

Programming Languages, F2002

Lecture 12, Wednesday 1 May 2002 *

- Recent trends in programming languages
- Partial evaluation and automatic program specialization
- Reflection in C# and Java
- Runtime code-generation in MS Common Language Runtime

T-C

Programming Languages, F2002

Page 12-1

Some trends in programming language implementation

- More execution stages:
 - compilation to bytecode (JavaC)
 - 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 compiled code: bytecode must be verifiable
- Reflection: a running program can inspect its own classes, methods, etc.
- Runtime code generation: a running program can create new classes, methods, etc.
- Stronger type systems in source languages (Generic Java, Generic C#, Cyclone)
- Stronger type systems in intermediate languages (Generic Common Language Runtime)
- Necessary for mobile code: code downloaded from outside should come with 'good behaviour' certificate

T-C

Programming Languages, F2002

Page 12-2

Languages, programs, and interpreters

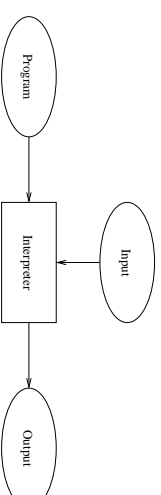
We can abstract a program p by its input-output behaviour.

Let $\llbracket p \rrbracket_L d$ be the result of running program p , written in language L , with input d .

An interpreter int for language S , itself written in language L , takes an S -program and runs it:

$$\llbracket int \rrbracket_L [p, d] = \llbracket p \rrbracket_S [d]$$

Interpreting program p with input d :



T-C

Programming Languages, F2002

Page 12-3

What is a compiler, and why?

A compiler $comp$ from S to T takes an S -program $source$ and makes an equivalent T -program $target$:

$$\llbracket comp \rrbracket_L [source] = target \text{ implies } \llbracket target \rrbracket_T [d] = \llbracket source \rrbracket_S [d]$$

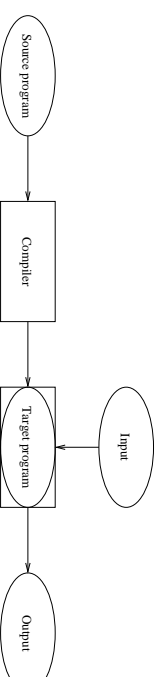
In other words,

$$\llbracket \llbracket comp \rrbracket_L [source] \rrbracket_T [d] = \llbracket source \rrbracket_S [d]$$

So compilation is a two-stage version of interpretation.

Why is compilation useful? Because we may have a real T -machine but only a (slow) interpreter for S .

Compiling program p and then running it with input d :



T-C

Programming Languages, F2002

Page 12-4

Partial evaluation, or automatic program specialization

A function to compute x^n :

```
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;
}
```

Given arguments $n = 5$ and $x = 7$, we can compute `power(5, 7)`, obtaining the result $7^5 = 16807$.

Assume we must compute `power(n, x)` for $n = 5$ and many different values of x .

We can specialize or partially evaluate `power` with $n = 5$, obtaining this *specialized function*:

```
static int power_5(int x) {
    int p;
    p = x;
    x = x * x;
    x = x * x;
    p = p * x;
    return p;
}
```

IT-C

Partial evaluation of interpreters and partial evaluators

What if we apply an interpreter *int* only to the program *p*? That is, we partially evaluate *int* with respect to *p*:

If *target* = $\llbracket spec \rrbracket [int, p]$ then

$$\llbracket target \rrbracket_L [d] = \llbracket spec \rrbracket [int, p] \llbracket_L [d] = \llbracket int \rrbracket_L [p, d] = \llbracket p \rrbracket_S [d]$$

So *target* is an *L*-program equivalent to the *S*-program *p*.

Using partial evaluation, we have compiled the *S*-program to an *L*-program using only an interpreter.

What if we apply the partial evaluator *spec* only to the interpreter (and ignore *p*)?

If *comp* = $\llbracket spec \rrbracket [spec, int]$ then

$$\llbracket comp \rrbracket [p] = \llbracket spec \rrbracket [spec, int] \llbracket [p] = \llbracket int \rrbracket [p, d] = target$$

So *comp* is a compiler from the language *S* to the language *I*.

