Runtime code generation in JVM and CLR:

Experience and potential

Peter Sestoft

IT University of Copenhagen, Denmark

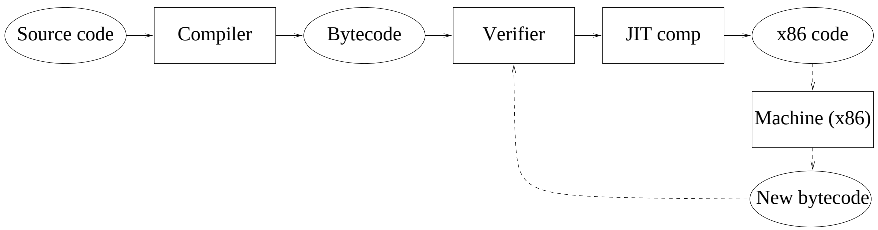
and

Royal Veterinary and Agricultural University, KVL

Mainstream execution platforms: The Java Virtual Machine (JVM) and Common Language Runtime (CLR)

At compiletime, Java and C# are compiled to bytecode for a stack machine.

At runtime, the bytecode is compiled to real machine code by a just-in-time compiler (JIT).



Runtime code generation (RTCG) in JVM and CLR

The runtime systems support reflection — more similar to Scheme/Smalltalk than to C/C++.

Moreover, at runtime new bytecode can be generated and loaded, and the JIT will compile it to machine code.

Same purpose as compile-time specialization: Generate specialized fast code once, use it many times.

Runtime code generation: Some research topics

- Type safety and nice type systems for multi-stage languages [not this talk].
- Language notations for convenient code generation [not this talk].

This talk: Runtime code generation has lots of practical potential

- Traditionally there has been a conflict between:
  - Portability: Generate bytecode; slow, and closed world, separate from mainstream software.
  - Speed: Generate machine code; non-portable, and too much engineering effort to maintain.
- Claim: JVM and CLR provide portability **and** speed.
  Because of bytecode plus JIT plus lots of engineering effort, made by others.
- We show that bytecode can be very fast, and not to great a pain to generate.
- Case study: the Advanced Encryption Standard (AES, Rijndael) in C#.
- Case study: Fast sparse matrix multiplication in Java.
- The need for better tools: type systems and notations.

The Java Virtual Machine (JVM)

The JVM is a specification of an abstract machine. Some implementations:

- Sun HotSpot Client VM, ‘fast JIT, slow code’; `java -client`
- Sun HotSpot Server VM, ‘slow JIT, fast code’; `java -server`
- IBM JIT JVM, ‘slow JIT, fast code’

Runtime code generation kit: `gnu.bytecode`; others include Apache BCEL and ASM.

The Common Language Runtime (CLR) for C#, VB.Net, . . .

The CLR is a specification of an abstract machine. Some implementations:

- Microsoft’s CLR 2.0, part of Visual Studio 2005 and Windows Vista (2006).
- The Mono project’s CLR 1.1.9

Runtime code generation kit: namespace System.Reflection.Emit

Both platforms are widely used: desktop and server installations are counted in tens or hundreds of millions.

Initial question: Is bytecode generated at runtime fast enough to bother?

Initial answer: Yes, surprisingly fast

Consider a trivial loop:

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

A generated version of the loop expressed in CLR bytecode:

```
start: Ldarg_0
       Ldc_I4_1
       Sub
       Starg_0
       Ldarg_0
       Brtrue start
```

The machine code JIT'ted from the bytecode (by Mono for x86) is:

```
12:   dec    %edi
13:   test   %edi,%edi
15:   jne    12
```


Generated bytecode is as fast as compiled Java and C# and even C (1.6 GHz Pentium M)

	Sun HotSpot		IBM	MS	Mono	C gcc -O3
	Client	Server	JVM	CLR	CLR	gnulig'n
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

Iterations per second: bigger is better.

Generated bytecode is faster than C (2.8 GHz Pentium 4): Long pipeline, alignment artefacts?

