Abstrakte Maschinen
(Virtuelle Maschinen)

M. Anton Ertl

Was ist eine virtuelle Maschine?

Programmiermodell einer realen Maschine

Aus „Pentium Processor User's Manual, Vol. 3“, Seite 3-1:

- Memory Organization
- Data Types
- Registers
- Instruction format
- Operand selection
- Interrupts and exceptions

entworfen für Implementierung in Hardware
Befehle im RAM für Ausführung
Befehle in ausführbaren Dateien für Speicherung und Übertragung
Befehle in Objektdateien und Libraries zum Linken

Was ist eine virtuelle Maschine? Beispiel: Java VM

- Speicher: Menge von Arrays und Objekten, mit Garbage Collection
- Stack, Locals
- Datentypen
  Stack:
  1 Slot: `int` (i32), `float` (f32), `address`
  2 Slots: `long` (i64), `double` (f64)
  Nur im Speicher: `byte` (i8), `short` (i16), `char` (u16).
  statisches Typechecking (nicht Java Typechecking)
- Befehle: Bytes (bytecode), Immediate-Operanden als Folgebytes
- Exceptions

Zweck von virtuellen Maschinen

- Modulare Programmiersprachenimplementierung
- Schnelle Interpretation (WAM, Java VM anfangs)
  im Gegensatz zu anderen Zwischendarstellungen
  oder Übersetzung in realen Maschinencode (Webassembly)
  Ist LLVM eine VM? Textdarstellung, keine Übersetzung zur Laufzeit
- Übertragungsformat (Java VM, Webassembly, Smalltalk/Squeak)
  Image (Smalltalk) entspricht ausführbarer Datei
  oder Pakete (Java .class) entsprechen DLLs
- oder nur zur Laufzeit (CPython)

Ist eine Virtuelle Maschine nicht sowas wie VirtualBox?

- Im Betriebssystembereich
  Simuliert reale Maschine
  Meist der selbe Befehlssatz
  Betriebssystem für reale Maschine kann darauf laufen
  Hypervisor
  nicht Thema dieser LVA
- Im Programmiersprachenbereich (auch: Abstrakte Maschine, Bytecode)
  Zwischenschicht für die Implementierung
  gewisse Ähnlichkeiten zu realer Maschine
  optimiert für Implementierung in Software
  oder als Zwischenstufe zur Übersetzung
  soll auf verschiedener Hardware effizient sein
  Thema dieser LVA
- Überlappungen
  Sandboxing
  Programmiersprachen-VM als Basis für Betriebssystem (UCSD p-System)

Java VM

- Befehle im Speicher: Interpretation, JIT Compilation
- Befehle in `.class`-Dateien: für dynamisches Linken
- Dauerhaft stabiles Format
  viele Compiler erzeugen es
  viele Implementierungen
  viele Werkzeuge
  maschinenunabhängig
- Unterschiedliche Anforderungen:
  Linken: Referenzen über constant pool
  Interpretation: fixe offsets
  Quickening:
  ersetze z.B. `getfield` mit Referenz auf constant pool
  durch `getfield_quick` mit offset

Java VM

```
static int square(int num) {
    return num * num;
}

static int sumsq(int a, int b) {
    int c=square(b);
    return square(a)+c;
}
```

```
static int square(int);
     0: iload_0
     1: iload_0
     2: imul
     3: ireturn

static int sumsq(int, int);
     0: iload_1
     1: invokestatic #7
     4: istore_2
     5: iload_0
     6: invokestatic #7
     9: iload_2
    10: iadd
    11: ireturn
```

```
class Example2 {
    float x;
    long a[];
    void inca(long inc) {
        for (int i=0; i<a.length; i++) {
            a[i] += inc;
        }
    }
    void inc1() {
        inca(1);
    }
}
```

```
 0: iconst_0
 1: istore_3
 2: iload_3
 3: aload_0
 4: getfield    #7
 7: arraylength
 8: if_icmpge   27
11: aload_0
12: getfield    #7
15: iload_3
16: dup2
17: laload
18: lload_1
19: ladd
20: lastore
21: iinc        3, 1
24: goto        2
27: return
```

```
0: aload_0
1: lconst_1
2: invokevirtual #13
5: return
```

Befehlsübersicht Java VM, Teil 1

- ca. 200 Befehle, codiert als bytes mit immediate-Operanden (big-endian)
- Push Constants: z.B. `bipush`, `ldc1`, `ldc2w`
- Load local, z.B. `iload`, `iload_3`, `aload`
  `wide` prefix-Befehl für locals ab 256
- Store local, z.B. `lstore`, `dstore_3`
- Arrays, z.B. `anewarray`, `caload`, `lastore`
- Fields, z.B. `getfield`, `putstatic`
- Objekte: `new`, `checkcast`, `instanceof`
- Stack-Befehle, z.B. `pop`, `dup`, `dup2_x2`

Befehlsübersicht Java VM, Teil 2

- Arithmetik, z.B. `iadd`, `fmul`, `lrem`
- Bitweise, z.B. `ishr`, `lushr`, `ixor`
- Umwandlung, z.B. `i2l`, `d2f`, `int2char`
- Control transfer, z.B. `ifeq`, `if_icmpeq`, `lcmp`, `goto`
- switch, z.B. `tableswitch`, `lookupswitch`
- Exceptions: `athrow` (try-Block-Information out-of-band)
- Method invocation, z.B. `invokevirtual`, `invokeinterface`
- Return, z.B. `ireturn`
- Monitors: `monitorenter`, `monitorexit`
- Quick-Varianten: z.B. `getfield_quick`, `putfield2_quick`