Compiling Simple Assignments

Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

April 21, 1997

Subjects

- Compile- vs. Run-Time Information
- L-values vs. R-values
- The P-machine
- Code generation for expressions and assignments

Compile- vs. Run-Time Information

compile-time-information (static): Information contained in or derivable from the source program

- The source code
- The types of variables (in most languages)
- The scope of variables
- The values of constants
- The addresses of static variables
- The relative addresses of automatic variables
- The size of static data (scalars, stat. arrays, records)
- ⋮

run-time-information (dynamic): Information only available at run time

- The values of variables
- The values of conditions
- The depth of recursion
- The size of dynamic data (dyn. arrays, lists, trees)
- ⋮

L-values vs. R-values

- Assignment $x := exp$ is compiled into:
  1. Compute the **address** of x
  2. Compute the **value** of exp
  3. Store the value of exp at the address of x
- Generalization

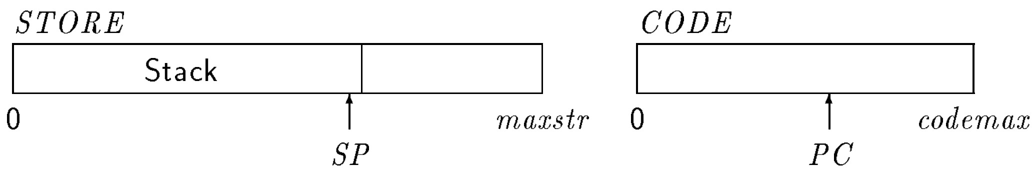
  R-value

$$
\begin{aligned}
\text{r-val}(x) &= \text{value of } x \\
\text{r-val}(5) &= 5 \\
\text{r-val}(x+y) &= \text{r-val}(x) + \text{r-val}(y)
\end{aligned}
$$

  L-value

$$
\begin{aligned}
\text{l-val}(x) &= \text{address of } x \\
\text{l-val}(5) &= \textbf{undefined} \\
\text{l-val}(x+y) &= \textbf{undefined} \\
\text{l-val}(a[i]) &= \text{l-val}(a) + \text{some function of } \text{r-val}(i)
\end{aligned}
$$

The P-Machine



- Memory

- Instructions operate on the stack

- Typed instructions (using Pascal ordinal types)

  $+_i$ 1. adds the two top most int values on the stack;
  2. removes these from the stack;
  3. stores (pushes) the result on the stack

- Operand types in the P-machine

i	integer
r	real
a	address
b	boolean

- Type indications in the instruction definitions

T	all types
N	numerical types

P-machine main loop

while true do begin
$PC := PC + 1$;
execute instruction in location $CODE[PC-1]$
end;

Why PC-increment before instruction execution?
Reason: jumps and procedure calls

Example Program

Pascal-program

```
program foo(input, output) ;
var x, y : integer;
begin
        x := 3 ;
        y := x + 7
end.
```

P-code-program

p		initialization code	
ssp	8		allocate stack space
sep	3		space for intermediate computations
ldc	a	5	pushes the address of x
ldc	i	3	pushes 3
sto	i		stores 3 in x
ldc	a	6	pushes the address of y
ldo	i	5	pushes the value of x
ldc	i	7	pushes 7
add	i		pushes x + 7
sto	i		stores x+7 in y
stp			

The definition of P-instructions

Instruction	Meaning	Condition	Result

Instruction: name of the instruction and list of its parameters,

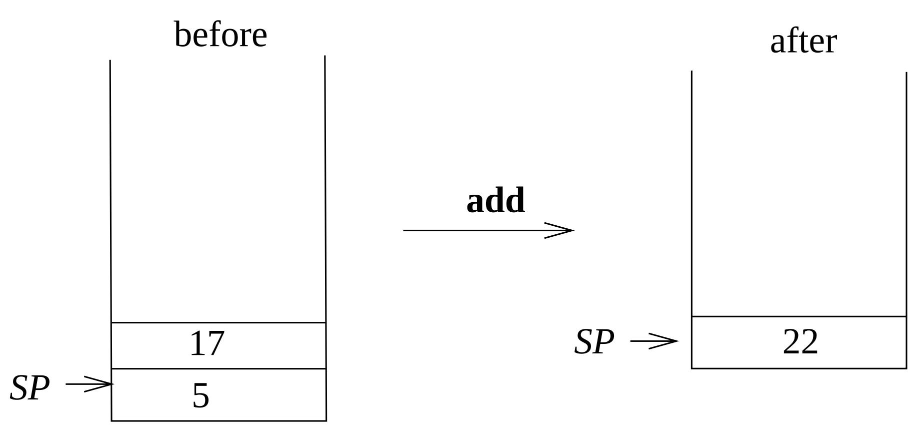
Meaning: program in a very reduced imperative language consisting of assignments between machine ressources and conditionals,

Condition: condition on the execution of the instruction, often a pattern describing the expected contents of the top end of the stack, i.e., the types of the cell contents,

Result: description of the result, a pattern describing the resulting stack contents.

