Script generated by TTT

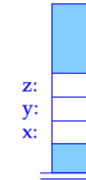
Title: Seidl: Virtual_Machines (20.04.2015)

Date: Mon Apr 20 10:16:15 CEST 2015

Duration: 90:39 min

Pages: 27

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 $\quad x = y + 1$

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:

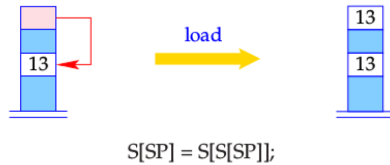
Not every expression has an L-value (Ex.: $x + 1$).

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.

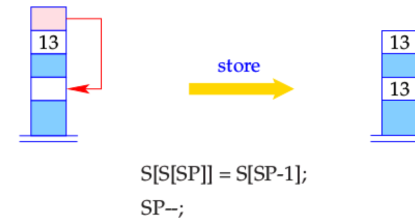


S[SP] = S[S[SP]];

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

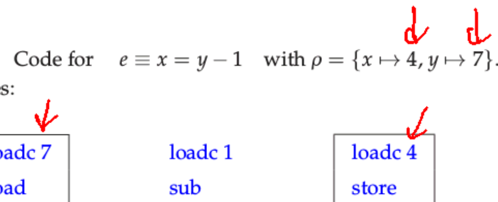
store writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

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



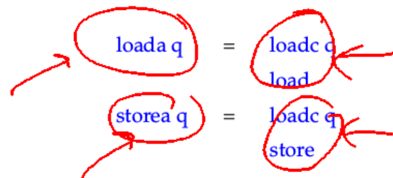
S[S[SP]] = S[SP-1];
SP--;

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.,



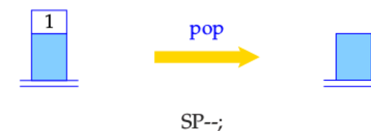
3 Statements and Statement Sequences

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:

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

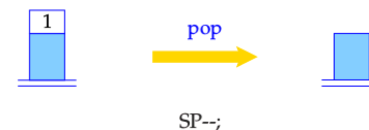
3 Statements and Statement Sequences

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.

$$\begin{array}{lll} \text{code}\ e;\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{pop} \end{array}$$

The instruction pop eliminates the top element of the stack.



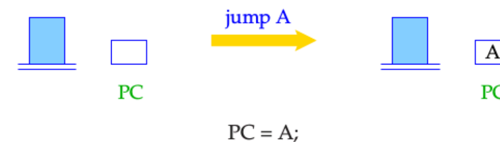
SP--;

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

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

4 Conditional and Iterative Statements

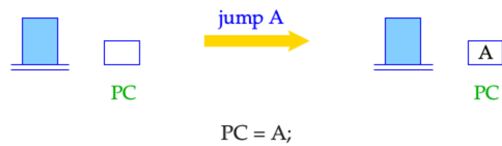
We need jumps to deviate from the serial execution of consecutive statements:

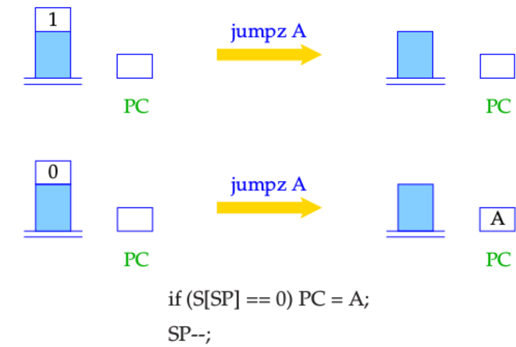


PC = A;

4 Conditional and Iterative Statements

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





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.

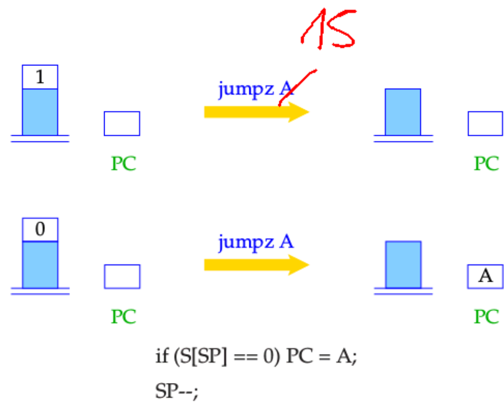
For ease of comprehension, we use symbolic jump targets. They will later be replaced by absolute addresses.

