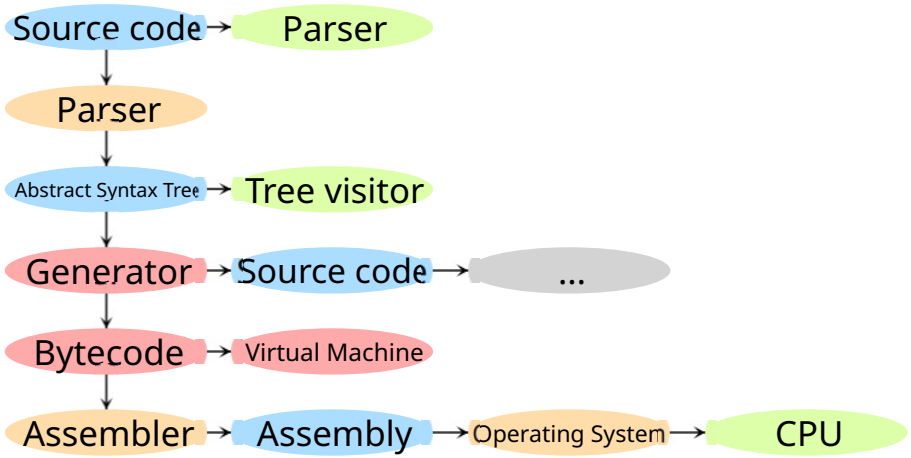
TDDA69 Data and Program Structure
Imperative Programming Implementation

Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

How is a program interpreted?



Lecture content

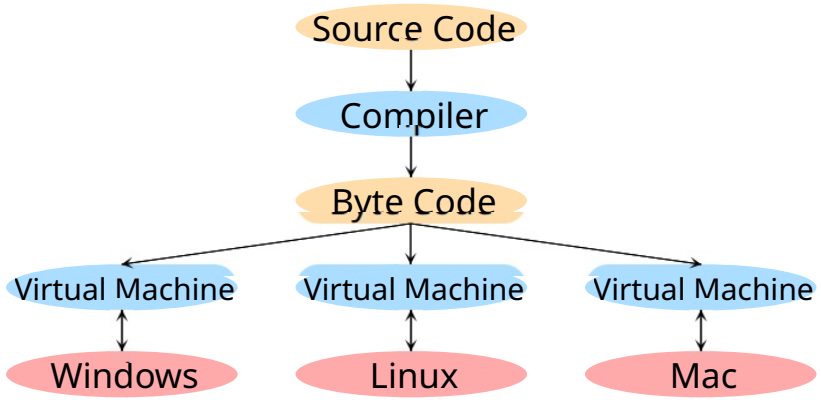
- Virtual Machines
  - Types of virtual/hardware machines
- Bytecode
  - Simple instruction set
  - From AST to Bytecode
  - Bytecode Interpreter

Virtual Machines

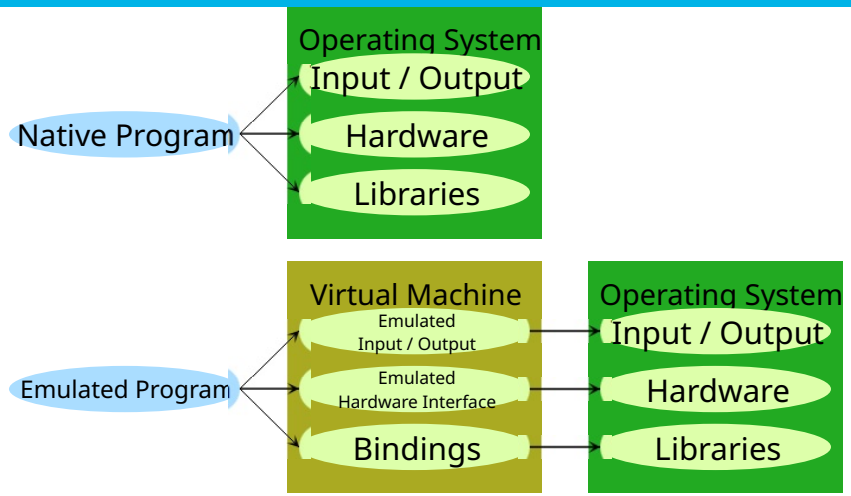
What is a Virtual Machine?

- A **Virtual Machine** is a hardware or software emulation of a real or hypothetical *computer system*
- A *system virtual machine* emulates a complete system and is intented to execute a complete operating system
  - Examples: VirtualBox, VMWare, Parallels...
- A *process/language virtual machine* runs a single program in a single process
  - Examples: JVM, CPython, V8, Dalvik...

