Virtual Machine

Dr. Edgar Lederer

Machine Specification

- the following descriptions are meant more as **specification** of a machine than an implementation thereof
- we just need a basis against which we can generate code
- the implementations can clearly be optimized

Implementations of Virtual Machine

- two implementations of the (nearly) same VM
  - Haskell
  - Java
- addresses are never used directly, but always via the stack
  - since they are in general computed at run time
  - such things could be clearly optimized (since addresses are often known at compile time)
- a heap pointer is available, but up to now no instructions for the heap
- VM based on Pascal’s P-machine and influenced by JVM

Storage

- two storage areas
  - code: for the program, access via *pc*
  - store: for the data, access via *sp*, *ep*, *hp*, *fp*

```
// stores the program
private IExecInstr[] code;

// program counter
private int pc;
```

Storage

```
// stores the data
// - stack: index 0 upto sp-1
// - heap: index store.length - 1 downto hp+1
private Data.IBaseData[] store;

// stack pointer
// - points to the first currently free location on the stack
// - stack grows from 0 upwards
private int sp;

// extreme pointer
// - points to the first always free location on the stack
private int ep;

// heap pointer
// - points to the first free location on the heap
// - heap grows from store.length - 1 downwards
private int hp;

// frame pointer
// - provides a reference to each routine incarnation
private int fp;
```

Storage

- the number of free locations in store is
  - *numFreeStores* = *hp* - *sp* + 1
- all instructions allocating stack or heap locations perform the test whether there is actually enough space by comparing *sp* and *hp*
- this can be optimized using the **extreme pointer** *ep*
- *ep* always points to the smallest **always** free location on the stack
- its value can be computed at compile time
- then only instructions for entering and exiting routines perform the test for enough space, now by comparing *ep* and *hp*

Initialization

- The compiler should not need a reference to a virtual machine object
- So the compiler generates the instructions without such a reference
- This reference is added when loading the program into the virtual machine

Initialization

```
public VirtualMachine(ICodeArray code, int storeSize)
        throws ExecutionError
{
    loadProgram(code);
    store= new Data.IBaseData[storeSize];
    execute();
}

// pre
// - (forall i | 0 <= i < code.getSize() :
//                         code.get(i) != null)
private void loadProgram(ICodeArray code) {
    this.code= new IExecInstr[code.getSize()];
    for (int i= 0; i < code.getSize(); i++) {
        this.code[i]= code.get(i).toExecInstr(this);
    }
}
```

Data

- the Java machine supports two types of data
  - int
  - float (not completely implemented)
- the Haskell machine supports four types of data
  - int (machine dependent), int32, int64, int1024
- booleans are represented by integers, in the standard way:
  - *false* by 0
  - *true* by 1

Data

```
class Data
{
   static interface IBaseData {}

   static class IntData implements IBaseData
   {
      private int i;
      IntData(int i) { this.i= i; }
      int getData() { return i; }
   }

   static class FloatData implements IBaseData
   { ... }
}
```

- the classes are read-only

Data

- thus, the store simultaneously contains tagged versions of ints and floats
- access to the store usually proceeds by knowing the type of the value to be accessed in advance
- for example, to access a value on top of the stack, one must know in advance whether this value is an int or a float
- if one's assumptions are wrong, a class cast exception is thrown
- if the compiler generates correct code, this exception will never be thrown

Data

```
// precondition: a has type IntData
static int intGet(IBaseData a)
{
    return ((IntData)a).getData();
}

static IntData intAdd(IBaseData a, IBaseData b)
{
    return intNew(intGet(a) + intGet(b));
}
```

Instructions

- each instruction is represented by a class
- the common property of all instructions is simply that they can be **executed**:

```
// executable form of instructions
interface IExecInstr extends IInstr {
    void execute() throws ExecutionError;
}
```

- execution of any instruction might affect:
  - *pc*, *store*, *sp*, *ep*, *hp*, *fp*
  - but not the program, that is, *code*
- execution might throw an execution error, for example, on:
  - division by zero, illegal input, overflow (in Haskell machine)
  - *sp* over *hp* (or *ep* over *hp*)

Main Loop

- the machine as a whole is very simple:

```
private void execute() throws ExecutionError
{
    pc= 0;
    sp= 0;
    ep= 0;
    hp= store.length - 1;
    fp= 0;
    while (pc > -1)
    {
        code[pc].execute();
    }
}
```

- Note: change of *pc* is contained in the individual instructions (otherwise, the loop would loop forever)

A Very Important Instruction

```
// executable form of instructions
interface IExecInstr extends IInstr {
    void execute() throws ExecutionError;
}

// stop instruction
public class StopExec extends Stop
        implements IExecInstr {

    public void execute()
    {
        pc= -1;
    }
}
```