	Sun HotSpot		IBM	MS	Mono	C gcc -O3
	Client	Server	JVM	CLR	CLR	gnulig'n
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

Iterations per second: bigger is better.

Sun Hotspot 1.5.0, IBM JIT 1.4.2, Mono 1.1.9 optimize=all for Linux; and MS CLR 2.0 beta 2.

What does runtime bytecode generation look like? Example: Evaluation of polynomials

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

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

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

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

Given coefficent 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 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 coeffient array $cs[\,]$; then (2) for every x, execute the specialized code.

The specialized code for a given polynomial

Let constant `cs_i` be 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
```

How to generate the specialized code in C#/CLR

The standard evaluation loop:

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

To generate bytecode, one uses a bytecode generator `ilg` of type `ILGenerator`:

```
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 res * x + cs[i]
}
ilg.Emit(OpCodes.Ret);                      // return res;
```


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

In the standard evaluation loop the optimization makes no sense; an addition is likely faster than a test:

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

But in the polynomial evaluator generator, the test can be performed once, at bytecode generation time:

```
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 coefficient array $cs[]$ is long ($\geq$ 10) or many coefficients are zero.

For more complicated expressions the speed-up is larger. Spreadsheets, graphing applications, . . .

A surprising concern: How do we get to execute the generated code?

Bytecode must belong to a method: to create and execute bytecode we must create and call a method.

- Approach 1: A method must belong to a class; a class to a module; and a module to an assembly.

  So create an AssemblyBuilder, a ModuleBuilder, a TypeBuilder, a MethodBuilder, and an ILGenerator.

  Use to generate bytecode in a method in a class implementing a given interface. Create instance of class and cast to the interface, then call generated method by an interface call.

  ```
  public interface IMyInterface {
    double MyMethod(double x);
  }
  ```

- Approach 2 (since CLR 2.0): Use class DynamicMethod to add a new static method to an existing module.

  But a DynamicMethod must be called as a delegate, and that is surprisingly slow:

Call to generated	Time (ns/call)
Interface call	10
Delegate call	46
Reflective call	2834

- Approach 3: Same as approach 1, but use generic types and reflection to hide the gory details.

Intergalatic phrasebook: Function types and function values in C# and Java

C# delegate type and anonymous method	**Standard ML**
`public delegate R D(A x);`	`type D = A -> R`
`delegate(A x) { return e; }`	`fn (x : A) => e`
C# and Java interface types	**Standard ML**

`public interface IFun {`
`  R Invoke(A x);`
`}`
	`type IFun = { Invoke : A -> R }`

`public interface IFun<A,R> {`
`  R Invoke(A x);`
`}`
	`type ('A, 'R) IFun`
	`    = { Invoke : 'A -> 'R }`

Approach 1: Generating a virtual method in a class implementing an interface, as if declared like this:

```
class MyClass : IMyInterface {
  public virtual double MyMethod(double x) { return x + 2.1; }
}
```

Generate assembly, module, class and method; create instance, cast to interface, and call method:

```
AssemblyName assemblyName = new AssemblyName();
assemblyName.Name = "myassembly";
AssemblyBuilder assemblyBuilder =
  AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
ModuleBuilder moduleBuilder = assemblyBuilder.DefineDynamicModule("mymodule");
TypeBuilder typeBuilder =
  moduleBuilder.DefineType("MyClass", TypeAttributes.Class | TypeAttributes.Public,
                           typeof(Object));
typeBuilder.AddInterfaceImplementation(typeof(IMyInterface));
{ ConstructorBuilder constructorBuilder =
    typeBuilder.DefineConstructor(MethodAttributes.Public, CallingConventions.HasThis,
                                  new Type[] { });
  ILGenerator ilg = constructorBuilder.GetILGenerator();
  ilg.Emit(OpCodes.Ldarg_0);      // push this
  ilg.Emit(OpCodes.Call, typeof(Object).GetConstructor(new Type[] {}));
  ilg.Emit(OpCodes.Ret);
}
{ MethodBuilder methodBuilder =
    typeBuilder.DefineMethod("MyMethod",
                             MethodAttributes.Virtual | MethodAttributes.Public,
                             typeof(double),
                             new Type[] { typeof(double) });
  ILGenerator ilg = methodBuilder.GetILGenerator();
  ilg.Emit(OpCodes.Ldarg_1);
  ilg.Emit(OpCodes.Ldc_R8, 2.1);
  ilg.Emit(OpCodes.Add);
  ilg.Emit(OpCodes.Ret);
}
Type ty = typeBuilder.CreateType();
Object obj = ty.GetConstructor(new Type[] {}).Invoke(new Object[] { });
IMyInterface mm = (IMyInterface)obj;
mm.MyMethod(5.7);
```


