Programs as Data
Real-world abstract machines for Java and C#/.NET.
Garbage collection techniques

Peter Sestoft
Monday 2013-10-21*
Today

• Java Virtual Machine
• .NET Common Language Infrastructure (CLI)
• Garbage collection (GC) techniques
  – Reference-counting
  – Mark-sweep
  – Two-space stop and copy
  – The garbage collectors in JVM and .NET
• List-C, a version of Micro-C with a heap and GC
Example Java program (ex6java.java)

class Node extends Object {
    Node next;
    Node prev;
    int item;
}

class LinkedList extends Object {
    Node first, last;

    void addLast(int item) {
        Node node = new Node();
        node.item = item;
        if (this.last == null) {
            this.first = node;
            this.last = node;
        } else {
            this.last.next = node;
            node.prev = this.last;
            this.last = node;
        }
    }

    void printForwards() { ... }
    void printBackwards() { ... }
}
JVM class file (LinkedList.class)

Generated by javac ex6java.java

Shown by javap -c -v LinkedList

Stack=2, Locals=3, Args_size=2
0 new #2 <Class Node>
3 dup
4 invokespecial #3 <Method Node()> 
7 astore_2
8 aload_2
9 iload_1
10 putfield #4 <Field int item> 
13 ...
### Some JVM bytecode instructions

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>push constant</td>
<td>icost, ldc, aconst_null, ...</td>
</tr>
<tr>
<td>arithmetic</td>
<td>iadd, isub, imul, idiv, irem, ineg, iinc, fadd, ...</td>
</tr>
<tr>
<td>load local variable</td>
<td>iload, aload, fload, ...</td>
</tr>
<tr>
<td>store local variable</td>
<td>istore, astore, fstore, ...</td>
</tr>
<tr>
<td>load array element</td>
<td>iaload, baload, aaload, ...</td>
</tr>
<tr>
<td>stack manipulation</td>
<td>swap, pop, dup, dup_x1, dup_x2, ...</td>
</tr>
<tr>
<td>load field</td>
<td>getfield, getstatic</td>
</tr>
<tr>
<td>method call</td>
<td>invokevirtual, invokevirtual, invokespecial</td>
</tr>
<tr>
<td>method return</td>
<td>return, ireturn, areturn, freturn, ...</td>
</tr>
<tr>
<td>unconditional jump</td>
<td>goto</td>
</tr>
<tr>
<td>conditional jump</td>
<td>ifeq, ifne, iflt, ifle, ...; if_icmpeq, if_icmpne, ...</td>
</tr>
<tr>
<td>object-related</td>
<td>new, instanceof, checkcast</td>
</tr>
</tbody>
</table>

Type prefixes: i=int, a=object, f=float, d=double, s=short, b=byte, ...
JVM bytecode verification

The JVM bytecode is *verified at loadtime*, before execution:

- An instruction must work on stack operands and local variables of the correct type
- A method must use no more local variables and no more local stack positions than it claims to
- For every point in the bytecode, the local stack has the same depth whenever that point is reached
- A method must throw no more exceptions than it admits 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 (e.g. a long) as a one-word value (int)
Additional JVM *runtime* checks

- Array-bounds checks on arr[i]
- Array assignment checks: Can store only subtypes of A into an A[] array
- Null-reference check (a reference is null or points to an object or array, because no pointer arithmetics)
- Checked casts: Cannot make arbitrary conversions between object classes
- Memory allocation succeeds or throws exception
- No manual memory deallocation or reuse

- Bottom line: A JVM program cannot read or overwrite arbitrary memory
- Better debugging, better security
- No buffer overflow attacks, worms, etc as in C/C++
The JVM runtime stacks

• One runtime stack per thread
  – Contains activation records, one for each active function call
  – Each activation record has program counter, local variables, and local stack for intermediate results

