Script generated by TTT

Title: Seidl: Virtual_Machines (17.04.2013)

Date: Wed Apr 17 16:02:17 CEST 2013

Duration: 87:08 min

Pages: 36

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.

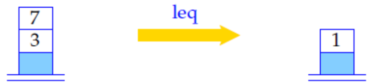
Note: the content of register SP is only implicitly represented, namely through the height of the stack.

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.

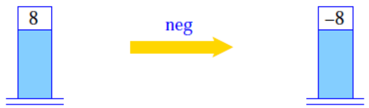
... the other binary arithmetic and logical instructions, add, sub, div, mod, and, or and xor, work analogously, as do the comparison instructions eq, neq, le, leq, gr and geq.

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: Code for $1 + 7$:

loadc 1 loadc 7 add

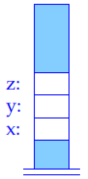
Execution of this code sequence:

Example: Code for $(1 + 7) * 3$

loadc 1 loadc 7 add loadc 22 mult

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

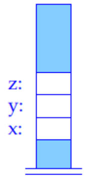
R-value of x = content of x

$\text{code}_R\ e\ \rho$	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.: $x + 1$).

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

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

Note:

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

We define:

code$_R$ $(e_1 + e_2)$ ρ = code$_R$ e_1 ρ
code$_R$ e_2 ρ
add
... analogously for the other binary operators

code$_R$ $(-e)$ ρ = code$_R$ e ρ
neg
... analogously for the other unary operators

code$_R$ q ρ = loadc q

code$_L$ x ρ = loadc $(\rho\, x)$

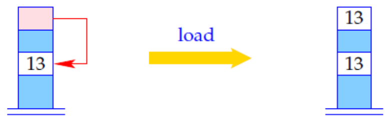
...

Code$_R$ x ρ

code$_R$ x ρ = code$_L$ x ρ
load

loadc (ρ x)

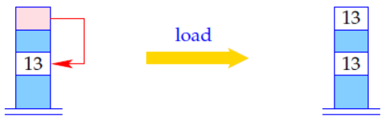
The instruction load loads the contents of the cell, whose address is on top of the stack.



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

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

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

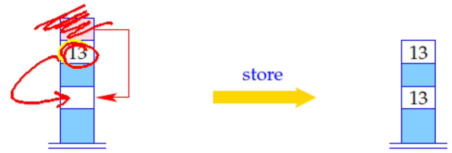


S[SP] = S[S[SP]];

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

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:

loadc 7 load	loadc 1 sub	loadc 4 store

Improvements:

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

loada q	=	loadc q load
storea q	=	loadc q store

x = y = z = 0;   1;   x = 1;

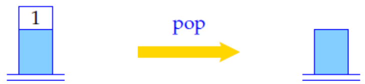
3 Statements and Statement Sequences

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.

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

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:

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

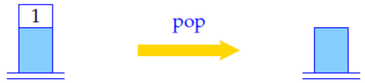
3 Statements and Statement Sequences

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--;

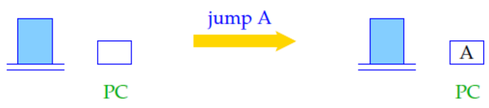
The code for a statement sequence is the concatenation of the code for the statements of the sequence:

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

28

4 Conditional and Iterative Statements

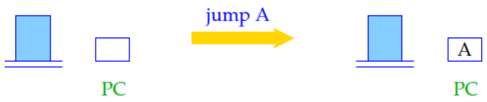
We need jumps to deviate from the serial execution of consecutive statements:



PC = A;

4 Conditional and Iterative Statements

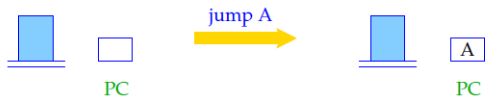
We need jumps to deviate from the serial execution of consecutive statements:



PC = A;

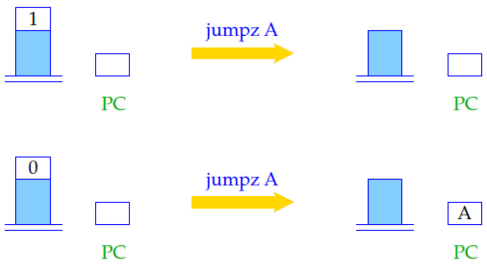
4 Conditional and Iterative Statements

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



PC = A;

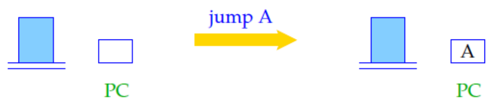
29



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

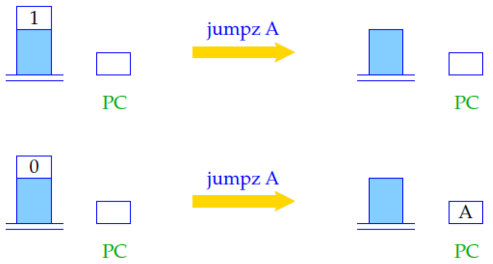
4 Conditional and Iterative Statements

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



PC = A;

29




```
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.

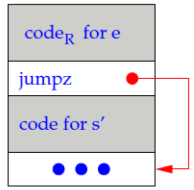
4.1 One-sided Conditional Statement

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

Idea:

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

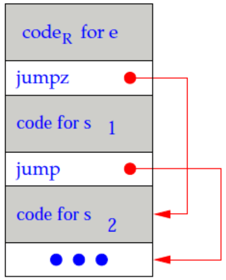
$$
\begin{array}{rcl}
\text{code}\ s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code}\ s'\ \rho \\
\text{A}: & & \ldots
\end{array}
$$



4.2 Two-sided Conditional Statement

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

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



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

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

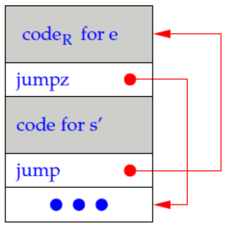
code s ρ produces:

loada 4	loada 4	A:	loada 7
loada 7	loada 7		loada 4
gr	sub		sub
jumpz A	storea 4		storea 7
	pop		pop
	jump B	B:	...
(i)	(ii)		(iii)

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 :	...



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

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

code s ρ produces:

loada 4	loada 4	A:	loada 7
loada 7	loada 7		loada 4
gr	sub		sub
jumpz A	storea 4		storea 7
	pop		pop
	jump B	B:	...
(i)	(ii)		(iii)

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

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

code $s\ \rho$ produces:

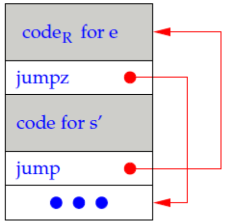
loada 4	loada 4	A:	loada 7
loada 7	loada 7		loada 4
gr	sub		sub
jumpz A	storea 4		storea 7
	pop		pop
	jump B	B:	...
(i)	(ii)		(iii)

4.3 while-Loops

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

code $s\ \rho$ =

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



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

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

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

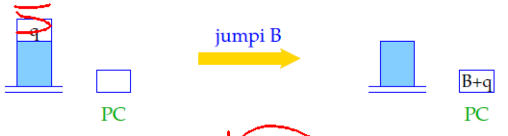
The for-loop $s \equiv \textbf{for}\ (e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;\ \textbf{while}\ (e_2)\ \{s'\ e_3;\}$ – provided that s' contains no **continue**-statement.
We therefore translate:

code $s\ \rho$ = $\text{code}_R\ e_1\ \rho$
pop
A : $\text{code}_R\ e_2\ \rho$
jumpz B
code $s'\ \rho$
$\text{code}_R\ e_3\ \rho$
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−−;