Approach 2: Generating a static method in an existing module, as if declared like this:

```
public static double MyMethod(double x) { return x + 2.1; }
```

The generated method can be wrapped as a delegate of type D2D:

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

Generate dynamic method, create delegate, and call it:

```
DynamicMethod methodBuilder =
  new DynamicMethod("MyMethod",
                    typeof(double),
                    new Type[] { typeof(double) },
                    typeof(String).Module);
ILGenerator ilg = methodBuilder.GetILGenerator();
ilg.Emit(OpCodes.Ldarg_0);
ilg.Emit(OpCodes.Ldc_R8, 2.1);
ilg.Emit(OpCodes.Add);
ilg.Emit(OpCodes.Ret);
D2D mm = (D2D)methodBuilder.CreateDelegate(typeof(D2D));
mm(5.7);
```


Approach 3: Generating virtual method in class implementing generic interface instance:

We want a method, as if declared like this:

```
class MyClass : IFun<double,double> {
  public virtual double Invoke(double x) { return x + 2.1; }
}
```

The generic interface `IFun<A,R>` describes functions of type `A -> R`:

```
public interface IFun<A,R> {
  R Invoke(A x);
}
```

Generate class and method; create instance, cast to interface, and call method. An anonymous delegate generates the method body:

```
IFun<double,double> mm
  = RtcgFun<double,double>(delegate(ILGenerator ilg) {
                             ilg.Emit(OpCodes.Ldarg_1);
                             ilg.Emit(OpCodes.Ldc_R8, 2.1);
                             ilg.Emit(OpCodes.Add);
                             ilg.Emit(OpCodes.Ret);
                           });
mm.Invoke(5.7);
```


The implementation of approach 3

Method `MethodObject` encapsulates most of the boilerplate from approach 1.

```
static Object MethodObject(ILFiller ilFiller, Type iface,
                           Type rTy, params Type[] argTy) { ... }
```

The method makes a class as if declared like this:

```
public class ClassK : iface {
  public virtual rTy Invoke(argTy) { <<body generated by ilFiller>> }
}
```

and then creates and returns an instance of it.

Method `RtcgFun<A,R>` calls `MethodObject` and casts the resulting object to a suitable interface:

```
public static IFun<A,R> RtcgFun<A,R>(ILFiller ilFiller) {
  return (IFun<A,R>)
     MethodObject(ilFiller, typeof(IFun<A,R>), typeof(R), typeof(A));
}
```

This is typesafe and makes essential use of reflection on generic type instances.

This works in C#/CLR, but not in Java/JVM due its ‘generics by erasure’ implementation.

Case study: The Advanced Encryption Standard (AES, Rijndael)

US Federal standard for sensitive information, since May 2002.

AES is a block cipher with 128-bit blocks, and key size 128, 192, or 256 bit.

(1) Given a key, generate an array `rk[0..ROUNDS]` of round keys, where `ROUNDS` = 10, 12, or 14.

(2) For each 128-bit data block `d` to encrypt, do:

(2.1) Xor first round key `rk[0]` into the data block `d`.

(2.2) For the middle rounds `r = 1..ROUNDS-1` do:

```
Substitution(d, S)         // S defines an invertible affine mapping
ShiftRow(d)                // rotate each row by a different amount
MixColumn(d)               // transform columns by polynomial mult.
KeyAddition(d, rk[r])      // xor round key rk[r] into the data block
```

(2.3) The last round, with `r = ROUNDS`, is like (2.2) but has no `MixColumn`.

