Script generated by TTT

Title: Seidl: Virtual Machines (07.04.2014)

Date: Mon Apr 07 10:16:09 CEST 2014

Duration: 90:21 min

Pages: 27

Helmut Seidl

Virtual Machines

München

Summer 2014

0 Introduction

Principle of Interpretation:

Program + Input ➡ **Interpreter** ➡ Output

Advantage: No precomputation on the program text $\Longrightarrow$ no/short startup-time

Disadvantages: Program parts are repeatedly analyzed during execution + less efficient access to program variables
$\Longrightarrow$ slower execution speed

0 Introduction

Principle of Interpretation:

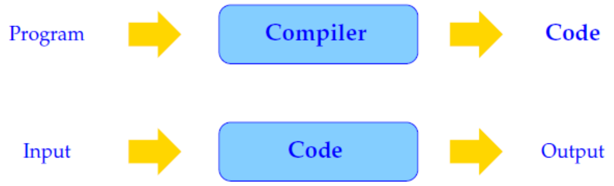
Program + Input ➡ **Interpreter** ➡ Output

Advantage: No precomputation on the program text $\Longrightarrow$ no/short startup-time

Disadvantages: Program parts are repeatedly analyzed during execution + less efficient access to program variables
$\Longrightarrow$ slower execution speed

2

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).

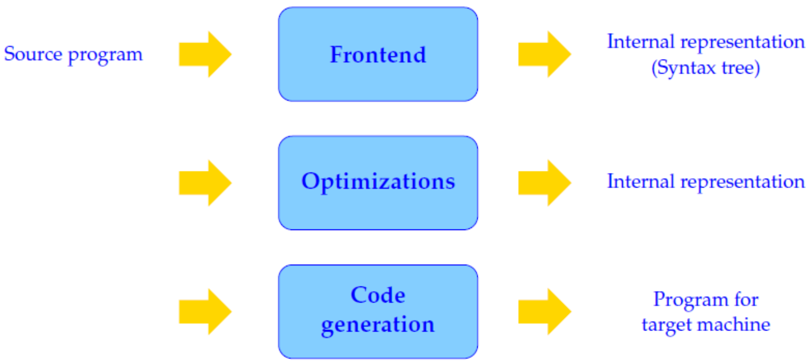
Preprocessing of the source program provides for

- 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 $\Longrightarrow$ compilation pays off in long running or often run programs

Structure of a compiler:

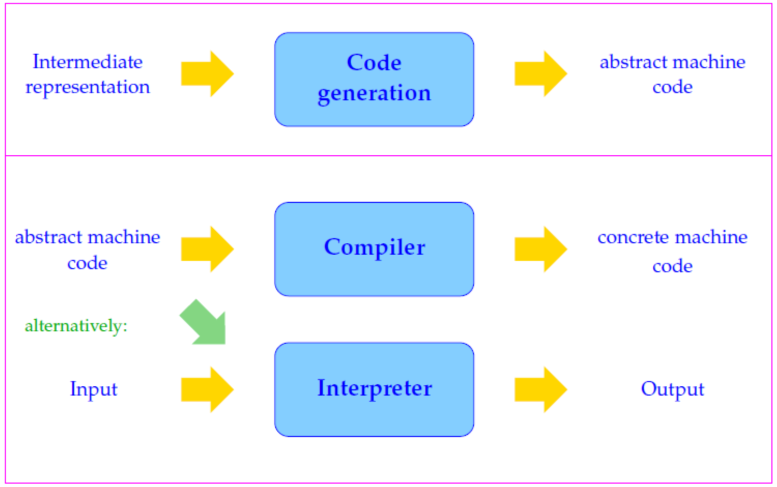


Subtasks in code generation:

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:



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:

Pascal	$\rightarrow$	P-machine	
Smalltalk	$\rightarrow$	Bytecode	
Prolog	$\rightarrow$	WAM	("Warren Abstract Machine")
SML, Haskell	$\rightarrow$	STGM	
Java	$\rightarrow$	JVM	

Virtual (or: abstract) machines for some programming languages:

Pascal	$\rightarrow$	P-machine	
Smalltalk	$\rightarrow$	Bytecode	
Prolog	$\rightarrow$	WAM	("Warren Abstract Machine")
SML, Haskell	$\rightarrow$	STGM	
Java	$\rightarrow$	JVM	

