

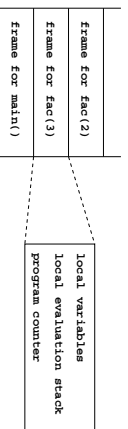
Programming Languages, F2002

Lecture 10, Wednesday 17 April 2002

- The Java Virtual Machine (JVM), a real-world abstract machine
- Compiling micro-C to JVM code
- Microsoft's Common Language Runtime (CLR, .NET)
- The stack, the heap, and garbage collection
- The garbage collectors of Moscow ML and of the Sun HotSpot JVM

The Java Virtual Machine (JVM), a stack abstract machine

Every thread of execution has its own stack, containing stack frames:



A stack frame represents one method that has been called but has not yet returned.

The stack frame contains the method's local variables and parameters, and its local expression evaluation stack.

The JVM bytecode must be verified before execution, requiring for instance:

- all instructions must work on stack operands and local variables of the right type;
- a method must use no more local variables, and no more local stack positions, than it claims to;
- for every given point in the bytecode, the local stack has the same fixed depth whenever that point is reached;
- a method must throw no more exceptions than it claims to;
- the execution of a method must end with a return or throw instruction, not 'fall off the end';
- execution must not use one half of a two-word value (a long or double) as a one-word value.

JVM class files

JVM code and metadata (names and types of fields, methods, etc) are stored in *class files*.

A class file contains:

- the name and package of the class; the superclass, superinterfaces, and access flags (public, ...);
 - the *field declarations* and *method declarations* of the class;
 - the *constant pool* containing field descriptions and method descriptions, string constants, etc.
- For each method declaration (and similarly for field declarations), the class files describes:

- the *name* of the method, the *signature* of the method, and its *access modifiers* (static, public, ...);
- the *attributes*, including the code for the method, and the exceptions thrown by the method.

The *code* for a method is stored in attribute CODE; it includes:

- the maximal depth of the local stack in a stack frame for the method;
 - the number of local variables in the method;
 - the bytecode itself, as a list of JVM instructions;
 - the *exception handlers*, that is, try-catch blocks, of the method body.
- Each handler describes the bytecode range covered by the handler, its entry, and its exception class.

Java Virtual Machine instructions

- push constant onto local stack: `bipush, sipush, iconst, ldc, aconst_null, ...`
 - arithmetic: `iadd, isub, imul, idiv, irem, ineq, inc, fadd, fsub, ...`
 - load local variable onto local stack: `iload, aload, fload, ...`
 - store local variable from local stack: `istore, astore, fstore, ...`
 - load array element onto local stack: `iaload, baload, aaload, faload, ...`
 - stack manipulation: `swap, pop, dup, dup_x1, dup_x2, ...`
 - load field onto local stack: `getfield, getstatic`
 - method call: `invokestatic, invokevirtual, invokespecial`
 - method return: `return, ireturn, areturn, freturn, ...`
 - unconditional jumps: `goto`
 - conditional jumps (compare to 0): `ifeq, ifne, iflt, ifle, ifgt, ifge`
 - conditional jumps (compare two values): `if_icmpeq, if_icmpne, ...`
 - object-related: `new, instanceof, checkcast`
- Instruction prefixes: `i=int, a=object, f=float, d=double, s=short, b=byte, ...`

Generating JVM class files from SML: The SML-JVM toolkit

Manual generation of JVM class files is very cumbersome.

The SML-JVM toolkit helps generating binary JVM class files in SML.

```
fun jvmcompile2file program (classname : string) =
  let open Classdecl LocalVar Label Bytecode
      ..
      (* The method public static void main(String[] args) { ... } *)
      val main : method_decl =
          = freshLocals
          let val locals0
              val (locals1, args) = nextVar1 locals0
              val (code, locals2) = cProgram args locals1 program
          in
              {flags = [Accpublic, Accstatic],
               name = "main",
               msig = (JVMType.Tarray (JVMType.Tclass stringClass)], NONE),
               attrs = [CODE {attrs = [],
                               stack = Stackdepth.maxdepth code [],
                               locals = LocalVar.count locals2,
                               code = code,
                               hdls = []}]
              }
          end
      }
  val myClass = { mdecls = [main, ...], ... } : class_decl
  in ... Classfile.emt ... myClass ... end
```

A compiler from micro-C to JVM bytecode

We compile only a subset of micro-C: no pointers (because of verification), only one function: main.

The compiler (in imp/jvmcomp.sml) is a continuation-based compiler, just as imp/contcomp.sml.

The compiler uses an SML representation JIadd, ... of JVM instructions iadd, ... and JVM class files.