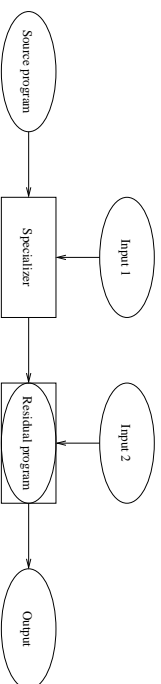
IT-C

Definition of a partial evaluator

A *specializer* or *partial evaluator* *spec* takes a two-argument program *p* and its first input d_1 .

It creates a specialized (residual) program *resid* that when run on d_2 gives the same result as *p* run on (d_1, d_2) .

A *specializer* *spec* satisfies $\llbracket spec \rrbracket [p, d_1] = resid$ implies $\llbracket resid \rrbracket [d_2] = \llbracket p \rrbracket [d_1, d_2]$ for all p, d_1, d_2 .



Or in other words (the 'mix equation'): $\llbracket spec \rrbracket [p, d_1] \llbracket [d_2] = \llbracket p \rrbracket [d_1, d_2]$

A partial evaluator turns a one-stage computation *p* into a two-stage computation by *spec* and then *resid*.

Why useful? Because *resid* is specialized for first input d_1 , and may be much faster than the original *p*.

For instance, *p* may be a regular expression matcher, d_1 a regular expression, and d_2 a string to search.

If the same d_1 is applied to thousands of strings d_2 , it's worth-while to specialize *p* with respect to d_1 .

IT-C

Compiler generation and compiler generators

What if we apply the partial evaluator *spec* only to itself (and ignore *int*)?

If *cogen* = $\llbracket spec \rrbracket [spec, spec]$ then

$$\llbracket cogen \rrbracket [int] = \llbracket spec \rrbracket [spec, spec] \llbracket [int] = \llbracket spec \rrbracket [spec, int] = comp$$

So *cogen* is a compiler generator; it turns interpreters into compilers.

Moreover,

$$\llbracket cogen \rrbracket [spec] = \llbracket spec \rrbracket [spec, spec] = cogen$$

So *cogen* is also a compiler generator generator;

and *spec* is to a compiler generator what an interpreter is to a compiler.

IT-C

History of partial evaluation

Yoshihiko Futamura (Japan, 1971) described compilation by partial evaluation (specialization) of interpreters, and the generation of compilers (single self-application) and compiler generators (double self-application).

From 1972 partial evaluation was studied in Sweden and USSR (Turchin in Moscow and Ershov in Novosibirsk).

The equations for compilation, compiler generation etc were called *Futamura projections* by Ershov.

In 1984, we invented binding-time analysis to guide specialization, and realized self-application for the first time.

Binding-time analysis: automatically classify variables, operators etc as *static* (early) or *dynamic* (late).

Partial evaluators exist for sublanguages of Scheme, C, Standard ML, Haskell, Prolog, Java, ...

Main practical problem: general partial evaluation needs guidance to produce (small) specialized programs.

Literature and references:

Mogensen, Sestoft: Encyclopedia article 1997 (available from course homepage).

Jones, Gomard, Sestoft: Partial Evaluation and Automatic Program Generation, Prentice-Hall 1993.

Available from <http://www.dina.kvl.dk/~sestoft/pebook/pebook.html>

IT-C

Programming Languages, F2002

Page 12-9

How does partial evaluation work?

Binding-time analysis: classify variables and operations as *static* or *dynamic* to guide the specifier.

Static operations are performed (computed) by the specifier.

Dynamic operations cause the specifier to generate code (as part of the specialized program).

Polyvariant specialization: a *program point* is a label, function, ...)

A source program point gives rise to several specialized program points, indexed by static variables' values etc.

Example: The `power` function as a flow-chart program:

```
p = 1
lab1:  if (n <= 0) goto lab3
      if (n % 2 != 0) goto lab2
      x = x * x
      n = n / 2
      goto lab1
lab2:  p = p * x
      n = n - 1
      goto lab1
lab3:  return(p)
```

Computations involving only `n` are static and can be executed by the partial evaluator.

