# Memory Allocation

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

April 22, 1997

# Subjects

- Fixed size data
- Static aays
- Dynamic" Arrays
- Records
- Pointers

# Compile-Time Information on Variables

- Name
- Type
- Dimension of arrays
- Selectors in records
- Scope — when is it recognized
- Size — How many bytes are required at run-time to represent the object

# Assumptions

- Code generation for a single procedure
- The first $5$ stack locations are reserved; they will later be needed for procedure organization.

# The function $\rho$

- For every data type $t$
  $size(t)$ is the number of bytes required at run-time to represent objects of (static) type $t$

- Assumption:
  $size(integer) = size(real) = size(char) = size(boolean) = 1$

- Memory allocation for variables in order of appearance
  **var** $v_0 : t_0; v_1 : t_2; \ldots; v_k : t_k;$

$$\rho(v_i) = 5 + \sum_{j=0}^{i-1} size(t_j)$$

- Example: **var** $x, y : \mathbf{real}; z : \mathbf{boolean}$

$$\rho = [x \mapsto 5, y \mapsto 6, z \mapsto 7]$$

# Static Arrays

- The function $size$ is defined inductively
- Example:
  **var** $a$ : **array** $[-1..5, 6..7, 3..8]$ **of integer**
  - consists of $5 - (-1) + 1 = 7$ subarrays of type **array** $[6..7, 3..8]$ **of integer**
  - which consist of $7 - 6 + 1 = 2$ subarrays of type **array** $[3..8]$ **of integer**
  - whose sizes are $(8 - 3 + 1) \times size(int) = 6$.

  Thus $size(a) = 7 * 2 * 6$ words.

- For an array type

$$at : \mathbf{array}[l_1..u_1, l_2..u_2, \ldots, l_k..u_k] \ \mathbf{of}\ t$$

$size(at) = (u_1 - l_1 + 1) * size(\mathbf{array}[l_2..u_2, \ldots, l_k..u_k] \mathbf{of} t)$

Therefore

$$size(at) = \prod_{j=1}^{k} d_j * size(t) \text{ where}$$
$$d_j = u_j - l_j + 1$$

# Row Major Array Ordering

**var** $a$: **array**$[-5..5, 1..9]$ **of integer**

$$
\begin{array}{llll}
a[-5,1], & a[-5,2], & \ldots & , a[-5,9], \\
a[-4,1], & a[-4,2], & \ldots & , a[-4,9], \\
\vdots & & & \\
a[5,1], & a[5,2], & \ldots & , a[5,9],
\end{array}
$$

$$
\begin{array}{lcl}
\textsf{l-val}(a[i,j]) & = & \textsf{l-val}(a) \\
 & + & \text{pos. in 1st dimen.} * \text{size of 1-dim. subarray} \\
 & + & \text{pos. in 2nd dimen.} * \text{size of int} \\
 & = & \textsf{l-val}(a) \\
 & + & (i-(-5)) * (9-1+1) \\
 & + & j-1 \\
 & = & \textsf{l-val}(a) \\
 & + & (i+5) * 9 + j - 1 \\
 & = & \textsf{l-val}(a) \\
 & + & 9 * i + j + 44
\end{array}
$$

# Row Major General Array Ordering

**var** $a$: **array**$[l_1..u_1, l_2..u_2, \ldots, l_n..u_n]$ **of** $t$

$$
\begin{array}{lll}
a[l_1, \ldots, l_n], & \ldots & , a[l_1, \ldots, l_{n-1}, u_n], \\
\vdots & & \\
a[u_1, \ldots, u_{n-1}, l_n], & \ldots & , a[u_1, \ldots, u_{n-1}, u_n]
\end{array}
$$