P-instructions for Arithmetic

Instr.	Meaning	Cond.	Res.
add N	$STORE[SP-1] := STORE[SP-1] +_N STORE[SP]$; $SP := SP-1$	$\binom{N}{N}$	(N)
sub N	$STORE[SP-1] := STORE[SP-1] -_N STORE[SP]$; $SP := SP-1$	$\binom{N}{N}$	(N)
mul N	$STORE[SP-1] := STORE[SP-1] *_N STORE[SP]$; $SP := SP-1$	$\binom{N}{N}$	(N)
div N	$STORE[SP-1] := STORE[SP-1] /_N STORE[SP]$; $SP := SP-1$	$\binom{N}{N}$	(N)
neg N	$STORE[SP] := -_N STORE[SP]$	(N)	(N)



P-instructions for Boolean Operations

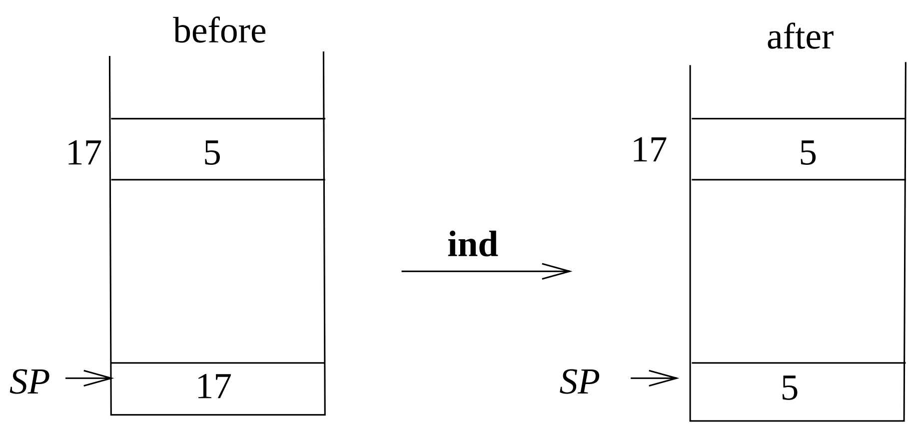
Instr.	Meaning	Cond.	Res.
and	$STORE[SP-1] := STORE[SP-1]\ and\ STORE[SP]$; $SP := SP-1$	$\binom{b}{b}$	(b)
or	$STORE[SP-1] := STORE[SP-1]\ or\ STORE[SP]$; $SP := SP-1$	$\binom{b}{b}$	(b)
not	$STORE[SP] := not\ STORE[SP]$	(b)	(b)

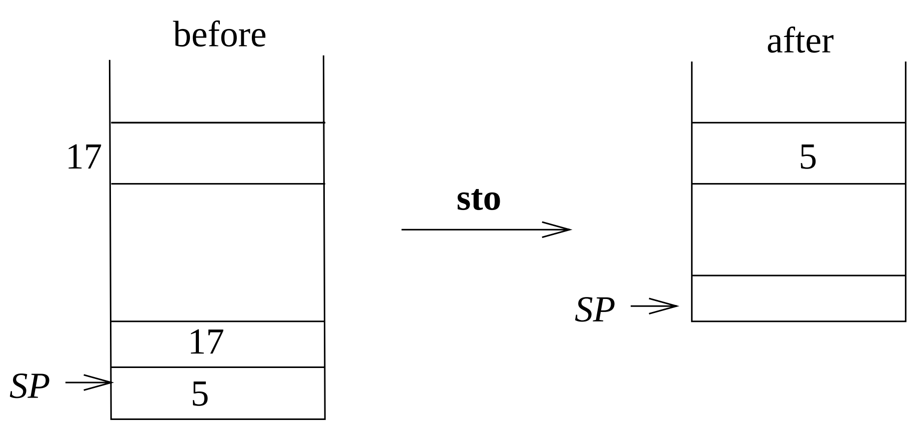
P-instructions for comparisons

Instr.	Meaning	Cond.	Res.
equ T	$STORE[SP-1] := STORE[SP-1] =_T STORE[SP]$; $SP := SP-1$	$\binom{T}{T}$	(b)
geq T	$STORE[SP-1] := STORE[SP-1] \geq_T STORE[SP]$; $SP := SP-1$	$\binom{T}{T}$	(b)
leq T	$STORE[SP-1] := STORE[SP-1] \leq_T STORE[SP]$; $SP := SP-1$	$\binom{T}{T}$	(b)
les T	$STORE[SP-1] := STORE[SP-1] <_T STORE[SP]$; $SP := SP-1$	$\binom{T}{T}$	(b)
grt T	$STORE[SP-1] := STORE[SP-1] >_T STORE[SP]$; $SP := SP-1$	$\binom{T}{T}$	(b)
neq T	$STORE[SP-1] := STORE[SP-1] \neq_T STORE[SP]$; $SP := SP-1$	$\binom{T}{T}$	(b)