IT-C

Programming Languages, F2002

Page 12-10

Scheme crash course for SML hackers

Construct	Standard ML	Scheme
Variable	<code>x</code>	<code>x</code>
Addition	<code>7 + x</code>	<code>(+ 7 x)</code>
Comparison	<code>7 = x</code>	<code>(eq? 7 x)</code>
Conditional	<code>if 7=x then 11 else 22</code>	<code>(if (eq? 7 x) 11 22)</code>
Nameless function	<code>fn x => 7 + x</code>	<code>(lambda (x) (+ 7 x))</code>
Named function	<code>fun f x = 7 + x</code>	<code>(define (f x) (+ 7 x))</code>
Function call	<code>f 78</code>	<code>(f 78)</code>
Let-binding	<code>let val x = 7 in x + 78 end</code>	<code>(let ((x 7)) (+ x 78))</code>
List	<code>[5, x+4, 2]</code>	<code>(list 5 (+ x 4) 2)</code>
Head of list	<code>hd xs</code>	<code>(car xs)</code>
Tail of list	<code>tl xs</code>	<code>(cdr xs)</code>
A number?		<code>(number? x)</code>
An empty list?		<code>(null? x)</code>
A tree?		<code>(pair? x)</code>

IT-C

Programming Languages, F2002

Page 12-11

Recursive functions in Scheme

```
fun pow 0 x = 1
  | pow n x = if n mod 2 = 0 then pow (n div 2) (x * x)
              else x * pow (n-1) x

(define (pow n x)
  (if (equal? n 0)
      1
      (if (even? n)
          (pow (quotient n 2) (* x x))
          (* x (pow (- n 1) x)))))
```

Scheme data

Three basic data values: a **number** 4, a **symbol** 'foo, the empty tree '(), and a **pair** (v1 . v2).
A list '(1 2 3) is really a right-linear tree '(1 . (2 . (3 . ())))).

Measuring the length of a list:

```
fun length [] = 0
  | length (x::xr) = x + length xr

(define (length xs)
  (if (null? xs)
      0
      (+ 1 (length (cdr xs)))))
```

IT-C

Programming Languages, F2002

Page 12-12

A simple self-applicable partial evaluator for Scheme (JGS1993, appendix A)

The core of the specializer works on expressions annotated as static (ifs) or dynamic (ifc).

Static expressions are evaluated; dynamic expressions give rise to residual code in the specialized program:

```
(define (reduce e xs vs xd vd p)
  (if (op null? e) e
      (if (op boolean? e) e
          (if (op number? e) e
              (if (op atom? e) (call lookupvar e xs vs xd vd)
                  (if (op equal? (tag e) 'quote) (el e)
                      (if (op equal? (tag e) 'ifc)
                          (if (call reduce (el e) xs vs xd vd p)
                              (call reduce (e2 e) xs vs xd vd p)
                              (call reduce (e3 e) xs vs xd vd p))
                          (if (op equal? (tag e) 'ifd)
                              (list 'if (call reduce (el e) xs vs xd vd p)
                                      (call reduce (e2 e) xs vs xd vd p)
                                      (call reduce (e3 e) xs vs xd vd p))
                              (op evalbase (funname e) (call reduce* (callargs e) xs vs xd vd p))
                              (if (op equal? (tag e) 'opd)
                                  (:: 'op (:: (funname e) (call reduce* (callargs e) xs vs xd vd p))))
                              (if (op equal? (tag e) 'lift)
                                  (list 'quote (call reduce (el e) xs vs xd vd p))
                                  )))))))))))
```

FIG Programming Languages, F2002

Page 12-13

Using the self-applicable partial evaluator

Let us partially evaluate the power function.

First a binding-time analysis with n static and x dynamic:

```
(define pa2 (monotone power2 '(s d)))
| ((define ((pow 2) s d) (n) (x))
| (ifs (ops equal? n 0)
| (lift 1)
| (ifs (ops even? n)
| (callid ((pow 2) s d) ((ops quotient n 2)) ((opd * x x)))
| (opd * x (calls ((pow 2) s d) ((ops - n 1)) (x)))))
|
```

Then specialize with respect to n=5:

```
(scheme (spec pa2 '(5)))
| ((define (pow*sd-1 x) (* x (pow*sd-2 (* x x))))
| (define (pow*sd-2 x) (pow*sd-3 (* x x)))
| (define (pow*sd-3 x) (* x (quote 1))))
```

