Script generated by TTT

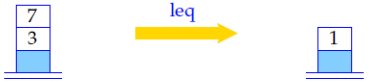
Title: Seidl: Virtual_machines (24.04.2012)

Date: Tue Apr 24 14:04:49 CEST 2012

Duration: 90:33 min

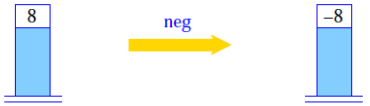
Pages: 33

Example: The operator leq



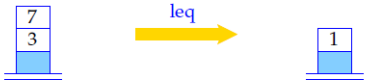
Remark: 0 represents *false*, all other integers *true*.

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



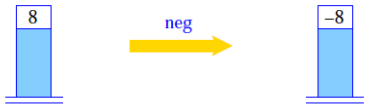
S[SP] = – S[SP];

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

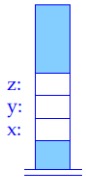
Execution of this code sequence:

Example: Code for $1+7$:

loadc 1 loadc 7 add

Execution of this code sequence:

Variables are associated with cells in S:



Code generation will be described by some Translation Functions, code, code_L, and code_R.

Arguments: A program construct and a function ρ. 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:

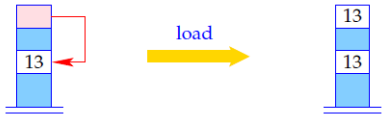
Not every expression has an L-value (Ex.: $x+1$).

We define:

$$
\begin{array}{lcl}
\text{code}_R\ (e_1 + e_2)\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{add} \\
 & & \quad \text{... analogously for the other binary operators} \\
\text{code}_R\ (-e)\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{neg} \\
 & & \quad \text{... analogously for the other unary operators} \\
\text{code}_R\ q\ \rho & = & \text{loadc q} \\
\text{code}_L\ x\ \rho & = & \text{loadc}\ (\rho\, x) \\
 & \ldots &
\end{array}
$$

$$\text{code}_R\ x\ \rho \quad = \quad \text{code}_L\ x\ \rho$$
$$\text{load}$$

The instruction load loads the contents of the cell, whose address is on top of the stack.



S[SP] = S[S[SP]];

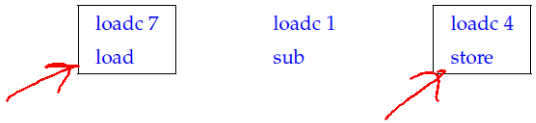
$$\text{code}_R\ (x = e)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{code}_L\ x\ \rho$$
$$\text{store}$$

store writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

Note: this differs from the code generated by gcc ??

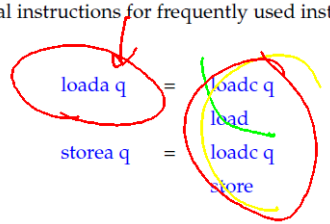
S[S[SP]] = S[SP-1];
SP--;

Example: Code for $e \equiv x = y - 1$ with $\rho = \{x \mapsto 4, y \mapsto 7\}$.
$\text{code}_R\ e\ \rho$ produces:



Improvements:

Introduction of special instructions for frequently used instruction sequences, e.g.,



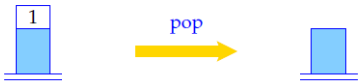
3 Statements and Statement Sequences

Is e an expression, then $e;$ is a statement.

Statements do not deliver a value. The contents of the SP before and after the execution of the generated code must therefore be the same.

$$\text{code}\ e;\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{pop}$$

The instruction pop eliminates the top element of the stack.



SP--;

The code for a statement sequence is the concatenation of the code for the statements of the sequence:

$$\begin{array}{lcll} \text{code}\ (s\ ss)\ \rho & = & \text{code}\ s\ \rho & \\ & & \text{code}\ ss\ \rho & \\ \text{code}\ \varepsilon\ \rho & = & // & \textit{empty sequence of instructions} \end{array}$$

$$\begin{array}{lcl} \text{code}_R\ (x = e)\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{code}_L\ x\ \rho \\ & & \text{store} \end{array}$$

store writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

Note: this differs from the code generated by gcc ??