We will consider the following languages and virtual machines:

C	→	CMa	//	*imperative*
PuF	→	MaMa	//	*functional*
Proll	→	WiM	//	*logic based*
C±	→	OMa	//	*object oriented*
multi-threaded C	→	threaded CMa	//	*concurrent*

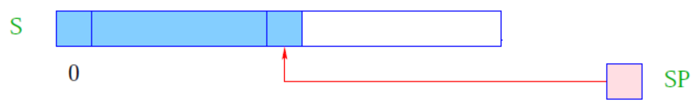
The Translation of C

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

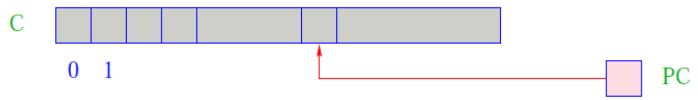
For the CMa we need:

The Data Store:



- S is the (data) store, onto which new cells are allocated in a LIFO discipline
  $\Longrightarrow$ Stack.
- SP ($\widehat{=}$ 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 (≙ Program Counter) is a register, which contains the address of the instruction to be executed next.
- Initially, PC contains the address 0.
  ⟹ C[0] 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

```
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 $(1+7)*3$!

This means: generate an instruction sequence, which

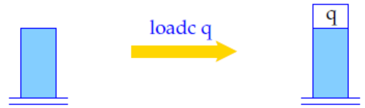
- determines the value of the expression and
- pushes it on top of the stack...

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.




```
SP++;
S[SP] = q;
```

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.

2 Simple expressions and assignments

Problem: evaluate the expression $(1 + 7) * 3$!

This means: generate an instruction sequence, which

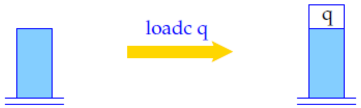
- determines the value of the expression and
- pushes it on top of the stack...

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.



SP++;
S[SP] = q;

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

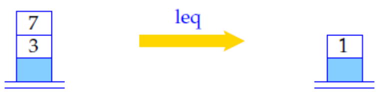
Note: the content of register SP is only implicitly represented, namely through the height of the stack.

SP--;
S[SP] = S[SP] $*$ S[SP+1];

mul expects two operands on top of the stack, consumes both, and pushes their product onto the stack.

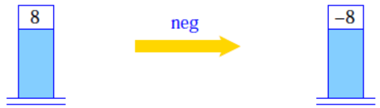
... the other binary arithmetic and logical instructions, add, sub, div, mod, and, or and xor, work analogously, as do the comparison instructions eq, neq, le, leq, gr and geq.

Example: The operator leq



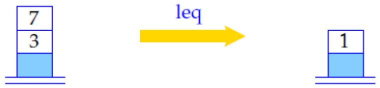
Remark: 0 represents *false*, all other integers *true*.

Unary operators neg and not consume one operand and produce one result.



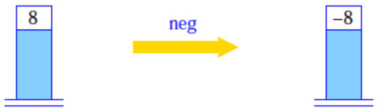
S[SP] = – S[SP];

Example: The operator leq



Remark: 0 represents *false*, all other integers *true*.

Unary operators neg and not consume one operand and produce one result.

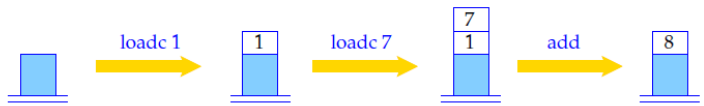


S[SP] = – S[SP];

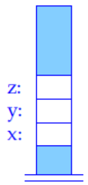
Example: Code for $1 + 7$:

loadc 1 loadc 7 add

Execution of this code sequence:



Variables are associated with memory cells in S:



ρ delivers for each variable x the relative address of x.

ρ is called Address Environment.

Variables can be used in two different ways:

Example: $x = y + 1$

We are interested in the value of y, but in the address of x.

The syntactic position determines, whether the L-value or the R-value of a variable is required.

L-value of x = address of x
R-value of x = content of x

$\text{code}_R\ e\ \rho$	produces code to compute the R-value of e in the address environment ρ
$\text{code}_L\ e\ \rho$	analogously for the L-value

Note:

Not every expression has an L-value (Ex.: $x + 1$).