In the compiler, we replace our homegrown stack machine instructions with JVM instructions:

```
fun addCst i C =
  case (i, C) of
    (0, JIadd)   => C1
  | (0, JIsub)   => C1
  | (1, JImul)   => C1
  | (1, JIdiv)   => C1
  | (_, JIpop)   => C1
  | (0, JIfeq lab :: C1) => addCsto lab C1
  | (_, JIfeq lab :: C1) => C1
  | (0, JIfne lab :: C1) => C1
  | (_, JIfne lab :: C1) => addCsto lab C1
  | _            => intConst i :: C?

fun cExpr e (env : envy) (C : jvm_instr list) : jvm_instr list =
  case e of
    Access(AccVar x) => JIload (cAccess x env) :: C
  | ...
  | Cst (Cst i)      => addCst i C
  | ...
```

Compiling integer comparisons in the JVM

In the JVM, integer comparisons (<, <=, =, !=, >=, >) can be made only in conditional jumps.

There are one- and two-argument comparison instructions: less, equal, greater, and their negations.

Thus all in all there are $2 \cdot 3 \cdot 2 = 12$ different instructions: if_icmpeq, ifeq, if_icmpne, ifne, ...

This complicates the code generation for an integer comparison:

- if used in an if- or while-condition, generate a conditional jump;
- if used in a composite logical expression (&& or ||), sometimes generate a conditional jump, sometimes not;
- if assigned to a variable, a 0 (false) or 1 (true) must be generated on the stack;
- if one operand of a comparison is 0, use a one-argument comparison, otherwise a two-argument comparison;
- the usual optimizations should apply if the comparison is followed by a negation, or jump, or ...

Handling all these special cases requires a systematic approach.

Calling the comparison compiler mknumtest from the expression compiler

```
fun cExpr e (env : envy) (C : jvm_instr list) : jvm_instr list =
  case e of
    ...
    | Prim2(ope, e1, e2) =>
        (case ope of
          "!=" => mknumtest env e1 e2 EQUAL true C
        | "!=" => mknumtest env e1 e2 EQUAL false C
        | "<"  => mknumtest env e1 e2 LESS true C
        | "<=" => mknumtest env e1 e2 LESS false C
        | ">"  => mknumtest env e1 e2 GREATER true C
        | ">=" => mknumtest env e1 e2 GREATER false C
        | _    =>
            (cExpr e1 env
             (cExpr e2 env
              (case ope of
                "*" => JImul :: C
                "+" => JIadd :: C
                "-" => JIsub :: C
                "/" => JIdiv :: C
                "%" => JIrem :: C
                _   => raise Fail "unknown primitive 2")))))
```

The ultimate comparison compiler

```

fun mknumtest env e1 e2 cmp sign C =
  let val l1test = ((jiflt, jifge), (* Compare one operand to zero *)
    (jifeq, jifne),
    (jifgt, jifle))
    val i2test = ((jif_lcmplt, jif_lcmpeq), (* Compare two operands *)
    (jif_lcmpeg, jif_lcmgne),
    (jif_lcmglt, jif_lcmple))
    fun rev LESS = GREATER | ... (* Reverse comparison *)
    fun select ((lt, ge), (eq, ne), (gt, le)) ord b =
      case ord of
        LESS => if b then lt else ge
        | EQUAL => if b then eq else ne
        | GREATER => if b then gt else le
    fun genTest b l1l C2 = (* Optimize if one operand is zero *)
      case (e1, e2) of
        (_, Cst (CstI 0)) =>
          |> Cst (CstI 0), _ =>
            |> Cst (CstI 0), _ =>
              |> CExpr e2 env (select l1test (rev cmp) b l1l :: C2)
                |> _
                  |> CExpr e1 env (CExpr e2 env
                    (select i2test cmp b l1l :: C2))
            in
              case C of
                jifne l1l :: C1 => genTest sign l1l C1
                | jifeq l1l :: C1 => genTest (not sign) l1l C1
                | _ => ... Push 0 or 1 onto stack depending on comparison ...
            end
  end

```

FC

The quality of the generated code: example imp/ex1.3.c

```

void main(int n) {
  int y;
  y = 1889;
  while (y < n) {
    y = y + 1;
    if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)
      print y;
  }
}

```

The code generated by our `jvnmcomp` is the same as that generated by `javac` from `imp/ex1.3.java`

Speed of the generated JVM code

Loop executing 20 million times (file `imp/ex8.c`) and `n` queens problem (file `imp/ex1.1.c`):

Target	20 mill loop	12 queens
Interpreting old stack machine code	8.90	29.90
Executing JVM code	0.47	1.91

Why so much faster to execute `Ex1.1.class` than to execute `Machine.class` interpreting `ex1.1.out`?