Language Virtual Machine



Native Environment vs Emulated Environment



Benefits of Virtual Machines

- **Portability:** Virtual Machines are compatible with various hardware platforms and Operating Systems
- **Isolation:** Virtual Machines are isolated from each other
  - For running incompatible applications concurrently
- **Encapsulation:** computation is seperated from the operating system
  - Beneficial for security

Types of virtual/hardware machines

Types of virtual/hardware machines

- Register Machine
- Stack Machine

Register Machine: Formal definition

- A (in)finite set of **registers**, which holds a single non-negative integer
- An **instruction set**, which defines the operation on registers
  - Arithmetic, control, input/output...
- A **state register**, which holds the current instruction and its index
- Sequential **list of labeled instructions** which defines the program to be executed

Stack Machine: Formal definition

- An (almost) infinite **stack**, which holds integers
- An **instruction set**, which defines the operation on the stack
  - Arithmetic operations are always applied on the top two elements and the results is stored in the top
- A **state register**, which holds the current instruction and its index
- Sequential **list of labeled instructions** which defines the program to be executed

Stack Machines vs Register Machines

- Stack Machines need more compact
  - Arithmetic instruction are smaller
- Stack Machines have simpler compiler, interpreters and a minimal processor
- Stack Machines have a performance disadvantages
  - More memory references, less caching of temporaries
  - Higher cost of factoring out common subexpressions
    It has to be stored as a temporary variable
  - Most common hardware are register machines

Virtual Machines for Dynamic Typing

- Remember:
  - With static typing, types are checked during compilation
  - With dyamic typing, types are checked during execution
- Implication
  - Stack/Registers contains pointer to objects
  - Function call convention

Bytecode

Bytecode

- For a virtual stack machine
- Instruction set
- Generate the bytecode
- Interpreting the bytecode

Simple instruction set

Stack managment

- PUSH [constant_value]
  - Push the constant on the stack
- POP [number]
  - Pop a certain numbers of variables from the stack
- DUP
  - Duplicate the top of the stack
- SWAP
  - Swap the first two items on the stack

Environment

- MAKE_REF [varname]
  - Create a reference to a value/cell in an environment
- STORE
  - Store a value in an environment
- DCL_CELL [varname]
  - Declare a cell in an environment
- DEF_VALUE [varname]
  - Define a value in an environment
- ENV
  - Push the current environment on the stack
- NEW_ENV
  - Create an environment and set it as the current environement
- DROP_ENV
  - Drop the current environment

Jumps

- JMP [idx]
  - Jump to execute instruction at the given index
- IFJMP [idx]
  - Pop the value and if true jump to [idx]
- UNLESSJMP [idx]
  - Pop the value and if false jump to [idx]

Functions

- CALL [arguments]
  - Pop the function object and call it with the given number of arguments
- RET
  - Return
- MAKE_FUNC [#args]
  - Create a new function from the arguments name on the stack and the bytecode

Exceptions

- TRY_PUSH [idx]
  - Indicates the begining of an exception block, idx correspond to the instruction number where to jump if an exception occurs
- TRY_POP
  - Indicates the end of the exception block
- THROW
  - Throw the exception object from the top of the stack

Common extensions

- Arithmetic operators
  - ADD, MUL...
- Array manipulation
- Object Creation
- Call native function

...

From AST to Bytecode

From AST to Bytecode

- With a tree visitor…
- It can take several pass:
  - Find the variables
  - Compute the jumps
  - Generate the code
- The real challenge is to map high level language to instructions

Computing jumps

Instructions are generated in this order

2 IFJMP ?

3 … then statements …

4 JMP ?

5 … else statements …

6 … after statements …

Now we can compute the jump indices

Environment operation

- `define a = 1`

```
PUSH 1
ENV
DEF_VALUE 'a'
```

- `cell a=1`

```
PUSH 1
ENV
DEF_VALUE 'a'
```

- `a`

```
ENV
MAKE_REF'a'
```

- `a = 2`

```
PUSH 2
ENV
MAKE_REF 'a'
STORE
```

- `a.b=c`

```
ENV
MAKE_REF 'c'
ENV
MAKE_REF 'a'
MAKE_REF 'b'
STORE
```

Function call (1/2)

- `func(1,2)`

  PUSH 1
  PUSH 2
  ENV
  MAKE_REF 'func'
  CALL 2

Function call (2/2)

- `console.log('Hello world!')`

  PUSH 'Hello world!'
  ENV
  MAKE_REF 'console'
  MAKE_REF 'log'
  CALL 1

Cond

- `cond(a, { console.log('test') })`

```
ENV
MAKE_REF 'a'
UNLESSJMP +6
PUSH 'test!'
ENV
MAKE_REF 'console'
MAKE_REF 'log'
CALL 1
...
```

- `cond(a, { console.log('hello') }, console.log('world') )`

```
ENV
MAKE_REF 'a'
IFJMP +6
PUSH 'world!'
ENV
MAKE_REF 'console'
MAKE_REF 'log'
CALL 1
JMP +6
PUSH 'hello!'
ENV
MAKE_REF 'console'
MAKE_REF 'log'
CALL 1
...
```


While loop

```
cell a = 10;
while(gt(a,0)) { a =
sub(a, 1); }
PUSH 10
ENV
DCL_CELL 'a'
ENV
MAKE_REF 'a'
STORE
ENV
MAKE_REF 'a'
PUSH 0
```

```
PUSH gt
CALL 2
UNLESSJMP +9
ENV
MAKE_REF'a'
PUSH 1
PUSH sub
ENV
MAKE_REF 'a'
STORE
JMP -13
...
```

Binary representation

- 2 bytes: opcode
- Arguments:
  - integers
  - strings
  - ...

