overhead of calling the Java or C# compiler.

RTCG in C#: Generating bytecode for the function $\text{power}_n(x)$ for a fixed n (rtcg/RTCG4D.cs)

```
public static void PowerGen(ILGenerator ilg, int n) {
    ilg.DeclareLocal(typeof(int)); // p is local_0, x is arg_0
    ilg.Emit(OpCodes.Ldc_I4_1);
    ilg.Emit(OpCodes.Stloc_0); // p = 1;
    while (n > 0) {
        if (n % 2 == 0) {
            ilg.Emit(OpCodes.Ldarg_0); // x is arg_0
            ilg.Emit(OpCodes.Ldarg_0);
            ilg.Emit(OpCodes.Mul);
            ilg.Emit(OpCodes.Starg_S, 0); // x = x * x
            n = n / 2;
        } else {
            ilg.Emit(OpCodes.Ldloc_0);
            ilg.Emit(OpCodes.Ldarg_0);
            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;
}
```

For $n = 16$, the specialized code is 55 percent faster than the general code.

(Later we see where the `ILGenerator ilg` comes from).

Runtime code generation example: Evaluation of polynomials (rtcg/RTCG5D.cs)

The polynomial $p(x)$ of degree n with coefficient array $cs[0 \dots n]$:

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

According to Horner's rule, this is equivalent to:

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

Given coefficient array $cs[]$ and x , we can compute $p(x)$ with result in variable `res`:

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

Potential for staging, or splitting of the binding-times:

If a given polynomial $p(x)$ must be evaluated for many different values of x , then do it in two stages:

- (1) Generate specialized code for the given coefficient array $cs[0 \dots n]$;
- (2) for every x , execute the specialized code.

How to generate the specialized code in C#/CLR

```
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_0); // load x
    ilg.Emit(OpCodes.Mul); // compute res * x
    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;
```

Further optimization: skip term if coefficient $cs[i]$ is zero

```
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_0); // 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;
```

The generated code is faster only if the array $cs[]$ has more than 10–20 elements or many coefficients are zero.

The specialized code for a given polynomial

The constant cs_i is the value of $cs[i]$:

```
double res = 0.0;
res = res * x + cs_n;
...
res = res * x + cs_1;
res = res * x + cs_0;
return res;
```

The corresponding stack-oriented bytecode (for CLR)

```
Ldc_R8 0.0 // push res = 0.0 on stack
Ldarg_0 // load x
Mul // compute res * x
Ldc_R8 cs_n // load cs[n]
Add // compute res * x + cs[n]
...
Ldarg_0 // load x
Mul // compute res * x
Ldc_R8 cs_0 // load cs[0]
Add // compute res * x + cs[0]
Return // return res
```

The full story: runtime code generation in the CLR (rtcg/RTCG4D.cs)

In CLR 1.1, runtime code generation was rather complicated: to generate byte code, one needs a method, which needs a class, which needs a module, which needs an assembly, which needs an assembly name; in total at least 30 lines of C# setup code.

In CLR 2.0, a `DynamicMethod` is a static method that can be inserted into an existing module or class.

Runtime code generation uses the `System.Reflection.Emit` namespace of the .NET Framework.

Here's how to build a method `public static int MyPower(int x) { ... }`:

```
DynamicMethod methodBuilder =
    new DynamicMethod("MyPower",
        typeof(int),
        new Type[] { typeof(int) },
        typeof(String).Module);
ILGenerator ilg = methodBuilder.GetILGenerator();
```

The `ilg` object can then be used in `PowerGen(ilg, n)` to generate the body of the method.

Calling the created method

From a `DynamicMethod` object one can create a delegate (a function value) which can then be called:

```
Int2Int power_n_dlg =
    (Int2Int)(methodBuilder.CreateDelegate(typeof(Int2Int)));
int res = power_n_dlg(3);
```

where the delegate type `Int2Int` represents the type `int -> int`:

```
public delegate int Int2Int(int x);
```

Delegate calls are sometimes slower than interface methods calls.

So the 'old' complicated technique for runtime code generation can give faster calls than `DynamicMethod`:

- Let the generated method `M` be an instance method in a class `C` implementing an existing interface `I`.
- Create an instance of `C` and bind it: `I c = (C)(...create instance of C...)`.
- Call `c.M(...)` on the object `c` via the interface as many times as necessary.

This is illustrated in `rtcg/RTCG4.cs` and `rtcg/RTCG4.java`.

It remains the only reasonable approach in Java runtime code generation (no delegates, and slow reflective calls).

One more example: Sparse matrix multiplication (`rtcg/RTCG8.java`)

Plain multiplication $R = A \cdot B$ of two $n \times n$ matrices uses n^3 scalar multiplications:

```
for (int i=0; i<rRows; i++)
    for (int j=0; j<rCols; j++) {
        double sum = 0.0;
        for (int k=0; k<aCols; k++)
            sum += A[i][k] * B[k][j];
        R[i][j] = sum;
    }
```

If B has few non-zero elements, we can find the non-zeroes of each row and then multiply with A 's rows.

A `SparseMatrix` is an array of lists of the non-zero elements of each column.

```
SparseMatrix sparseB = new SparseMatrix(B);
for (int i=0; i<rRows; i++) {
    final double[] Ai = A[i], Ri = R[i];
    for (int j=0; j<rCols; j++) {
        double sum = 0.0;
        Iterator iter = sparseB.getCol(j).iterator();
        while (iter.hasNext()) {
            final NonZero nz = (NonZero)iter.next();
            sum += Ai[nz.k] * nz.Bkj;
        }
        Ri[j] = sum;
    }
}
```

What if we need to compute $A \cdot B$ for fixed B and many different A ?