<table>
<thead>
<tr>
<th>Frame</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame for fac(0)</td>
<td></td>
</tr>
<tr>
<td>frame for fac(1)</td>
<td></td>
</tr>
<tr>
<td>frame for fac(2)</td>
<td></td>
</tr>
<tr>
<td>frame for fac(3)</td>
<td></td>
</tr>
<tr>
<td>frame for main()</td>
<td></td>
</tr>
</tbody>
</table>

local variables
local evaluation stack
program counter
Example JVM runtime state

```java
void m() {
    LinkedList lst = new LinkedList();
    lst.addLast(5);
    lst.addLast(7);
    Node node = lst.first;
}
```
The .NET Common Language Infrastructure (CLI, CLR)

- Same philosophy and design as JVM
- Some improvements:
  - Standardized bytecode assembly (text) format
  - Better versioning, strongnames, ...
  - Designed as target for multiple source languages (C#, VB.NET, JScript, Eiffel, F#, Python, Ruby, ...)
  - User-defined value types (structs)
  - Tail calls to support functional languages
  - True generic types in bytecode: safer, more efficient, and more complex

- The .exe file = stub + bytecode
- Standardized as Ecma-335
## Some .NET CLI bytecode instructions

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>push constant</td>
<td>ldc.i4, ldc.r8, ldnull, ldstr, ldtoken</td>
</tr>
<tr>
<td>arithmetic</td>
<td>add, sub, mul, div, rem, neg; add.ovf, sub.ovf, ...</td>
</tr>
<tr>
<td>load local variable</td>
<td>ldloc, ldarg</td>
</tr>
<tr>
<td>store local variable</td>
<td>stloc, starg</td>
</tr>
<tr>
<td>load array element</td>
<td>ldelem.i1, ldelem.i2, ldelem.i4, ldelem.r8</td>
</tr>
<tr>
<td>stack manipulation</td>
<td>pop, dup</td>
</tr>
<tr>
<td>load field</td>
<td>ldfld, ldstfld</td>
</tr>
<tr>
<td>method call</td>
<td>call, calli, callvirt</td>
</tr>
<tr>
<td>method return</td>
<td>ret</td>
</tr>
<tr>
<td>unconditional jump</td>
<td>br</td>
</tr>
<tr>
<td>conditional jump</td>
<td>brfalse, brtrue; beq, bge, bgt, ble, blt, ...; bge.un ...</td>
</tr>
<tr>
<td>object-related</td>
<td>newobj, isinst, castclass</td>
</tr>
</tbody>
</table>

Type suffixes: i1=byte, i2=short, i4=int, i8=long, r4=float, r8=double, ...
From Java and C# to bytecode

• Consider the Java/C#/C program ex13:

```java
static void Main(string[] args) {
    int n = int.Parse(args[0]);
    int y;
    y = 1889;
    while (y < n) {
        y = y + 1;
        if (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0))
            InOut.PrintI(y);
    }
    InOut.PrintC(10);
}
```

• Let us compile and disassemble it twice:
  - `javac ex13.java` then `javap -c ex13`
  - `csc /o ex13.cs` then `ildasm /text ex13.exe`
00    aload_0                      |            0000    ldarg.0
01    iconst_0                     |            0001      ldc.i4.0
02    aaload                       |            0002      ldelem.ref
03    invokevirtual parseInt       |            0003      call Parse
06    istore_1                     |            0008      stloc.0
07    sipush 1889                  |            0009      ldc.i4 0x761
10    istore_2                     |            000e      stloc.1
11    iload_2                      |            000f      br 003b
12    iload_1                      |            0015      ldc.i4.1
13    if_icmpge 48                 |            0016      add
17    iconst_1                     |            0017      stloc.1
18    iadd                         |            0018      ldloc.1
19    istore_2                     |            0019      ldc.i4.4
20    iload_2                      |            001a      rem
21    iconst_4                     |            001b      brtrue 003b
22    irem                         |            0020      ldloc.1
23    ifne 11                      |            0021      ldc.i4.s 100
26    iload_2                      |            0023      rem
27    bipush 100                   |            0024      brtrue 0035
29    irem                         |            0025      ldloc.1
30    ifne 41                      |            0029      ldc.i4 0x190
33    iload_2                      |            002a      rem
34    sipush 400                   |            0030      brtrue 003b
37    irem                         |            0035      ldloc.1
38    ifne 11                      |            0036      call PrintI
41    iload_2                      |            003b      ldloc.1
42    invokevirtual printi         |            003c      ldcloc.0
45    goto 11                      |            003d      blt 0014
48    bipush 10                    |            0042      ldc.i4.s 10
50    invokevirtual printc         |            0044      call PrintC
53    return                       |            0049      ret
Ten-minute exercise