P-instructions for load/store

Instr.	Meaning	Cond.	Res.
ldo $T\ q$	$SP := SP + 1;$ $STORE[SP] := STORE[\mathsf{q}]$	$q \in [0, maxstr]$	(T)
ldc $T\ q$	$SP := SP + 1;$ $STORE[SP] := q$	$Typ(q) = T$	(T)
ind T	$STORE[SP] := STORE[STORE[SP]]$	(a)	(T)
sro $T\ q$	$STORE[q] := STORE[SP];$ $SP := SP - 1$	(T) $q \in [0, maxstr]$	
sto T	$STORE[STORE[SP-1]] := STORE[SP];$ $SP := SP - 2$	$\begin{pmatrix} a \\ T \end{pmatrix}$	





Code Generation

- Assumptions about the input program:
  - No errors (syntax, types)
  - Structure and type information is available
  - No procedures (for now)

- $\rho(v)$ is the relative address of program variable v

- Invariant I_0 about the state of the P-machine:
  Let $code_R\ e\ \rho = is$,
  let sp be the value of SP before the execution of is.
  The execution of is will leave the value of e in $STORE[sp+1]$, and SP's value will be $sp+1$. $STORE$ is otherwise unchanged.

The translation of assignments and expressions

Function		Condition
$code_R(e_1 = e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **equ** T	$Typ(e_1) = Typ(e_2) = T$
$code_R(e_1 \neq e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **neq** T	$Typ(e_1) = Typ(e_2) = T$
$\vdots$		
$code_R(e_1 + e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **add** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 - e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **sub** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 * e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **mul** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 / e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **div** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(-e)\ \rho$	$= code_R\ e\ \rho;$ **neg** N	$Typ(e) = N$
$code_R\ x\ \rho$	$= code_L\ x\ \rho;$ **ind** T	x is a variable of type T
$code_R\ c\ \rho$	$=$ **ldc** T c	c is a constant of type T
$code(x := e)\ \rho$	$= code_L\ x\ \rho;\ code_R\ e\ \rho;$ **sto** T	$Typ(e) = T$,
		x is a variable
$code_L\ x\ \rho$	$=$ **ldc** a $\rho(x)$	x is a variable

Correctness of the code generation scheme

Proof by induction of the invariant I_0

Base cases:

$code_R\ x\ \rho$ and $code_R\ c\ \rho$

Inductive step:

Example: $code_R(e_1{+}e_2)\ \rho = code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **add** $=$ is

Let sp be the value of SP before the execution of is

Ind. Assumptions:

- The execution of $code_R\ e_1\ \rho$ leaves the value of e_1 in $STORE[SP+1]$ and SP has value $sp+1$.
- The execution of $code_R\ e_2\ \rho$ leaves the value of e_2 in $STORE[SP+2]$ and SP has value $sp+2$.

Step:

- **add** adds the topmost values and leaves the result, i.e. the value of e in $STORE[SP+1]$ and SP has value $sp+1$.

Example

Assume that $Typ(x) = int$ and $\rho(x) = 5$

$$
\begin{aligned}
&code(x := 3) \quad \rho \\
&\quad = code_L\ x\ \rho;\ code_R\ 3\ \rho;\ \textbf{sto}\ \mathsf{i} \\
&\quad = \textbf{ldc}\ \mathsf{a}\ 5;\ code_R(3)\ \rho;\ \textbf{sto}\ \mathsf{i} \\
&\quad = \textbf{ldc}\ \mathsf{a}\ 5;\ \textbf{ldc}\ \mathsf{i}\ 3;\ \textbf{sto}\ \mathsf{i}
\end{aligned}
$$

Example 2.1

Assume that $\rho(a) = 5$, $\rho(b) = 6$, and $\rho(c) = 7$ and that

$$Typ(a) = Typ(b) = Typ(c) = Typ(b{*}c) = Typ(b{+}b{*}c) = int$$

$code(a := (b + (b * c)))\ \ \rho$
$= \ code_L\ a\ \rho$; $code_R\ (b + (b * c))\ \rho$; **sto** i
$=$ **ldc** a 5; $code_R(b + (b * c))\ \rho$; **sto** i
$=$ **ldc** a 5; $code_R(b)\ \rho$; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; $code_L(b)\ \rho$; **ind** i; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b)\ \rho$; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_L(b)\ \rho$; **ind** i; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_L(c)\ \rho$; **ind** i; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; **ldc** a 7; **ind** i; **mul** i; **add** i; **sto** i

Interpretation of the generated code

q

```
ssp 9
sep 4
ldc a 5
ldc a 6
ind i
ldc a 6
ind i
ldc a 7
ind i
mul i
add i
sto i
```

(Non-)Optimality of the generated code

- Example 1: $a := b$
  - Generated code:
    ldc a $\rho(a)$; **ldc** a $\rho(b)$; **ind** i; **sto**
  - Minimal code
    ldo i $\rho(b)$; **sro** i $\rho(a)$
- Example 2: $a := a * a;$
  - Generated code:
    ldc a $\rho(a)$
    ldc a $\rho(a)$
    ind i
    ldc a $\rho(a)$
    ind i
    mul i
    sto
  - Minimal code
    ldo i $\rho(a)$
    dpl i
    mul i
    sro i $\rho(a)$

Summary

- An inductive definition of the generated code
- Assuming that the relative addresses of variables are known at compile-time
- The stack machine simplifies the task of handling complex expressions