Runtime code generation in Java

The JDK has no standard classes to support runtime code generation.

Two sets of tools for runtime code generation in Java are:

- Package `gnu.bytecode` from <http://www.gnu.org/software/kawa/>
Bytecode generation tools developed for Kawa, a JVM-based implementation of Scheme.
- Bytecode Engineering Library (BCEL, formerly JavaClass) from <http://jakarta.apache.org/bcel/>

Generating JVM code for the second stage

```
Label loop = new Label(jvmg);
loop.define(jvmg);
jvmg.emitLoad(varA); jvmg.emitLoad(vari); // do {
jvmg.emitArrayLoad(double1D_type);
jvmg.emitStore(varAi); // Ai = A[i]
jvmg.emitLoad(varR); jvmg.emitLoad(vari);
jvmg.emitArrayLoad(double1D_type);
jvmg.emitStore(varRi); // Ri = R[i]
for (int j=0; j<B.cols; j++) {
    jvmg.emitLoad(varRi); // Load Ri
    jvmg.emitPushInt(j);
    jvmg.emitPushDouble(0.0); // sum = 0.0
    Iterator iter = B.getCol(j).iterator();
    while (iter.hasNext()) {
        final NonZero nz = (NonZero)iter.next();
        jvmg.emitPushDouble(nz.Bkj); // load B[k][j]
        jvmg.emitLoad(varAi); // load A[i]
        jvmg.emitPushInt(nz.k);
        jvmg.emitArrayLoad(Type.double_type); // load A[i][k]
        jvmg.emitMul(); // prod = A[i][k]*B[k][j]
        jvmg.emitAdd('D'); // sum += prod
    }
    jvmg.emitArrayStore(Type.double_type); // R[i][j] = sum
}
jvmg.emitLoad(vari); jvmg.emitPushInt(1);
jvmg.emitAdd('I'); jvmg.emitStore(vari); // i++
jvmg.emitLoad(vari); jvmg.emitPushInt(aRows);
jvmg.emitGotoIfLt(loop); // } while (i<aRows);
jvmg.emitReturn();
```

Performance of sparse matrix multiplication (100×100 matrices, 5% non-zeroes)

	100 matrix multiplications				1000 matrix multiplications			
	Sun HotSpot		IBM	MS	Sun HotSpot		IBM	MS
	Client	Server	JVM	CLR	Client	Server	JVM	CLR
Plain	2.749	2.302	1.067	1.432	27.489	21.890	10.535	14.230
Sparse, recompute sparseB	1.118	0.820	0.904	1.191	10.660	5.548	7.405	11.567
Sparse, reuse sparseB	0.993	0.458	0.684	0.931	9.814	4.438	6.737	9.263
Sparse, rtcg	0.222	0.467	1.341	0.300	0.920	6.995	1.482	0.691

Sun HotSpot 1.4.0 and IBM JIT 1.3.1 for Linux, and MS .Net CLR 1.0 on Windows 2000; 850 MHz Pentium 3.

It takes 3.1 ms to generate the second stage code in Sun HotSpot Client VM with `gnu.bytecode`.

Approximately 37.5 KB bytecode is generated; the total space overhead is 130 KB.

Only one matrix multiplication is needed for runtime code generation to pay for itself.

Because ... the generated code is used 100 times, once for each row of A .

Conclusions and observations

- The JVM and CLR are interesting, widely available platforms for runtime code generation.
- Bytecode generation plus just-in-time compilation provides for portability and fast generated code.
- Code generation (incl. JIT) is fast: 200,000 instructions/sec for Sun HotSpot Client VM; 100,000 for MS CLR.
- Sun HotSpot Client VM and Microsoft's CLR support runtime code generation well.
- IBM's JVM and Sun HotSpot Server VM are somewhat less suitable.
- For Java, both the `gnu.bytecode` and BCEL libraries are robust and usable.
- JIT-based implementations use much memory (at least 10 MB), so RTCG is not for embedded systems.
- The performance of JIT-based implementations is somewhat unpredictable, due to adaptive optimizations.

Two-level source languages instead of bytecode

- Lisp/Scheme backquote (```) and comma (`,`) and `eval` (MIT Lisp 1978). Polynomial example:

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

For instance, `(polygen '(11 22 33))` gives `(+ 11 (* x (+ 22 (* x (+ 33 (* x 0)))))`

- Typed two-level Java: DynJava (Oiwa et al. 2001); surprisingly poor speed-up. Polynomial example:

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

- Types can help avoid binding-time mistakes, and make sure that generated code is well-formed.
- Untyped two-level C: Tick C (Engler et al. 1996); no longer maintained.
- Runtime program specialization in C: The Tempo system (Consel and Noël 1996)
- Untyped two-level OCaml (bytecode, untyped): Dynamic Caml (Lomov and Moskal 2001).
- Multi-level typed ML (machine code): MetaML (Sheard 1998).
- Multi-level typed OCaml (bytecode): MetaOCaml (Calcagno, Taha, Huang, Leroy 2001).
- Metaphor, Multi-level C#. Neverov and Roe, 2004. <<http://sky.fit.qut.edu.au/~neverov/metaphor/>>