Indexing the array:

l-val$(a[\hat{i}_1, \ldots, \hat{i}_n])$

$$
\begin{array}{rl}
= & \text{l-val}(a)+ \\
& (\hat{i}_1 - l_1) * \mathit{size}(\mathbf{array}[l_2..u_2, \ldots, l_n..u_n]\ \mathbf{of}\ t)+ \\
& (\hat{i}_2 - l_2) * \mathit{size}(\mathbf{array}[l_3..u_3, \ldots, l_n..u_n]\ \mathbf{of}\ t) + \cdots + \\
& (\hat{i}_n - l_n) * \mathit{size}(t) \\
= & (\hat{i}_1 - l_1) * (\prod_{j=2}^{n} d_j) * \mathit{size}(t)+ \\
& (\hat{i}_2 - l_2) * (\prod_{j=3}^{n} d_j) * \mathit{size}(t) + \cdots + \\
& (\hat{i}_n - l_n) * \mathit{size}(t) \\
= & \hat{i}_1 * (\prod_{j=2}^{n} d_j) * \mathit{size}(t)+ \\
& \hat{i}_2 * (\prod_{j=3}^{n} d_j) * \mathit{size}(t) + \cdots + \\
& \hat{i}_n * \mathit{size}(t)- \\
& (l_1 * \prod_{j=2}^{n} d_j + l_2 * \prod_{j=3}^{n} d_j + l_n) * \mathit{size}(t) \\
= & \hat{i}_1 * g * d^{(1)} + \hat{i}_2 * g * d^{(2)} + \cdots + \hat{i}_n * g - d * g
\end{array}
$$

where $\hat{i}_j =$ r-val$(i_j)$, $\quad d^{(i)} = \prod_{j=i+1}^{n}, \quad g = \mathit{size}(t)$.

# P-Code for Array Indexing

| Instr. | Meaning | Cond. | Result |
|---|---|---|---|
| **ixa** $q$ | $STORE[SP-1] :=$ $STORE[SP-1] +$ $STORE[SP] * q;$ $SP := SP - 1$ | $\binom{i}{a}$ | $(a)$ |
| **inc** $T q$ | $STORE[SP] :=$ $STORE[SP] + q$ | $(T),\ Typ(q) = i$ | $(T)$ |
| **dec** $T q$ | $STORE[SP] :=$ $STORE[SP] - q$ | $(T),\ Typ(q) = i$ | $(T)$ |

$$
\begin{array}{lcl}
code_L\ b[i_1,\ldots,i_n]\ g\ \rho & = & \textbf{ldc}\ \mathrm{a}\ \rho(b);\ code_I\ [i_1,\ldots,i_n]\ g\ \rho \\
code_I\ [i_1,\ldots,i_n]\ g\ \rho & = & code_R\ i_1\ \rho;\ \textbf{ixa}\ g * d^{(1)}; \\
 & & code_R\ i_2\ \rho;\ \textbf{ixa}\ g * d^{(2)}; \\
 & & \ \vdots \\
 & & code_R\ i_n\ \rho;\ \textbf{ixa}\ g; \\
 & & \textbf{dec}\ \mathrm{a}\ g * d;
\end{array}
$$

# Approaches to Array Bound Checking

- No checking at run-time (C)
- Generate a run-time check that the overall index expression is in range (PL1)
- Generate a run-time check that every array index is within range (Pascal, Java)

# P-code for Array Checking

| Instr. | Meaning | Cond. | Result |
|---|---|---|---|
| **chk** $p$ $q$ | **if not** $(p \leq STORE[SP] \leq q)$ **then** $error($*"value out of range"*$)$ **fi** | $(i)$ | $(i)$ |

New code for indexing including array bound checks

$$
\begin{array}{l}
code_I\ [i_1, \ldots, i_n]\ desc\ \rho = \\
\quad code_R\ i_1\ \rho; \textbf{chk}\ l_1\ u_1; \textbf{ixa}\ g \cdot d^{(1)}; \\
\quad code_R\ i_2\ \rho; \textbf{chk}\ l_2\ u_2; \textbf{ixa}\ g \cdot d^{(2)}; \\
\quad\quad \vdots \\
\quad code_R\ i_n\ \rho; \textbf{chk}\ l_n\ u_n; \textbf{ixa}\ g; \\
\quad \textbf{dec}\ \mathrm{a}\ g \cdot d;
\end{array}
$$

where $desc = (g; l_1, u_1, \ldots, l_n, u_n)$

This array description is made available through the symbol table, c.f. Semantic Analysis.