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.

31



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.

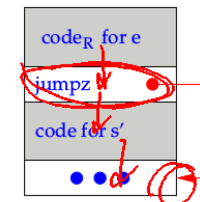
4.1 One-sided Conditional Statement

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

Idea:

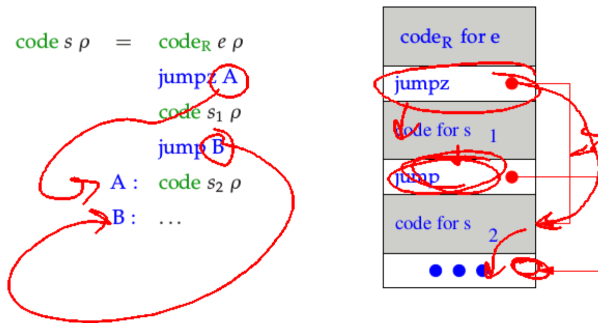
- Put code for the evaluation of e and s' consecutively in the code store,
- Insert a conditional jump (jump on zero) in between.

$$
\begin{array}{rcl}
\text{code } s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code } s'\ \rho \\
\text{A}: & & \ldots
\end{array}
$$



4.2 Two-sided Conditional Statement

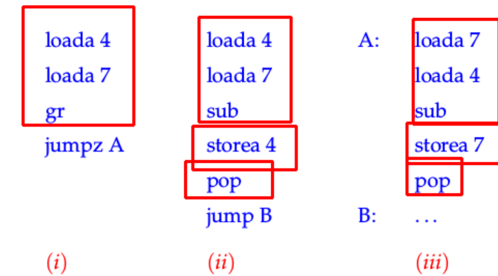
Let us now regard $s \equiv$ **if** (e) s_1 **else** s_2. The same strategy yields:



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

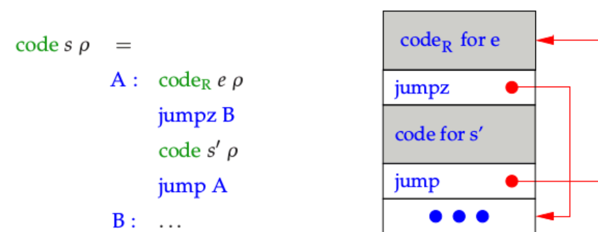
$$s \equiv \textbf{if}\ (x > y) \quad (i)$$
$$x = x - y; \quad (ii)$$
$$\textbf{else}\ y = y - x; \quad (iii)$$

code s ρ produces:



4.3 while-Loops

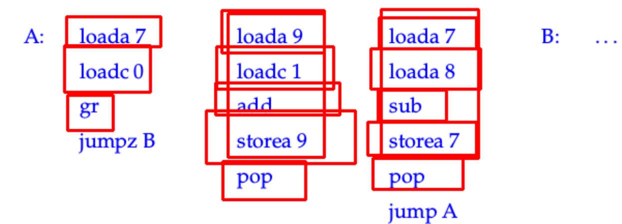
Let us regard the loop $s \equiv$ **while** (e) s'. We generate:



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 ρ produces the sequence:



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

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

code $s\ \rho$ produces the sequence:

A:	loada 7	loada 9	loada 7	B: ...
	loadc 0	loadc 1	loada 8	
	gr	add	sub	
	jumpz B	storea 9	storea 7	
		pop	pop	
			jump A	

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 ρ  =      codeR e1 ρ
                 pop
            A :  codeR e2 ρ
                 jumpz B
                 code s′ ρ
                 codeR e3 ρ
                 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:

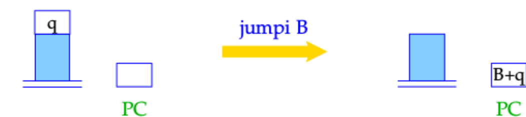
```
code s ρ  =      codeR e1 ρ
                 pop
            A :  codeR e2 ρ
                 jumpz B
                 code s′ ρ
                 codeR e3 ρ
                 pop
                 jump A
            B :  ...
```

38

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--;