Inner Classes

```
public class VirtualMachine implements IVirtualMachine {

    private Data.IBaseData[] store;
    private int pc, sp, ep, hp;

    public class StopExec extends Stop implements IExecInstr {
        public void execute()
        {
            pc= -1;
        }
    }

    public class LoadImIntExec extends LoadImInt implements IExecInstr {
        public LoadImIntExec(int value) { super(value); }

        public void execute() throws ExecutionError
        {
            // remove following check if use ep
            if (sp > hp) { throw new ExecutionError(SP_OVER_HP); }
            store[sp]= Data.intNew(value);
            sp= sp + 1;
            pc= pc + 1;
        }
    }
}
```

Programming the Machine

```
// non-executable form of instructions
interface IInstr {
    IExecInstr toExecInstr(VirtualMachine vm);
}

// stop instruction
class Stop implements IInstr {
    public String toString() { return "Stop"; }

    public IExecInstr toExecInstr(VirtualMachine vm) {
        return vm.new StopExec();
    }
}
```

Programming the Machine

```
public interface ICodeArray {
    // for the COMPILER:
    // a CodeTooSmallError indicates that the code
    // is too small to hold the complete program
    class CodeTooSmallError extends Exception {}
    void put(int loc, IInstr instr)
        throws CodeTooSmallError;
    void resize();
    String toString();

    // for the VM:
    int getSize();
    IInstr get(int loc);
    // no textual interface available up to now:
    void fromString();
}
```

Programming the Machine

```
… ICodeArray codeArray …

codeArray.put(0, new AllocBlock(storeAddress));
codeArray.put(1, new AllocStack(EXTREME_DUMMY));
int loc= cmd.code(2);
codeArray.put(loc, new Stop());
```

Kinds of Instructions

- monadic operations
  - NegInt
- dyadic operations
  - AddInt, SubInt, MultInt, DivTruncInt, ModTruncInt
  - EqInt, NeInt, LtInt, GeInt, GtInt, LeInt
- jumps
  - UncondJump, CondJump
- input/output
  - InputBool, InputInt, OutputBool, OutputInt
- load/store
  - LoadImInt, LoadAddrRel, Deref, Store
- routine calls/returns
  - AllocBlock, AllocStack, Call, Return
- stop
  - Stop

Further Example: SubInt

```
public class SubIntExec extends SubInt implements IExecInstr {
    public void execute()
    {
        sp= sp - 1;
        store[sp-1]= Data.intSub(store[sp-1], store[sp]);
        pc= pc + 1;
    }
}
```

- Note: for subtraction (and all other non-symmetric operations), the order of operands on the stack matters

Parameters

- some instructions have **parameters**:

```
class UncondJump implements IInstr {

    protected int jumpAddr;

    public UncondJump(int jumpAddr) { this.jumpAddr= jumpAddr; }
    public String toString() { return "UncondJump(" + jumpAddr + ")"; }
    public IExecInstr toExecInstr(VirtualMachine vm) {
        return vm.new UncondJumpExec(jumpAddr);
    }
}

public class UncondJumpExec extends UncondJump implements IExecInstr {

    public UncondJumpExec(int jumpAddr) { super(jumpAddr); }

    public void execute()
    {
        pc= jumpAddr;
    }
}
```

Format for Instructions

- the following slides contain descriptions of all instructions of the VM (in Java)
- we specify the instructions by giving:
  - the parameters
  - the code of the execute method

NegInt

```
public class NegIntExec … {
    public void execute()
    {
        store[sp-1]= Data.intInv(store[sp-1]);
        pc= pc + 1;
    }
}
```

AddInt, SubInt

```
public class AddIntExec … {
    public void execute()
    {
        sp= sp - 1;
        store[sp-1]=
            Data.intAdd(store[sp-1], store[sp]);
        pc= pc + 1;
    }
}

public class SubIntExec … {
    public void execute()
    {
        sp= sp - 1;
        store[sp-1]=
            Data.intSub(store[sp-1], store[sp]);
        pc= pc + 1;
    }
}
```

MultInt, DivTruncInt, ModTruncInt

```
public class MultIntExec … {
    public void execute()
    {
        sp= sp - 1;
        store[sp-1]=
            Data.intMult(store[sp-1], store[sp]);
        pc= pc + 1;
    }
}
```

- DivTruncInt and ModTruncInt analogous
- the truncated versions in the Java version up to now simply since Java / and % are truncated versions
- the Haskell version supports all versions: Euclidean, floored, and truncated

EqInt, NeInt, GtInt, GeInt, LtInt, LeInt

```
public class EqIntExec … {
    public void execute()
    {
        sp= sp - 1;
        store[sp-1]=
            Data.intEQ(store[sp-1], store[sp]);
        pc= pc + 1;
    }
}
```

