Script generated by TTT

Textbook → Library

Slides → Ralf Vogler

Title: Seidl: Virtual Machines (08.04.2014)

Date: Tue Apr 08 10:15:33 CEST 2014

Duration: 91:32 min

Pages: 32

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 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.: $x + 1$).

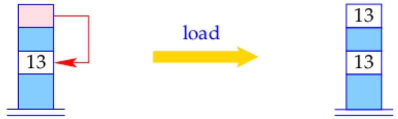
22

We define:

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

$$
\begin{array}{lcl}
\text{code}_R\ x\ \rho & = & \text{code}_L\ x\ \rho \\
 & & \text{load}
\end{array}
$$

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

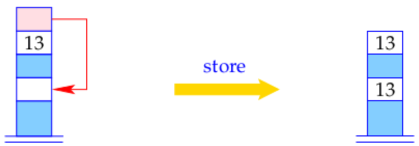


S[SP] = S[S[SP]];

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

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 ??

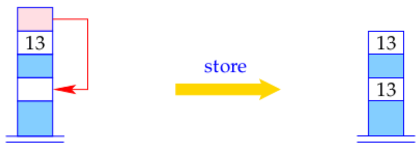


S[S[SP]] = S[SP-1];
SP--;

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

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--;

25

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

loadc 7	loadc 1	loadc 4
load	sub	store

Improvements:

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

$$
\begin{array}{lll}
\text{loada q} & = & \text{loadc q} \\
 & & \text{load} \\
\text{storea q} & = & \text{loadc q} \\
 & & \text{store}
\end{array}
$$

x = 1 ;

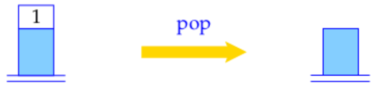
3 Statements and Statement Sequences

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.

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

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:

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

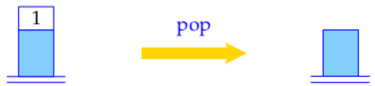
3 Statements and Statement Sequences

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:

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

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

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

x = y-1; s

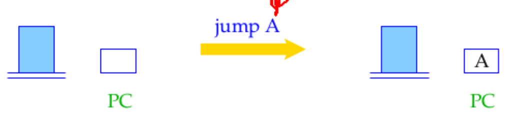
{int z; z = 7; y = z;}; s

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

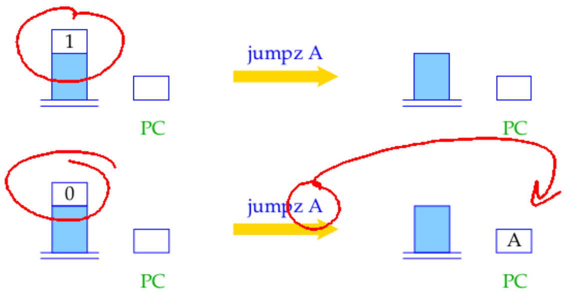
$$\begin{array}{lcl} \text{code}\ (s\, ss)\ \rho & = & \text{code}\ s\ \rho \\ & & \text{code}\ ss\ \rho \\ \text{code}\ \varepsilon\ \rho & = & \qquad //\ \textit{empty sequence of instructions} \end{array}$$

4 Conditional and Iterative Statements

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

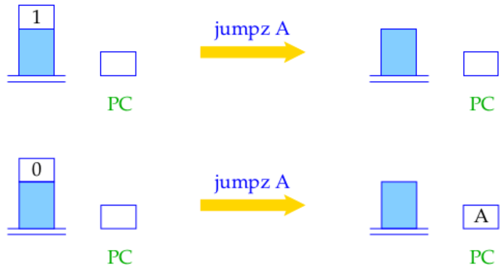


PC = A;



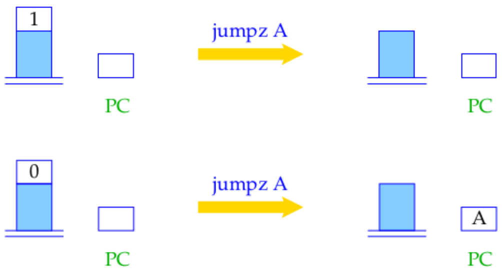
if (S[SP] == 0) PC = A;

SP--;



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

SP--;



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.

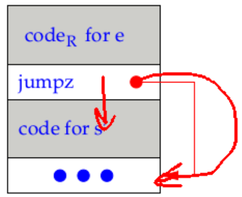
4.1 One-sided Conditional Statement

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

Idea:

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

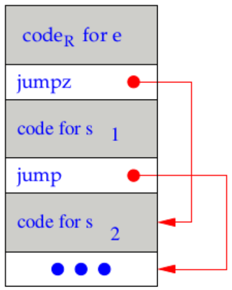
code $s\ \rho$	$=$	$\text{code}_R\ e\ \rho$
		jumpz A
		code $s'\ \rho$
A :		...



4.2 Two-sided Conditional Statement

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

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



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

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

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 $s \equiv$ **while** $(e)\ s'$. We generate:

code $s\ \rho$ =

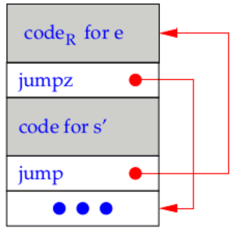
A : code$_R$ $e\ \rho$

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:

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 $s \equiv$ **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 $s\ \rho$ = code$_R$ $e_1\ \rho$

pop

A : code$_R$ $e_2\ \rho$

jumpz B

code $s'\ \rho$

code$_R$ $e_3\ \rho$

pop

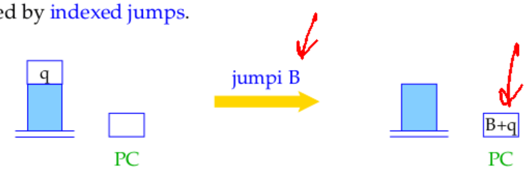
jump A

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--;

Simplification:

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

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

s is then translated into the instruction sequence:

Simplification:

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

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

s is then translated into the instruction sequence:

Simplification:

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

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

s is then translated into the instruction sequence:

$$
\begin{array}{lllllll}
\text{code}\ s\ \rho & = & \text{code}_R\ e\ \rho & C_0\text{:} & \text{code}\ ss_0\ \rho & B\text{:} & \text{jump}\ C_0 \\
 & & \text{check}\ 0\ k\ B & & \text{jump}\ D & & \ldots \\
 & & & & \ldots & & \text{jump}\ C_k \\
 & & & C_k\text{:} & \text{code}\ ss_k\ \rho & D\text{:} & \ldots \\
 & & & & \text{jump}\ D & &
\end{array}
$$

- 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.

$$
\begin{array}{lllllll}
\text{code}\; s\; \rho & = & \text{code}_{\text{R}}\; e\; \rho & \text{C}_0: & \text{code}\; ss_0\; \rho & \text{B}: & \text{jump C}_0 \\
 & & \text{check}\; 0\; k\; \text{B} & & \text{jump D} & & \ldots \\
 & & & & \ldots & & \text{jump C}_k \\
 & & & \text{C}_k: & \text{code}\; ss_k\; \rho & \text{D}: & \ldots \\
 & & & & \text{jump D} & &
\end{array}
$$

- 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.