FC

Interpretive overhead

The `Machine.java` stack machine interpreter spends most of the time figuring out what to do next:

```

for (;;) {
  if (trace)
    printspc(s, bp, sp, p, pc);
  switch (pl[pc++]) {
    case CSTR:
      s[sp+1] = pl[pc++]; sp++; break;
    case ADD:
      s[sp-1] = s[sp-1] + s[sp]; sp--; break;
    case SUB:
      s[sp-1] = s[sp-1] - s[sp]; sp--; break;
    case MUL:
      s[sp-1] = s[sp-1] * s[sp]; sp--; break;
    case DIV:
      s[sp-1] = s[sp-1] / s[sp]; sp--; break;
    case MOD:
      s[sp-1] = s[sp-1] % s[sp]; sp--; break;
    case EQ:
      s[sp-1] = (s[sp-1] == s[sp]) ? 1 : 0; sp--; break;
    case LT:
      s[sp-1] = (s[sp-1] < s[sp]) ? 1 : 0; sp--; break;
    ...
  }
}

```

In seminar 12 we shall see a systematic approach to removing interpretive overhead.

FC

Microsoft Common Language Runtime (CLR) and Common Intermediate Language (CIL)

CLR is part of Microsoft .NET; first released January 2002.

Goals of the CLR:

- provide a safe (managed) execution environment with verification and garbage collection (like the JVM);
- support a range of languages: C#, Visual Basic, JScript, Standard ML, Mercury, Eiffel#... (unlike the JVM);
- support integration of languages: a C# class may extend a Visual Basic class, etc;
- allow unmanaged, unverified execution to support C and C++ pointer arithmetics etc (unlike the JVM).

The .NET Framework SDK implementation of CLR can be downloaded from `msdn.microsoft.com/net/` It is free but requires MS Windows NT/2000/XP

A shared source version of CLR for FreeBSD (Unix) and MS Windows is available also (but is slow).

The Mono project (www.go-mono.com) produces an open source implementation of CLR and a C# compiler.

The CLR code of a program is bundled in an assembly (a so-called `.exe` file), not in several `.class` files. Let us compare the JVM code for file `imp/ex1.3.java` to the CIL code for `imp/ex1.3.cs`.

FC

JVM	CIL	Source
0 aload_0	IL_0000: ldarg.0	args
1 iconst_0	IL_0001: ldc.i4.0	
2 aaload	IL_0002: ldlem.ref	args[0]
3 invokevirtual #17 (...)	IL_0003: call (...)	parseInt
6 istore_1	IL_0008: stloc.0	n = ...
7 sipush 1889	IL_0009: ldc.i4	
10 istore_2	IL_000e: stloc.1	y = 1889;
11 goto.w 45	IL_000f: br.s	
16 iload_2	IL_0011: ldloc.1	
17 iconst_1	IL_0012: ldc.i4.1	
18 iadd	IL_0013: add	y = y + 1;
19 istore_2	IL_0014: stloc.1	
20 iload_2	IL_0015: ldloc.1	
21 iconst_4	IL_0016: ldc.i4.4	
22 irem	IL_0017: rem	
23 ifne 33	IL_0018: brtrue.s	y % 4 == 0
26 iload_2	IL_001a: ldloc.1	
27 bipush 100	IL_001b: ldc.i4.s	
29 irem	IL_001d: rem	
30 ifne 41	IL_001e: brtrue.s	y % 100 != 0
33 iload_2	IL_0020: ldloc.1	
34 sipush 400	IL_0021: ldc.i4	
37 irem	IL_0026: rem	
38 ifne 45	IL_0027: brtrue.s	y % 400 == 0
41 ...	IL_0029: ...	print y
45 iload_2	IL_003e: ldloc.1	
46 iload_1	IL_003f: ldloc.0	
47 if_icmplt 16	IL_0040: blt.s	while (y < n)
52 invokevirtual #27 (...)	IL_0042: call (...)	newline
55 return	IL_0047: ret	

The stack, the heap, and garbage collection

Stack allocation is efficient: allocate = increment stack pointer, deallocate = decrement stack pointer.

But a stack works only if lifetimes are (1) predictable, and (2) nested last-in-first-out.

Some values (arrays, objects, strings, closures, ...) have unpredictable lifetime.

Such values are allocated on a heap.

A heap is a collection of values with addresses; no relation to algorithmics (priority queues).

The running program can allocate values in the heap but cannot explicit deallocate them.

Deallocation is done automatically by a *garbage collector*.

The garbage collector recycles all memory that is not reachable from the *root set*: stack frames and registers.