FIG Programming Languages, F2002

Page 12-14

Partially evaluating the partial evaluator

Partially evaluate spec wrt p = power2; result is a generator of specialized versions of power:

```
(define sann (monotone specializer '(s d))) ; BTA of specializer
(define sp2 (spec sann (list pa2))) ; Specialize spec wrt power
(make 'p2gen sp2) ; (sp2 is a power-generator)
(scheme (p2gen '(5))) ; Define sp2 as function p2gen
; Apply it to n=5
| ((define (pow*sd-1 x) (* x (pow*sd-2 (* x x))))
| (define (pow*sd-2 x) (pow*sd-3 (* x x)))
| (define (pow*sd-3 x) (* x (quote 1))))
```

Now partially evaluate spec with respect to p=spec:

```
(define cc (spec sann (list sann))) ; Double self-application
(make 'cogen cc) ; (cc is a compiler generator)
(define cp2 (cogen (list pa2))) ; Define cc as function cogen
(equal? sp2 cp2) ; Apply it to power
(define ccc (cogen (list sann))) ; (cp2 is a power-generator)
(equal? ccc cc) ; (cp2 is identical to sp2)
; Apply cogen to spec
; (ccc is identical to cogen)
```

FIG Programming Languages, F2002

Page 12-15

Reflection in C#: manipulating classes, methods, etc at runtime

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

For every complete-time 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 precisely

Reflection allows a running program to obtain information about its classes, methods, state, ...

Reflection allows a running program to use this information to create instances of a class, call a method, ...

FIG Programming Languages, F2002

Page 12-16

Reflection in C#: calling a method

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;
using System.Reflection;

// For Type
// For MethodInfo
class Reflect1 {
    public static void Main(String [] args) {
        Type ty = typeof(Reflect1);
        MethodInfo m = ty.GetMethod("Foo");
        m.Invoke(null, new Object[] { });
    }
    public static void Foo() { Console.WriteLine("Foo"); }
```

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

The CLR namespace System.Reflection contains many classes related to reflection.

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/RTCG2.cs.
A delegate call is almost as fast as a compiled method call.

FIG

Programming Languages, F2002

Page 12-17

Some practical uses of reflection

Example 1: List all public methods in a class (as in component architectures).

Example 2: Find and call all methods whose name starts with Test (as in JUnit).

```
class Reflect2 {
    public static void Main(String [] args) {
        Type ty = typeof(Reflect2);
        MethodInfo[] ms = ty.GetMethods();
        Console.WriteLine("These static methods are available:");
        for (int i=0; i<ms.Length; i++)
            if (ms[i].IsStatic)
                Console.WriteLine(ms[i].Name);
        Console.WriteLine("");
        Console.WriteLine("Calling static methods whose names start with Test:");
        for (int i=0; i<ms.Length; i++)
            if (ms[i].IsStatic && ms[i].Name.IndexOf("Test") == 0)
                ms[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 TestFoo() { Console.WriteLine("TestFoo"); }
    public static void TestGoo() { Console.WriteLine("TestGoo"); }
    public static void TestFoo2() { Console.WriteLine("TestFoo2"); }
}
```

FIG

Programming Languages, F2002

Page 12-18

Reflection in Java: calling a method

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 CLR machinery is similar to Java's:

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

// For Method
class Reflect1 {
    public static void main(String[] args)
        throws NoSuchMethodException, IllegalAccessException,
            InvocationTargetException {
        Class ty = Reflect1.class;
        Method m = ty.getMethod("Foo", new Class[] { });
        m.invoke(null, new Object[] { });
    }
    public static void Foo() {
        System.out.println("Foo");
    }
}
```

FIG

Programming Languages, F2002

Page 12-19

Runtime code generation in C#: generate new classes, methods, etc. (file rtcg/RTCG1.cs)