All operations can be implemented using bitwise operations (shift, xor, or), and some auxiliary tables.

Potential for staging:

Step (1) is performed once for a given key, and step (2) is performed for each data block: many times.

A direct but naïve implementation of the AES middle rounds (2.2)

The 128 bit data block to encrypt is in `a[0..3]`, a four-element array of 32-bit unsigned integers.

```
for(int r = 1; r < ROUNDS; r++) {
  k = rk[r];
  uint[] t = new uint[4];
  for (int j = 0; j < 4; j++) {
    uint res = k[j];
    for (int i = 0; i < 4; i++)
      res ^= T[i][(a[(i + j) % 4] >> (24 - 8 * i)) & 0xFF];
    t[j] = res;
  }
  a[0] = t[0]; a[1] = t[1]; a[2] = t[2]; a[3] = t[3];
}
```

The round keys are in array `rk[0..ROUNDS]`.

The arrays `T[0..3]` are precomputed from table `S`.

Hand-optimized implementation of the AES middle rounds (AES submission)

```
for(int r = 1; r < ROUNDS; r++) {
  k = rk[r];
  uint t0 =
    T0[a0 >> 24] ^
    T1[(a1 >> 16) & 0xFF] ^
    T2[(a2 >> 8) & 0xFF] ^
    T3[a3 & 0xFF] ^ k[0];
  uint t1 =
    T0[a1 >> 24] ^
    T1[(a2 >> 16) & 0xFF] ^
    T2[(a3 >> 8) & 0xFF] ^
    T3[a0 & 0xFF] ^ k[1];
  uint t2 =
    T0[a2 >> 24] ^
    T1[(a3 >> 16) & 0xFF] ^
    T2[(a0 >> 8) & 0xFF] ^
    T3[a1 & 0xFF] ^ k[2];
  uint t3 =
    T0[a3 >> 24] ^
    T1[(a0 >> 16) & 0xFF] ^
    T2[(a1 >> 8) & 0xFF] ^
    T3[a2 & 0xFF] ^ k[3];
  a0 = t0; a1 = t1; a2 = t2; a3 = t3;
}
```

The inner loops have been unrolled, the data block array `a[0..3]` replaced by variables `a0`, . . . , `a3`, and 2D table `T[0..3]` has been replaced by 1D tables `T0`, . . . , `T3`.

A runtime code generator for optimized AES

```
for (int r = 1; r < ROUNDS; r++) {
  k = rk[r];
  for (int j = 0; j < 4; j++) {
    ilg.Emit(OpCodes.Ldc_I4, k[j]);          // Push k[j]
    for (int i = 0; i < 4; i++) {
      ilg.Emit(OpCodes.Ldloc, T[i]);
      ilg.Emit(OpCodes.Ldloc, a[(i+j) % 4]);
      if (i != 3) {
        ilg.Emit(OpCodes.Ldc_I4, 24 - 8 * i);
        ilg.Emit(OpCodes.Shr_Un);
      }
      if (i != 0) {
        ilg.Emit(OpCodes.Ldc_I4, 0xFF);
        ilg.Emit(OpCodes.And);
      }
      ilg.Emit(OpCodes.Ldelem_U4);
      ilg.Emit(OpCodes.Xor);
    }
    ilg.Emit(OpCodes.Stloc, t[j]);           // Assign to tj
  }
  for (int j = 0; j < 4; j++) {              // Generate: a0=t0; a1=t1; ...
    ilg.Emit(OpCodes.Ldloc, t[j]);
    ilg.Emit(OpCodes.Stloc, a[j]);
  }
}
```

Structure is similar that of naive algorithm, but performance better than the hand-optimized one.

Performance of specialized AES

No computations depend only on the round key `rk[r]`, so one should expect very little speed-up.