- NeInt, GtInt, GeInt, LtInt, LeInt analogous
  - Eq: =
  - Ne: ≠
  - Gt: >
  - Ge: ≥
  - Lt: <
  - Le: ≤

UncondJump, CondJump

```
public class UncondJumpExec … {
    public UncondJumpExec(int jumpAddr)
        { super(jumpAddr); }

    public void execute() {
        pc= jumpAddr;
    }
}

public class CondJumpExec … {
    public CondJumpExec(int jumpAddr)
        { super(jumpAddr); }

    public void execute() {
        sp= sp - 1;
        pc= (Data.boolGet(store[sp])) ?
            pc + 1 : jumpAddr;
    }
}
```

InputBool, InputInt

```
public class InputBoolExec … {
    public InputBoolExec(String indicator)
        { super(indicator); }

    public void execute() throws ExecutionError
    {
        System.out.print
            ("? " + indicator + " : bool = ");
        boolean input= InputUtility.readBool();
        int address= Data.intGet(store[sp - 1]);
        store[address]= Data.boolNew(input);
        sp= sp - 1;
        pc= pc + 1;
    }
}
```

- InputInt analogous

OutputBool, OutputInt

```
public class OutputBoolExec … {
    public OutputBoolExec(String indicator)
        { super(indicator); }

    public void execute()
    {
        sp= sp - 1;
        boolean output= Data.boolGet(store[sp]);
        System.out.println
            ("! " + indicator + " : bool = " + output);
        pc= pc + 1;
    }
}
```

- OutputInt analogous

LoadImInt

```
public class LoadImIntExec … {
    public LoadImIntExec(int value) { super(value); }

    public void execute() throws ExecutionError
    {
        // remove following check if use ep
        if (sp > hp)
            { throw new ExecutionError(SP_OVER_HP); }
        store[sp]= Data.intNew(value);
        sp= sp + 1;
        pc= pc + 1;
    }
}
```

LoadAddrRel

```
public class LoadAddrRelExec … {
    public LoadAddrRelExec(int relAddress)
        { super(relAddress); }

    public void execute() throws ExecutionError
    {
        // remove following check if use ep
        if (sp > hp)
            { throw new ExecutionError(SP_OVER_HP); }
        store[sp]= Data.intNew(fp + relAddress);
        sp= sp + 1;
        pc= pc + 1;
    }
}
```

Deref

```
public class DerefExec … {
    public void execute()
    {
        int address= Data.intGet(store[sp - 1]);
        store[sp - 1]= store[address];
        pc= pc + 1;
    }
}
```

- Note: sp remains unmodified

Store

```
public class StoreExec … {
    public void execute()
    {
        int address= Data.intGet(store[sp - 2]);
        store[address]= store[sp - 1];
        sp= sp - 2;
        pc= pc + 1;
    }
}
```

Dup

```
public class DupExec … {
    public void execute() throws ExecutionError
    {
        // remove following check if use ep
        if (sp > hp)
            { throw new ExecutionError(SP_OVER_HP); }
        store[sp]= store[sp - 1].copy();
        sp= sp + 1;
        pc= pc + 1;
    }
}
```

AllocBlock

```
public class AllocBlockExec … {
    public AllocBlockExec(int size) { super(size); }

    public void execute() throws ExecutionError
    {
        sp= sp + size;
        // remove following check if use ep
        if (sp > hp + 1)
            { throw new ExecutionError(SP_OVER_HP); }
        pc= pc + 1;
    }
}
```

AllocStack

```
public class AllocStackExec … {
    public AllocStackExec(int maxSize)
        { super(maxSize); }

    public void execute() throws ExecutionError
    {
        ep= sp + maxSize;
        if (ep > hp + 1)
            { throw new ExecutionError(EP_OVER_HP); }
        pc= pc + 1;
    }
}
```

Call

```
public class CallExec … {
    public CallExec(int routAddress)
        { super(routAddress); }

    public void execute() throws ExecutionError
    {
        // remove following check if use ep
        if (sp + 2 > hp)
            { throw new ExecutionError(SP_OVER_HP); }
        store[sp]= Data.intNew(fp);
        store[sp + 1]= Data.intNew(ep);
        store[sp + 2]= Data.intNew(pc);
        fp= sp;
        sp= sp + 3;
        pc= routAddress;
    }
}
```

Return

```
public class ReturnExec … {
    public ReturnExec(int size) { super(size); }

    public void execute() throws ExecutionError
    {
        sp= fp - size;
        pc= Data.intGet(store[fp + 2]) + 1;
        ep= Data.intGet(store[fp + 1]);
        fp= Data.intGet(store[fp]);
        if (ep > hp + 1)
            { throw new ExecutionError(EP_OVER_HP); }
    }
}
```

Stop

```
public class StopExec … {
    public void execute()
    {
        pc= -1;
    }
}
```