# Example

$$\begin{array}{rl} \textbf{var } i, j\text{:} & \textbf{integer};\\ a\text{:} & \textbf{array}[-5..5, 1..9] \textbf{ of integer} \end{array}$$

$code(a[i+1, j] := 0)\ \rho =$

**ldc** a $7$
**ldc** a $5$
**ind** i
**ldc** i $1$
**add** i
**chk** $-5$ $5$
**ixa** $9$
**ldc** a $6$
**ind** i
**chk** $1$ $9$
**ixa** $1$
**dec** a $-44$
**ldc** i $0$
**sto** i

# Dynamic Arrays

- The size of the array is determined when the procedure is entered
- Compile-time information:
  - Dimension, i.e. the number of indices
  - Size of the components (if they are not and don't contain dynamic arrays)
- Run-time information:
  - Array bounds
  - Index ranges
  - Size of the array
  - Relative address where the array begins
  - Value of index expressions

# Row Major General Array Ordering

**var** $a$: **array**$[l_1..u_1, l_2..u_2, \ldots, l_n..u_n]$ **of** $t$

$$\begin{array}{lll} a[l_1,\ldots,l_n], & \ldots & ,a[l_1,\ldots,l_{n-1},u_n], \\ \vdots & & \\ a[u_1,\ldots,u_{n-1},l_n], & \ldots & ,a[u_1,\ldots,u_{n-1},u_n]. \end{array}$$

Indexing a dynamic array:

l-val$(a[\hat{i}_1, \ldots, \hat{i}_n])$

$$\begin{array}{ll} = & \text{l-val}(a) + \\ & (\hat{i}_1 - l_1) * (\prod_{j=2}^{n} d_j) * size(t) + \\ & (\hat{i}_2 - l_2) * (\prod_{j=3}^{n} d_j) * size(t) + \cdots + \\ & (\hat{i}_n - l_n) * size(t) \\ = & (\cdots((\hat{i}_1 * d_2 + \hat{i}_2) * d_3 + \hat{i}_3) * d_4 + \cdots) * d_n + \hat{i}_n) * size(t) - \\ & (\cdots((l_1 * d_2 + l_2) * d_3 + l_3) * d_4) + l_n) * size(t) \\ = & d_1 - d_2 \end{array}$$

where $\hat{i}_j$ is an abbreviation for $rval(i_j)$.
$d_j = u_j - l_j + 1$ and $d_2$ computed once, upon array creation.
$d_1$ computed for each array access.
Computing $d_1$ by a Horner scheme saves multiplications.
Subtracting $d_2$ from the array address yields the "adjusted address".

# Array Descriptor

| | |
|---|---|
| 0 | Adjusted Address :a |
| 1 | Array size :i |
| 2 | Subtr. part :i |
| 3 | $l_1$ :i |
| 4 | $u_1$ :i |
| $\vdots$ | $\vdots$ |
| $2n+1$ | $l_n$ :i |
| $2n+2$ | $u_n$ :i |
| $2n+3$ | $d_2$ :i |
| $\vdots$ | $\vdots$ |
| $3n+1$ | $d_n$ :i |

$code_{Ld}\ c[i_1, \ldots, i_k]\ \rho =$ **ldc** a $\rho(c)$; $code_{Id}\ [i_1, \ldots, i_k]\ g\ \rho$