But improved instruction scheduling and pipeline usage (?) do give a speed-up:

	1.6 Pentium M		2.8 Pentium 4	
	MS CLR	Mono	MS CLR	Mono
Hand-optimized	201	200	327	320
Specialized (RTCG)	260	255	465	386

Encryption speed in Mbit/s: Bigger is better.

(MS CLR 2.0 beta 2 on Windows 2000 and Mono 1.1.9 on Linux; 1.6 GHz P M and 2.8 GHz P 4).

The use of `DynamicMethod` would give a performance loss of 7.5%.

Best commercial C implementation for 1.6 GHz Pentium M encrypts ca. 560 Mbit/s.

Approximately 3 KB of bytecode is generated for each encryption key.

Bytecode generation and just-in-time compilation takes less than 10 ms and ca. 12.7 KB space for each key.

Sparse matrix multiplication

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 B column and then multiply with A's rows.

Assume we need to compute $A \cdot B$ for fixed sparse B and many different non-sparse A.

Potential for staging:

(1) Given B, generate specialized B-multiplier; (2) apply to all relevant A matrices.

Unrolling the `j` loop is essential because it selects the columns of the given B.

Unrolling the `i` loop would generate much more code and offers little extra benefit; so don't do it.

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	2.002	1.758	1.105	1.410	19.998	16.178	10.595	14.340
Sparse	1.005	0.480	0.677	0.951	10.047	4.526	7.151	9.654
Sparse, RTCG	0.256	0.703	1.296	0.290	1.057	7.450	1.823	0.851

Matrix multiplication speed in seconds: Smaller is better.

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

It takes 3.1 ms to generate the B-multiplier 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.

Severe usability problems: Simple (type) mistakes are hard to debug

- MS CLR, applying instruction to too few arguments:

```
Unhandled Exception: System.InvalidProgramException:
    Common Language Runtime detected an invalid program.
 at MyClass.MyMethod(Double )
 at RTCG2.Main(String[] args)
```

- MS CLR, adding integer to double:

  No message, just a meaningless result.

- MS CLR, another typical error message:

```
Operation could destabilize the runtime.
```

- GNU Lightning, using the wrong macro:

```
jit_addr_i(JIT_R0, JIT_R0, JIT_R1);   // Effect: R0 = R0 + R1

jit_addi_i(JIT_R0, JIT_R0, JIT_R1);   // Effect: R0 = R0 + 66
```

- Sun Hotspot JVM with `gnu.bytecode`, applying instruction to too few arguments; almost helpful:

```
Exception in thread "main" java.lang.Error:
popType called with empty stack MyClass.MyMethod(double)double
```

Conclusions so far

- The JVM and CLR are interesting, widely available platforms supporting runtime code generation.
- Bytecode generation plus JIT compilation provides for portability **and** fast generated code.
- The performance of JIT-based implementations is somewhat unpredictable, due to adaptive optimizations.

Related work

- Lisp/Scheme quasiquotation with (`‘`) and (`,`) and `eval` from MIT Lisp, ca. 1978.
- Untyped two-level C: Tick C (Engler et al. 1996).
- Runtime code generation in Standard ML: Fabius (Leone and Lee 1994).
- Somewhat portable runtime code generation in C: GNU Lightning, seriously untyped.
- 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 2003).
- Runtime specialization in Tempo (Consel).

How to make RTGC on JVM and CLR more usable to average programmers?

- Better source language syntax: multi-stage versions of Java and C#:

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

  Some languages extensions with quasiquotation exist: DynJava (Oiwa 2001); MetaPhor/C# (Neverov and Roe 2004); Genoupe/C# (Draheim 2005); SafeGen/Java (Smaragdakis 2005).

- Other languages on same platforms. Multi-stage F#?
- Much needed: Type systems to help catch errors.

  Ideally: guarantee (1) sensible binding-times, (2) correct structure and (3) correct types of generated code.

  MetaOCaml, Genoupe, SafeGen ... represent work in that direction

- Simpler and better support for calling, and then discarding, generated code.
- Security: who can access and read the code I generate?