Bytecode Interpreter

Components of bytecode interpreter

- Verifier
  - Bytecode may come from buggy compiler or malicious source
  - Dynamic checking of types, array bounds, function arguments...
- Instruction executer

Verification

- Verifier checks correctness of bytecode
  - Every instruction must have a valid operation code
  - Every instruction must have valid parameters
  - Every branch instruction must branch to the start of some other instruction, not middle of instruction

Bytecode Interpreter

- Standard virtual machine interprets instructions
  - Perform run-time checks such as array bounds and types and function arguments
  - Possible to compile bytecode to native code (JIT: Just-In-Time)
- Call native methods
  - Typically functions written in C

Interpreter loop (1/2)

- instruction pointer (ip): points to current
- stack pointer (sp): topmost item in the operand stack
- current active method or block
- Interpreter
  1. branch to appropriate bytecode routine
  2. fetch next bytecode
  3. increment instruction pointer
  4. execute the bytecode routine
  5. return to 1.

Interpreter loop (2/2)

```
instruction_index = 0
instructions = [ ... ]
stack = [ ]
current_env = Environment()
while instruction_index < len(instructions):
  next_instruction = instructions[instruction_index]
  switch(next_instruction.opcode):
    case ADD:
      ..
    case JMP:
      ...
```


Interpreter Example

Print the absolute value of 4-1:

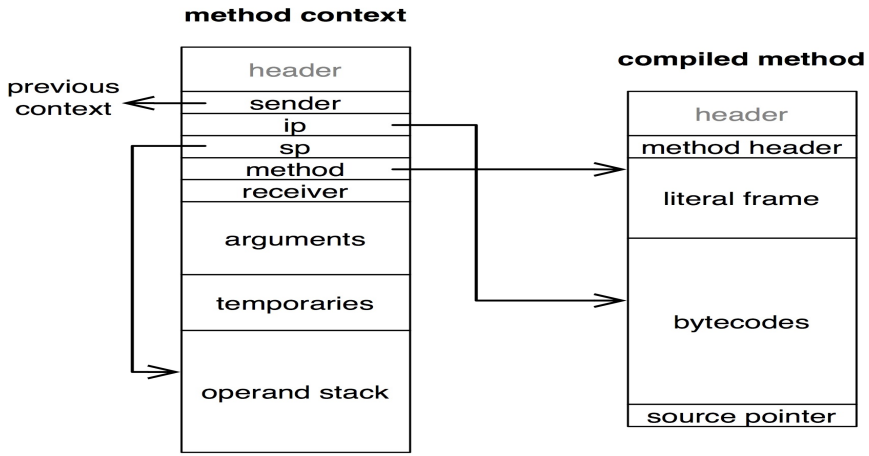
```
1 PUSH 4
2 PUSH 1
3 SUB
4 DUP
5 PUSH 0
6 SUP
7 IFJMP 9
8 NEG
9 LOAD
10 CALL
```

IP=10
Stack = [3, 'print function']

Function Call

- Recursive call
  - For a function call, instantiate a new interpreter loop
- Stack
  - Push on the stack some information on how to restore the interpreter when returning
  - More complex code, but higher performance and flexibility
  - Allow infinite recursion

Contexts



Function Call - Stack

```
instruction_index = 0
instructions = [ ... ]
stack = [ ]
current_env = Environment()
while instruction_index < len(instructions):
  next_instruction = instructions[instruction_index]
  switch(next_instruction.opcode):
     ...
     case CALL:
       env = Environment()
       func = stack.pop()
       for arg in func.args:
         value = stack.pop()
         env.set(arg.name, value)
       stack.push( [ instruction_index, instructions,
current_env ] )
       instruction_index = 0
       instructions = func.instructions
       current_env = env
```

```
case RET:
  retval = stack.pop()
  info = stack.pop()
  stack.push(retval)
  instruction_index = info[0]
  instructions      = info[1]
  current_env       = info[2]
```


Handling Exceptions (1/2)

- Use the implementation language exceptions
- Add exception information to the stack

Handling Exceptions (2/2)

```
instruction_index = 0
instructions = [ ... ]
stack = [ ]
current_env = Environment()
while instruction_index < len(instructions):
  next_instruction = instructions[instruction_index]
  switch(next_instruction.opcode):
      ...
      case TRY_PUSH:
        stack.push(Exception(next_instruction.rescue_index, instructions, current_env))
      case TRY_POP:
        stack.pop()
      case THROW:
        while True:
          info = stack.pop()
          if(info is Exception):
            instructions = info.instructions
            instruction_index = info.rescue_index
            current_env = info.environment
```


Conclusion

- Virtual Machines for interpreting programs
- Stack machines are easier to implement but slower than register machines
- Virtual Machines introduce compilation overhead, but are faster to execute