The running program is sometimes called the *mutator*, in contrast to the *collector*.

Garbage collection has been used since 1960 for functional, object-oriented, and scripting languages.

Java brought garbage collection into mainstream programming.

Pascal, C, C++ are not designed to work with a garbage collector, but can be used with *conservative* collectors.

Mark-sweep garbage collection, and the freelist

The heap consists of used and free blocks.

The free blocks are linked together in a *freelist*.

Allocation: search for a large enough free block on the freelist.

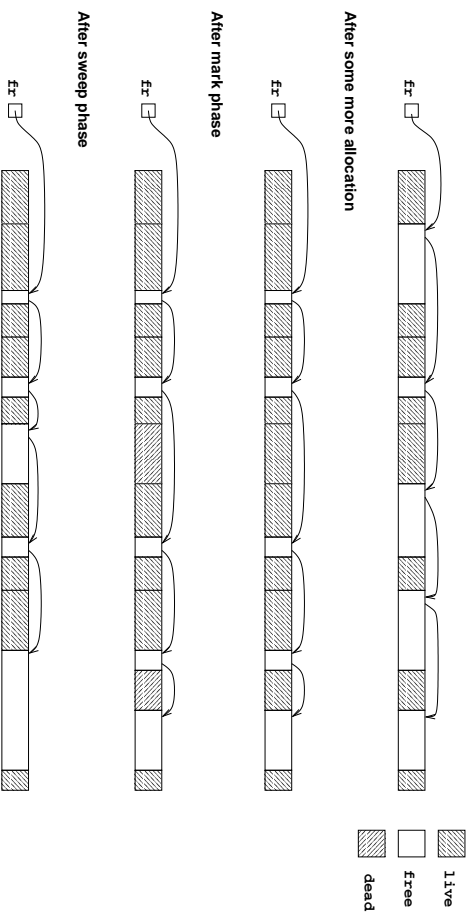
Garbage collection is done in two phases:

- Mark phase: find and mark all values that are reachable from the root set.
- Sweep phase: put unmarked values on the freelist.

Advantages: simple to implement; values do not move.

Disadvantages: heap may become fragmented; searching for a large enough free block may be slow.

Mark-sweep garbage collection



Two-space stop-and-copy garbage collection

The heap is divided into two equally large spaces: *from-space* and *to-space*.

Allocation: allocate in the free part of from-space.

Garbage collection:

All values reachable from the root set are moved from from-space to to-space.

Then the roles of from-space and to-space are swapped.

Advantages: copying will compact the live values; no fragmentation; fast if there is much memory available.

Disadvantages: can use at most half the available memory; values are moved; very slow if there is little memory.

TF-C

Generational garbage collection

Most values die young: old values most likely will live for yet some time.

It is wasteful to move all the old values, just to reclaim the space left by young values.

Solution: Divide the heap into several *generations*.

Allocate in generation 1, the youngest one.

Garbage collection in generation n moves live values to generation $n + 1$.

Consequences:

- Only a few values survive and need to be moved in every 'minor' garbage collection.
- Older generations need not be collected very often.

A complication: assignments may introduce references from an old generation into a younger one.

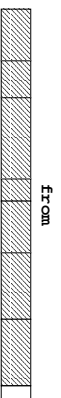
These must be taken into account when collecting the younger generation.

TF-C

Two-space stop-and-copy garbage collection



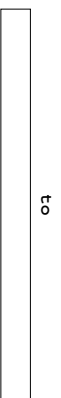
After some more allocation



After copying garbage collection



After swapping from/to



to

to

to

from

TF-C

The garbage collector in Moscow ML

The garbage collector was written for Caml Light by Damien Doligez, INRIA, France.

The heap has two generations.

The garbage collector copies from generation 1 and does incremental mark-sweep collection in generation 2.

Garbage collection in generation 1 (a minor collection) moves live values to generation 2.

Garbage collection in generation 2 is done by *incremental* mark-sweep.

The mark phase and the sweep phase are divided into *slices*.

For every minor collection, a slice of the mark phase or a slice of the sweep phase is done in generation 2.

The division into slices reduces garbage collection pauses and improves real-time response.

A somewhat complicated storage invariant is maintained by the garbage collector.

TF-C

The garbage collector in Sun JDK 1.3.1 Hotspot

The Sun JDK Hotspot 1.3.1 JVM heap has three generations.

A minor collection copies from generation 1 to generation 2.

Generation 2 consists of two semi-spaces with stop-and-copy garbage collection.

Values are moved to generation 3 when they are old enough.

Generation 3 uses (non-incremental) mark-sweep with compaction.

Generation 3 collections can be made incremental by passing the option `-Xincgc` to `java`.