S[S[SP]] = S[SP-1];

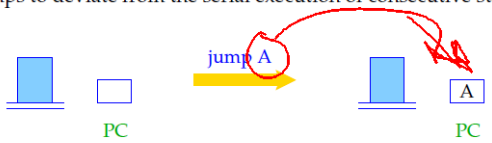
SP--;

The code for a statement sequence is the concatenation of the code for the statements of the sequence:

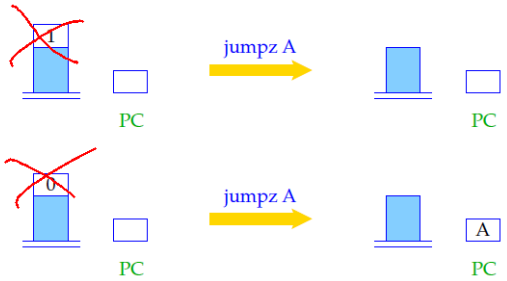
$$\begin{array}{lcll} \text{code}\ (s\ ss)\ \rho & = & \text{code}\ s\ \rho & \\ & & \text{code}\ ss\ \rho & \\ \text{code}\ \varepsilon\ \rho & = & // & \textit{empty sequence of instructions} \end{array}$$

4 Conditional and Iterative Statements

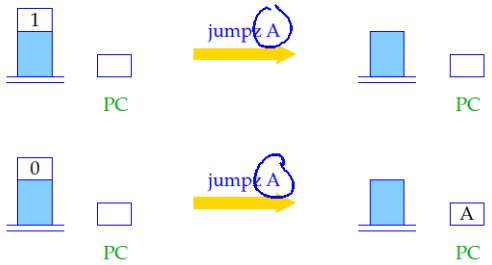
We need jumps to deviate from the serial execution of consecutive statements:



PC = A;



if (S[SP] == 0) PC = A;
SP--;



if (S[SP] == 0) PC = A;
SP--;

For ease of comprehension, we use symbolic jump targets. They will later be replaced by absolute addresses.

Instead of absolute code addresses, one could generate relative addresses, i.e., relative to the actual PC.

Advantages:

- smaller addresses suffice most of the time;
- the code becomes relocatable, i.e., can be moved around in memory.

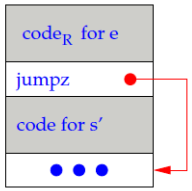
4.1 One-sided Conditional Statement

Let us first regard $s \equiv \textbf{if } e \textbf{ } s$.

Idea:

- Put code for the evaluation of e and s' consecutively in the code store,
- Insert a conditional jump (jump on zero) in between.

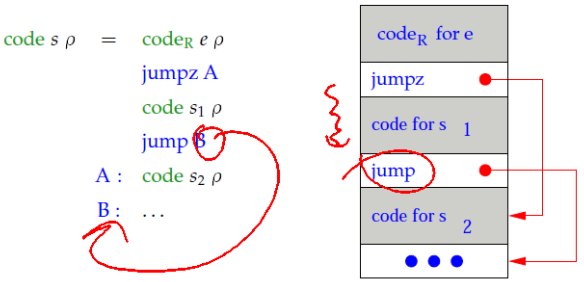
code $s\ \rho$ = code$_R$ $e\ \rho$
jumpz A
code $s'\ \rho$
A : ...



4.2 Two-sided Conditional Statement

Let us now regard $s \equiv$ **if** $(e)\ s_1$ **else** s_2. The same strategy yields:

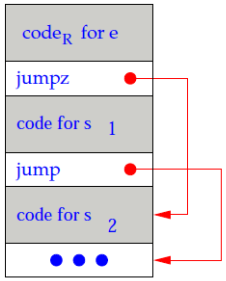
code $s\ \rho$ = code$_R$ $e\ \rho$
jumpz A
code $s_1\ \rho$
jump B
A : code $s_2\ \rho$
B : ...



4.2 Two-sided Conditional Statement

Let us now regard $s \equiv$ **if** $(e)\ s_1$ **else** s_2. The same strategy yields:

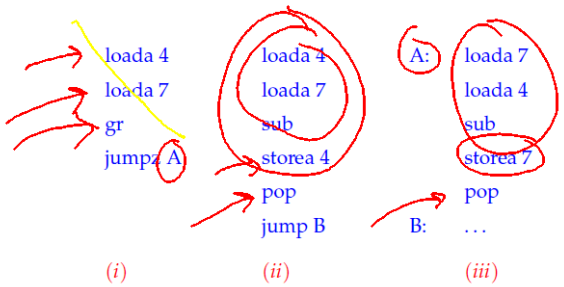
code $s\ \rho$ = code$_R$ $e\ \rho$
jumpz A
code $s_1\ \rho$
jump B
A : code $s_2\ \rho$
B : ...