• On a printout of the preceding slide
• For both the JVM and the .NET columns
  – Draw arrows to indicate where jumps go
  – Draw blocks around the bytecode segments corresponding to expressions and statements in the ex13.java and ex13.cs programs
Metadata and decompilers

• The .class and .exe files contains metadata: names and types of fields, methods, classes
• One can decompile bytecode into programs:

  ![Diagram](image)

  - .cs
  - csc
  - .exe

  Program → Compiler → Bytecode

  Reflector decompiler

• Bad for protecting your secrets (intellectual property)
• Bytecode obfuscators make decompilation harder
.NET CLI has generic types, JVM doesn’t

class CircularQueue<T> {
    private readonly T[] items;
    public CircularQueue(int capacity) {
        this.items = new T[capacity];
    }
    public T Dequeue() { ... }
    public void Enqueue(T x) { ... }
}

.class CircularQueue`1<T> ... {
    .field private initonly !T[] items ...
    .method !T Dequeue() { ... }
    .method void Enqueue(!T x) { ... }
}

class CircularQueue ... {
    public java.lang.Object dequeue(); ... 
    public void enqueue(java.lang.Object); ... 
}
Consequences for Java

- The Java compiler replaces T
  - with Object in C<T>
  - with Mytype in C<T extends Mytype>
- So this **doesn’t work** in Java, but works in C#:
  - Cast: (T)e
  - Instance check: (e instanceof T)
  - Reflection: T.class
  - Overload on different type instances of gen class:
    ```java
    void put(CircularQueue<Double> cqd) { ... }
    void put(CircularQueue<Integer> cqd) { ... }
    ```
  - Array creation: arr=new T[10]
    So Java versions of CircularQueue<T> must use ArrayList<T>, not T[]
Just-in-time (JIT) compilation

- Bytecode is compiled to real (e.g. x86) machine code at runtime to get speed comparable to C/C++
Just-in-time compilation

• How to inspect .NET JITted code

```csharp
static double Sqr(double x) {
    return x * x;
}
```

```
csc /debug /o Square.cs
```

```
IL_0000:  ldarg.0
IL_0001:  ldarg.0
IL_0002:  mul
IL_0003:  ret
```

```
Mono 3.2.3 MacOS 32 bit
mono -optimize=--inline
-v -v Square.exe
```

```
00 pushl  %ebp
01 movl   %esp,%ebp
03 subl   $0x08,%esp
06 fldl   0x08(%ebp)
09 fldl   0x08(%ebp)
0c fmulp %st,%st(1)
0e leave
0f ret
```

```
movl   %ebp,%esp
popq   %ebp
```
Garbage collection

- A: Reference counting
- B: Mark-sweep
- C: Two-space stop-and-copy, compacting
- D: Generational
- Conservative
The heap as a graph

- The heap is a graph: node=object, edge=reference
- An object is live if reachable from roots
- Garbage collection roots = stack elements
The freelist

• A freelist is a linked list of free heap blocks:

• Allocation from freelist:
  – Search for a large enough free block
  – If none found, do garbage collection
  – Try the search again
  – If it fails, we are out of memory
A: Reference counting with freelist

- Each object knows the number of references to it
- Allocate objects from the freelist
- After assignment `x=o;` the runtime system
  - Increments the count of object `o`
  - Decrements the count of `x`’s old reference (if any)
  - If that count becomes zero,
    - put that object on the freelist
    - recursively decrement count of all objects it points to

- Good
  - Simple to implement

- Bad
  - Reference count field takes space in every object
  - Reference count updates and checks take time
  - A cascade of decrements takes long time, gives long pause
  - Cannot deallocate cyclic structures
B: Mark-sweep with freelist

