Helmut Seidl

Virtual Machines

München
Summer 2013
0 Introduction

Principle of Interpretation:

Program + Input  \[\rightarrow\]  Interpreter  \[\rightarrow\]  Output

**Advantage:** No precomputation on the program text  \[\implies\]  no/short startup-time

**Disadvantages:** Program parts are repeatedly analyzed during execution + less efficient access to program variables  \[\implies\]  slower execution speed
Principle of Compilation:

Two Phases (at two different Times):

- Translation of the source program into a machine program (at compile time);
- Execution of the machine program on input data (at run time).
Preprocessing of the source program provides for

- efficient access to the values of program variables at run time
- global program transformations to increase execution speed.

**Disadvantage:** Compilation takes time

**Advantage:** Program execution is sped up \(\Rightarrow\) compilation pays off in long running or often run programs
Structure of a compiler:

Source program → Frontend → Internal representation (Syntax tree)

Optimizations → Internal representation

Code generation → Program for target machine
Subtasks in code generation:

Goal is a good exploitation of the hardware resources:

1. **Instruction Selection:** Selection of efficient, semantically equivalent instruction sequences;

2. **Register-allocation:** Best use of the available processor registers

3. **Instruction Scheduling:** Reordering of the instruction stream to exploit intra-processor parallelism

For several reasons, e.g. modularization of code generation and portability, code generation may be split into two phases:
Intermediate representation $\rightarrow$ Code generation $\rightarrow$ abstract machine code

abstract machine code $\rightarrow$ Compiler $\rightarrow$ concrete machine code

alternatively:

Input $\rightarrow$ Interpreter $\rightarrow$ Output
Virtual machine

- idealized architecture,
- simple code generation,
- easily implemented on real hardware.

Advantages:

- Porting the compiler to a new target architecture is simpler,
- Modularization makes the compiler easier to modify,
- Translation of program constructs is separated from the exploitation of architectural features.
Virtual (or: abstract) machines for some programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>P-machine</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>Bytecode</td>
</tr>
<tr>
<td>Prolog</td>
<td>WAM (&quot;Warren Abstract Machine&quot;)</td>
</tr>
<tr>
<td>SML, Haskell</td>
<td>STGM</td>
</tr>
<tr>
<td>Java</td>
<td>JVM</td>
</tr>
</tbody>
</table>
We will consider the following languages and virtual machines:

<table>
<thead>
<tr>
<th>Language</th>
<th>Virtual Machine</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>CMa</td>
<td>imperative</td>
</tr>
<tr>
<td>PuF</td>
<td>MaMa</td>
<td>functional</td>
</tr>
<tr>
<td>Proll</td>
<td>WiM</td>
<td>logic based</td>
</tr>
<tr>
<td>C±</td>
<td>OMa</td>
<td>object oriented</td>
</tr>
<tr>
<td>multi-threaded C</td>
<td>threaded CMa</td>
<td>concurrent</td>
</tr>
</tbody>
</table>
The Translation of C
1 The Architecture of the CMa

- Each virtual machine provides a set of instructions
- Instructions are executed on the virtual hardware
- This virtual hardware can be viewed as a set of data structures, which the instructions access
- ... and which are managed by the run-time system

For the CMa we need:
The Data Store:

- $S$ is the (data) store, onto which new cells are allocated in a LIFO discipline \(\Rightarrow \) Stack.
- $SP$ (\(\equiv\) Stack Pointer) is a register, which contains the address of the topmost allocated cell,
  
  **Simplification:** All types of data fit into one cell of $S$. 

The Code/Instruction Store:

- \( C \) is the Code store, which contains the program.
  Each cell of field \( C \) can store exactly one virtual instruction.

- \( PC \) (\( \cong \) Program Counter) is a register, which contains the address of the instruction to be executed next.

- Initially, \( PC \) contains the address 0.

\[ \longrightarrow \quad \text{C[0]} \text{ contains the instruction to be executed first.} \]
Execution of Programs:

- The machine loads the instruction in $C[PC]$ into a Instruction-Register IR and executes it.
- PC is incremented by 1 before the execution of the instruction.

```java
while (true) {
    IR = C[PC]; PC++;  
    execute (IR);  
}
```

- The execution of the instruction may overwrite the PC (jumps).
- The Main Cycle of the machine will be halted by executing the instruction `halt`, which returns control to the environment, e.g. the operating system.
- More instructions will be introduced by demand.
2 Simple expressions and assignments

Problem: evaluate the expression \((1 + 7) \times 3\)!

This means: generate an instruction sequence, which

- determines the value of the expression and
- pushes it on top of the stack...

Idea:

- first compute the values of the subexpressions,
- save these values on top of the stack,
- then apply the operator.
The general principle:

- instructions expect their arguments on top of the stack,
- execution of an instruction consumes its operands,
- results, if any, are stored on top of the stack.

Instruction `loadc q` needs no operand on top of the stack, pushes the constant \( q \) onto the stack.

Note: the content of register \( SP \) is only implicitly represented, namely through the height of the stack.