$code_{Id}\ [i_1, \ldots, i_n]\ g\ \rho\ =$

**dpl** i; descriptor address
**ind** i; adjusted address
**ldc** i 0;
$code_R\ i_1\ \rho$; **add** i; **ldd** $2n+3$; **mul** i;
$code_R\ i_2\ \rho$; **add** i; **ldd** $2n+4$; **mul** i;
...
$code_R\ i_{n-1}\ \rho$; **add** i; **ldd** $3n+1$; **mul** i;
$code_R\ i_n\ \rho$; **add** i;
**ixa** $g$;
**sli** a

# Additional P-code instructions

| Instr. | Meaning | Cond. | Result |
|---|---|---|---|
| **dpl** $T$ | $SP := SP + 1;$ $STORE[SP] := STORE[SP - 1]$ | $T$ | $\begin{pmatrix} T \\ T \end{pmatrix}$ |
| **ldd** $q$ | $SP := SP + 1;$ $STORE[SP] := STORE[STORE[SP - 3] + q]$ | $\begin{pmatrix} T_2 \\ T_1 \\ a \end{pmatrix}$ | $\begin{pmatrix} i \\ T_2 \\ T_1 \\ a \end{pmatrix}$ |
| **sli** $T_2$ | $STORE[SP - 1] := STORE[SP];$ $SP := SP - 1$ | $\begin{pmatrix} T_2 \\ T_1 \end{pmatrix}$ | $(T_1)$ |

# A Simplified Example

**program** test;
**var** i : **integer** ; /* $\rho(i) = 5$ */
**procedure** p ;
**var** j, a[1..5, 1..i] **of integer** ; /* $\rho(j) = 5, \rho(a) = 6$ */
**begin** a[3, j] := $\cdots$ /*

```
ldc a 6
dpl i
ind i
ldc i 0
ldc i 3
add i
ldd 2 * 2 + 3
mul i
ldc a 5
ind i
add i
ixa 1
sli a
. . .
sto i */
```

**end**;
**begin read**(i); p **end.**

# Records

- Size known at compile time
- We assume “unique names”
- Example:

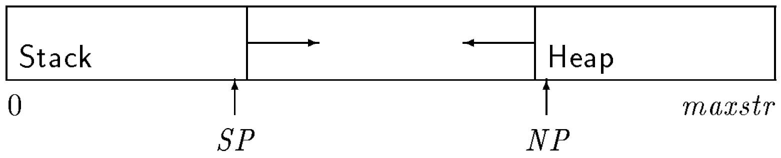
**var** v : **record** /* $\rho(v) = 5$*/
a : **integer** ; /* $\rho(a) = 0$ */
b : **integer** /* $\rho(b) = 1$ */
**end**