Example: Be $\rho = \{x \mapsto 4, y \mapsto 7\}$ and

$s \equiv$ **if** $(x > y)$ *(i)*
$x = x - y;$ *(ii)*
else $y = y - x;$ *(iii)*

code $s\ \rho$ produces:

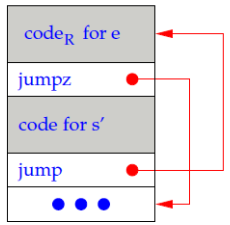


4.3 while-Loops

Let us regard the loop $s \equiv \textbf{while}\ (e)\ s'$. We generate:

code $s\ \rho$ =

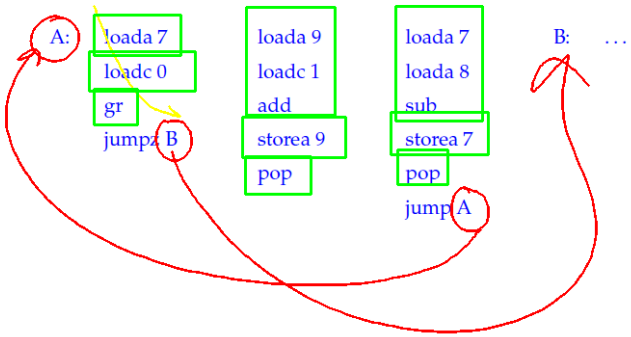
A : code$_R$ $e\ \rho$
jumpz B
code $s'\ \rho$
jump A
B : ...



Example: Be $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ and s the statement:

while $(a > 0)\ \{c = c + 1;\ a = a - b;\}$

code $s\ \rho$ produces the sequence:



4.4 for-Loops

The **for**-loop $s \equiv \textbf{for}\ (e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;\ \textbf{while}\ (e_2)\ \{s'\ e_3;\}$ – provided that s' contains no **continue**-statement.
We therefore translate:

code $s\ \rho$ = code$_R$ e_1
pop
A : code$_R$ $e_2\ \rho$
jumpz B
code $s'\ \rho$
code$_R$ $e_3\ \rho$
pop
jump A
B : ...

4.4 for-Loops

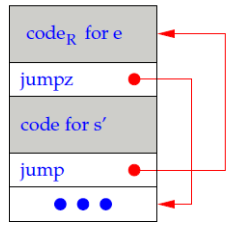
The **for**-loop $s \equiv \textbf{for}\ (e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;\ \textbf{while}\ (e_2)\ \{s'\ e_3;\}$ – provided that s' contains no **continue**-statement.
We therefore translate:

code $s\ \rho$ = code$_R$ e_1
pop
A : code$_R$ $e_2\ \rho$
jumpz B
code $s'\ \rho$
code$_R$ $e_3\ \rho$
pop
jump A
B : ...

4.3 while-Loops

Let us regard the loop $s \equiv$ **while** $(e)\ s'$. We generate:

```
code s ρ  =
        A :  code_R e ρ
             jumpz B
             code s' ρ
             jump A
        B :  ...
```



4.4 for-Loops

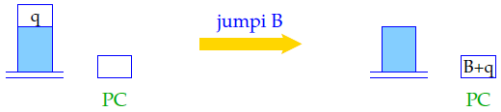
The **for**-loop $s \equiv$ **for** $(e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence e_1; **while** $(e_2)\ \{s'\ e_3;\}$ – provided that s' contains no **continue**-statement.
We therefore translate:

```
code s ρ  =  code_R e1
             pop
        A :  code_R e2 ρ
             jumpz B
             code s' ρ
             code_R e3 ρ
             pop
             jump A
        B :  ...
```


4.5 The switch-Statement

Idea:

- Multi-target branching in constant time!
- Use a jump table, which contains at its i-th position the jump to the beginning of the i-th alternative.
- Realized by indexed jumps.

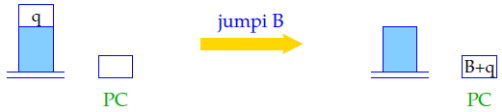


PC = B + S[SP];
SP−;

4.5 The switch-Statement

Idea:

- Multi-target branching in constant time!
- Use a jump table, which contains at its i-th position the jump to the beginning of the i-th alternative.
- Realized by indexed jumps.



PC = B + S[SP];
SP−;

Simplification:

We only regard **switch**-statements of the following form:

$$
\begin{array}{lll}
s & \equiv & \textbf{switch}\ (e)\ \{ \\
 & & \quad \textbf{case}\ 0\text{:}\quad ss_0\ \textbf{break;} \\
 & & \quad \textbf{case}\ 1\text{:}\quad ss_1\ \textbf{break;} \\
 & & \qquad \vdots \\
 & & \quad \textbf{case}\ k-1\text{:}\quad ss_{k-1}\ \textbf{break;} \\
 & & \quad \textbf{default:}\quad ss_k \\
 & & \}
\end{array}
$$

s is then translated into the instruction sequence:


```
code s ρ  =  code_R e ρ        C_0:  code ss_0 ρ      B:  jump C_0
             check 0 k B               jump D              ...
                                       ...                 jump C_k
                               C_k:  code ss_k ρ      D:  ...
                                     jump D
```

- The Macro `check 0 k B` checks, whether the R-value of e is in the interval $[0, k]$, and executes an indexed jump into the table `B`
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the **switch**-statement.