• Allocate objects from the freelist
• GC phase 1: mark phase
  – Assume all objects are white to begin with
  – Find all objects that are reachable from the stack, and color them black
• GC phase 2: sweep phase
  – Scan entire heap, put all white objects on the freelist, and color black objects white
• Good
  – Rather simple to implement
• Bad
  – Sweep must look at entire heap, also dead objects; inefficient when many small objects die young
  – Risk of heap fragmentation
C: Two-space stop and copy

- Divide heap into to-space and from-space
- Allocate objects in from-space
- When full, recursively move all reachable objects from from-space to the empty to-space
- Swap (empty) from-space with to-space
- Good
  - Need only to look at live objects
  - Good reference locality and cache behavior
  - Compacts the live objects: no fragmentation
- Bad
  - Uses twice as much memory as maximal live object size
  - Needs to update references when moving objects
  - Moving a large object (e.g. an array) is slow
  - Very slow (much copying) when heap is nearly full
D: Generational garbage collection

- Observation: Most objects die young
- Divide heap into young (nursery) and old generation
- Allocate in young generation
- When full, move live objects to old gen. (minor GC)
- When old gen. full, perform a (major) GC there
- Good
  - Recovers much garbage fast
- Bad
  - May suffer fragmentation of old generation (if mark-sweep)
  - Needs a write barrier test on field assignments:
    After assignment `o.f=y` where `o` in old and `y` in young, need to remember that `y` is live
Conservative garbage collectors

• Is 0xFFFFFFFFFA on the stack an int or a heap ref?
• If the GC doesn’t know, it must be conservative: Assume it could be a reference to an object
• Conservative collectors exist as C/C++ libraries

• Good
  – Can be added to C and C++ programs as a library
  – Works even with pointer arithmetics
• Bad
  – Unpredictable memory leaks
  – Cannot be compacting: updating a “reference” that is actually a customer number leads to madness
Concurrent garbage collection

• In a multi-cpu machine, let one cpu run GC
• Complicated
  – Race conditions when allocating objects
  – Race conditions when moving objects
• Typically suspends threads at "GC safe" points
  – May considerably reduce concurrency (because one thread may take long to reach a safe point)
GC in mainstream virtual machines

- Sun/Oracle Hotspot JVM (client+server)
  - Three generations
  - When gen. 0 is full, move live objects to gen. 1
  - Gen. 1 uses two-space stop-and-copy GC; when objects get old they are moved to gen. 2
  - Gen. 2 uses mark-sweep with compaction

- IBM JVM (used in e.g. Websphere server)
  - Highly concurrent generational; see David Bacon’s paper

- Microsoft .NET (desktop+server)
  - Three generation small-obj heap + large-obj heap
  - When gen. 0 is full, move to gen. 1
  - When gen. 1 is full, move to gen. 2
  - Gen. 2 uses mark-sweep with occasional compaction

- Mono .NET implementation
  - Boehm’s conservative collector (still standard May 2012)
  - New two-generational (stop-and-copy plus M-S or S-&-C)
Other GC-related topics

- **Large object space**: Large arrays and other long-lived objects may be stored separately.
- **Weak reference**: A reference that cannot itself keep an object live.
- **Finalizer**: Code that will be executed when an object dies and gets collected (e.g. close file).
- **Resurrection**: A finalizer may make a dead object live again (yrk!).
- **Pinning**: When Java/C# exports a reference to C/C++ code, the object must be pinned; if GC moves it, the reference will be wrong.
GC stress (StringConcatSpeed.java)

• What do these loops do? Which is better?

```java
StringBuilder buf
    = new StringBuilder();
for (int i=0; i<n; i++)
    buf.append(ss[i]);
res = buf.toString();
```

```java
String res = "";
for (int i=0; i<n; i++)
    res += ss[i];
```
New: List-C and the list machine

- list-c = micro-C with Lisp/Scheme data

```c
void main(int n) {
    dynamic xs;
    xs = nil;
    while (n>0) {
        xs = cons(n,xs);
        n = n - 1;
    }
    printlist(xs);
}

void printlist(dynamic xs) {
    while (xs) {
        print car(xs);
        xs = cdr(xs);
    }
}
```

```
x=1 2 3 nil
cons cell car cdr
```
List machine instructions