$code_L$(v.b) = **ldc** a 5 ; **inc** a 1

# Pointers and dynamic memory allocation

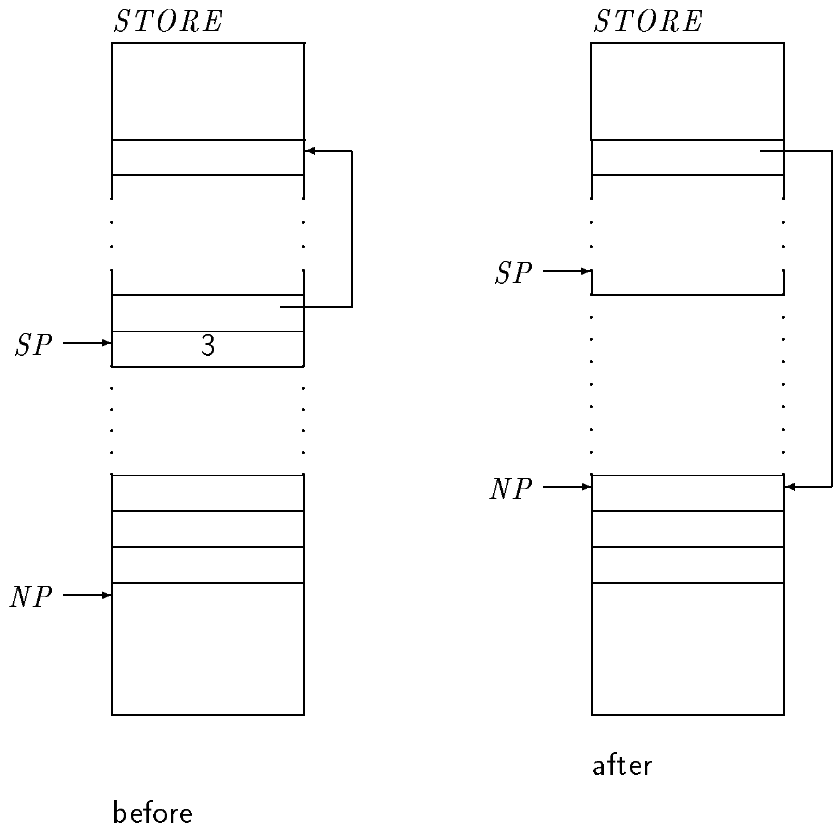
- Dynamic duration (lifetime) of the memory
- Storage can be freed in one of the following cases:
  - Can no longer be refererenced ("garbage collection")
  - Explicit deallocation (**cfree**, **dispose**)
- Space cannot be allocated on the "stack"

# The Heap of the P-Machine




| Instruct. | Meaning | Cond. | Result |
|---|---|---|---|
| **new** | **if** $NP - STORE[SP] \leq EP$<br>**then** *error* (“store overflow”)<br>**else**<br>$NP := NP - STORE[SP]$;<br>$STORE[STORE[SP-1]] := NP$;<br>$SP := SP-2$<br>**fi**; | $\begin{pmatrix} i \\ a \end{pmatrix}$ | |

# The Effect of New Statement



# The Generated P-Code

Assume that $Typ(x) = \uparrow t$

$code(\textbf{new}(x))\ \rho\ =$

**ldc** a $\rho(x)$
**ldc** i $size(t)$
**new**

# Complex Pointer Expressions

- Examples:
  - $x \uparrow [i+1, j].a \uparrow [i] \uparrow$
  - $y.a.b.c \uparrow [i, j+1].d$
- Recursive translation

$$
\begin{array}{lcl}
code_L(xr)\ \rho & = & \textbf{ldc}\ \mathrm{a}\ \rho(x); \text{for name } x \\
 & & code_M(r)\ \rho \\
code_M(.xr)\ \rho & = & \textbf{inc}\ \mathrm{a}\ \rho(x); \text{for name } x \\
 & & code_M(r)\ \rho \\
code_M(\uparrow r)\ \rho & = & \textbf{ind}\ \mathrm{a}; \\
 & & code_M(r)\ \rho \\
code_M([i_1, \ldots, i_n]r)\ \rho & = & code_{Id}\ [i_1, \ldots, i_n]\ g\ \rho; \\
 & & code_M(r)\ \rho \qquad \text{for array} \\
code_M(\varepsilon)\ \rho & = & \varepsilon
\end{array}
$$

# Example

| | | |
|---|---|---|
| **type** | $t =$ | **record** |
| | | $a$ : **array**$[-5.. + 5, 1..9]$ **of integer**; |
| | | $b : \uparrow t$ |
| | | **end**; |
| **var** | $i, j$ | : **integer**; |
| | $pt$ | $:\uparrow t$; |

$code_L\ pt \uparrow .b \uparrow .a[i + 1, j]$

| | |
|---|---|
| **ldc** a $\rho(pt)$; | Load Address of $pt$ |
| **ind** a; | Load start address of the record |
| **inc** a 99; | The start address of b |
| **ind** a; | Dereference pointer |
| **inc** a 0; | The start address of a |
| $code_{Id}[i + 1, j]\ 1\ \rho$ | |