Build an assembly containing a module containing a class MyClass containing a method MyMethod.

```
using System; using System.Reflection; using System.Reflection.Emit;
class RTCG1 {
    public static void Main(String [] args) {
        AssemblyName assemblyName = new AssemblyName();
        AssemblyBuilder assemblyBuilder =
            AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
                AssemblyBuilderAccess.Run);
        ModuleBuilder moduleBuilder =
            assemblyBuilder.DefineDynamicModule("myModule");
        TypeBuilder typeBuilder =
            moduleBuilder.DefineType("MyClass",
                TypeAttributes.Class | TypeAttributes.Public,
                typeof(Object));
        MethodBuilder methodBuilder =
            typeBuilder.DefineMethod("MyMethod",
                MethodAttributes.Static | MethodAttributes.Public,
                typeof(void),
                new Type[] { });
        ILGenerator ilg = methodBuilder.GetILGenerator();
        ilg.Emit(OpCodes.Ret);
        Type ty = typeBuilder.CreateType();
        ty.GetMethod("MyMethod").Invoke(null, new Object[] { });
    }
}
```

FIG

Programming Languages, F2002

Page 12-20

Runtime code generation in C#: a method containing a loop (file rtcg/RTCG3.cs)

Consider a simple loop:

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

Building the same method by hand, as public static void MyMethod1(int x) { ... }:

```
MethodBuilder methodBuilder = typeBuilder.DefineMethod("MyMethod1", ...);
ILGenerator ilg = methodBuilder.GetILGenerator();
Label start = ilg.DefineLabel();
ilg.MarkLabel(start);
ilg.Emit(OpCodes.Ldarg_0); // Label start:
ilg.Emit(OpCodes.Ldarg_0);
ilg.Emit(OpCodes.Ldc_I4_1);
ilg.Emit(OpCodes.Sub);
ilg.Emit(OpCodes.Starg_S, 0); // n = n - 1
ilg.Emit(OpCodes.Ldarg_0); // if (n!=0) goto start
ilg.Emit(OpCodes.Brtrue, start); // return
ilg.Emit(OpCodes.Ret);
```

Speed of the runtime-generated code:

- Similar to code compiled from C# code by csc, ca. 400 million iterations/second on 850 MHz Mobile P3.
- However, measurements exhibit incomprehensible but reproducible variations.

Runtime code generation in C#: specialized versions of Power (file rtcg/RTCG4.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); // p = 1;
    while (n > 0) {
        if (n & 2 == 0) {
            ilg.Emit(OpCodes.Ldarg_0);
            ilg.Emit(OpCodes.Ldarg_0);
            ilg.Emit(OpCodes.Ldarg_0);
            ilg.Emit(OpCodes.Mul); // x = x * x
            ilg.Emit(OpCodes.Starg_S, 0); // x = n / 2;
            n = n / 2;
        } else {
            ilg.Emit(OpCodes.Ldloc_0);
            ilg.Emit(OpCodes.Ldarg_0);
            ilg.Emit(OpCodes.Mul); // p = p * x;
            ilg.Emit(OpCodes.Stloc_0); // p = n - 1;
            n = n - 1;
        }
    }
    ilg.Emit(OpCodes.Ldloc_0); // return p;
    ilg.Emit(OpCodes.Ret);
}
```

The specialized Power-method for n=16 is 35% faster than the general one.

Runtime code generation in Java

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

Some tools for runtime code generation in Java:

- 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/`

Practical uses of runtime code generation

- Few, experimental: runtime support has been poor, complicated, and very specific to compiler and machine.
- Examples: bitmap graphics operations; packet filter rules.
- Just-in-time compilers (Sun Hotspot JVM, MS CLR) generate machine code from bytecode at runtime.
- AspectJ, aspect-oriented Java, uses runtime code generation (based on BCEL).
- The .NET Framework regular expression matcher builds an automaton using runtime code generation.
Fragment of `system/text/regularexpressions/regcompilerv.cs` (out of 3069 lines):

```
internal void LeftChar() {
    // Loads the char to the left of the current position
    Idloc(_textV);
    Idloc(_textposV);
    ldc(1);
    Sub();
    CallVirt(_getcharM);
}
```
- The implementation of Generic C# generates specialized field layouts and methods at runtime.
- Managed runtime systems (JVM, CLR) provide a uniform platform, and cheap access to metadata.
- Cisternino (Pisa) and Kennedy (Cambridge) work on a user-friendly CLR runtime code generation interface.