- List machine = micro-C abstract machine plus six extra instructions:
  - NIL: Put nil reference on stack
  - CONS: Allocate two-word block on heap, put reference to it on stack
  - CAR, CDR: Access word 1 or 2 of block
  - SETCAR, SETCDR: Set word 1 or 2 of block

<table>
<thead>
<tr>
<th>Instr</th>
<th>St before</th>
<th>St after</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 NIL</td>
<td>$s$</td>
<td>$s, nil$</td>
<td>Load nil reference</td>
</tr>
<tr>
<td>27 CONS</td>
<td>$s, v_1, v_2$</td>
<td>$s, p$</td>
<td>Create cons cell $p \rightarrow (v_1, v_2)$ in heap</td>
</tr>
<tr>
<td>28 CAR</td>
<td>$s, p$</td>
<td>$s, v_1$</td>
<td>Component 1 of $p \rightarrow (v_1, v_2)$ in heap</td>
</tr>
<tr>
<td>29 CDR</td>
<td>$s, p$</td>
<td>$s, v_2$</td>
<td>Component 2 of $p \rightarrow (v_1, v_2)$ in heap</td>
</tr>
<tr>
<td>30 SETCAR</td>
<td>$s, p, v$</td>
<td>$s$</td>
<td>Set component 1 of $p \rightarrow _$ in heap</td>
</tr>
<tr>
<td>31 SETCDR</td>
<td>$s, p, v$</td>
<td>$s$</td>
<td>Set component 2 of $p \rightarrow _$ in heap</td>
</tr>
</tbody>
</table>
The structure of the list machine heap

- The heap consists of 32-bit (4-byte) *words*
- The heap is covered by *blocks*
### Garbage collection bits gg

<table>
<thead>
<tr>
<th>Bits</th>
<th>Color</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>white</td>
<td>After mark phase: Not reachable from stack; may be collected</td>
</tr>
<tr>
<td>01</td>
<td>grey</td>
<td>During mark phase: Reachable, referred-to blocks not yet marked</td>
</tr>
<tr>
<td>10</td>
<td>black</td>
<td>After mark phase: Reachable from stack; cannot be collected</td>
</tr>
<tr>
<td>11</td>
<td>blue</td>
<td>On freelist, or is orphan block</td>
</tr>
</tbody>
</table>

- The *mark phase* paints all reachable blocks black
- The *sweep phase* paints black blocks white; paints white blocks blue and puts them on freelist
The freelist; orphans

- All blocks on the freelist are blue (gg=11)
- Word 1 contains a reference to the next freelist element, or nil:

- A block of length zero is an orphan
- It consists of a header only
- (Created by allocating almost all of a block)
- Cannot be on freelist: no room for next ref.
Distinguishing integers and references

- For exact garbage collection we need to distinguish integers from references
- Old trick:
  - Make all heap blocks begin on address that is a multiple of 4; in binary it has form xxxxxxx00
  - Represent integer n as 2n+1, so the integer’s representation has form xxxxxxx1
- Test for IsInt(v): \((v) \& 1 == 1\)
- Tagging an int: \((v) << 1 | 1\)
- Untagging an int: \((v) >> 1\)
An example list-C program, ex30.lc

- Each iteration allocates a cons cell that dies
- Without a garbage collector the program soon runs out of memory

```c
void main(int n) {
    dynamic xs;
    while (n>0) {
        xs = cons(n, 22);
        print car(xs);
        n = n - 1;
    }
}
```

- Your task in BOSC: Implement garbage collectors: mark-sweep, and stop-and-copy
Reading and homework

• This week’s lecture:
  – PLC chapters 9 and 10
  – Sun Microsystems: Memory Management in the Java Hotspot Virtual Machine
  – David Bacon, IBM: Realtime garbage collection
  – Exercise 9.1, and either exercise 9.2 (but many have problems with perfmon) or exercise 9.3 on next slide

• Next week’s lecture:
  – PLC chapter 11
Class SentinelLockQueue contains a memory management problem

Run it and see what happens

Find out what the problem is, explain it, and fix it

The corrected code should run to completion without error

Source code and more explanation is in file QueueWithMistake.java