Script **generated by TTT**

Title: Simon: Compilerbau (07.07.2014)

Date: Mon Jul 07 14:17:59 CEST 2014

Duration: 84:04 min

Pages: 80

Chapter 1:
Data Structures in Memory

Variables in Memory: L-Value and R-Value

Variables can be used in two different ways.

example: `a[x] = y + 1`

for `y` we need to know the value of the memory cell, for `a[x]` we are interested in the address

r-value of x	=	content of x
l-value of x	=	address of x

compute r- and l-value in register R_i:

$\mathrm{code}_\mathrm{R}^i\ e\ \rho$	generates code to compute the r-value of e, given the environment ρ
$\mathrm{code}_\mathrm{L}^i\ e\ \rho$	analogously for the l-value

note:
Not every expression has an l-value (e.g.: $x + 1$).

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: ~~a~~ variable ~~is~~ stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

$x = x$

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: a variable is stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: a variable is stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

accordingly, we define $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ as follows:

- $\rho\, x = \langle G, a \rangle$: variable x is stored at absolute address a
- $\rho\, x = \langle L, a \rangle$: variable x is stored at address $FP + a$
- $\rho\, x = \langle R, a \rangle$: variable x is stored in register R_a

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: a variable is stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

accordingly, we define $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ as follows:

- $\rho\, x = \langle G, a \rangle$: variable x is stored at absolute address a
- $\rho\, x = \langle L, a \rangle$: variable x is stored at address $FP + a$
- $\rho\, x = \langle R, a \rangle$: variable x is stored in register R_a

Observe: a variable x can only have one entry in ρ
However:

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: a variable is stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

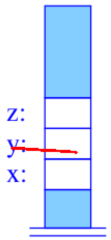
accordingly, we define $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ as follows:

- $\rho\, x = \langle G, a \rangle$: variable x is stored at absolute address a
- $\rho\, x = \langle L, a \rangle$: variable x is stored at address $FP + a$
- $\rho\, x = \langle R, a \rangle$: variable x is stored in register R_a

Observe: a variable x can only have one entry in ρ
However:

- ρ may be change with the program point
- that is, x may be assigned to a register at one point
- *and* to a memory location at another program point

Necessity of Storing Variables in Memory

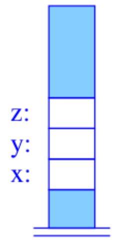


Global variables:

- could be assigned throughout to registers $R_1 \dots R_n$

Furthermore:

Necessity of Storing Variables in Memory

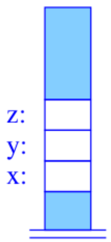


Global variables:

- could be assigned throughout to registers $R_1 \dots R_n$

Furthermore:

Necessity of Storing Variables in Memory

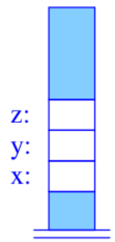


Global variables:

- could be assigned throughout to registers $R_1 \dots R_n$
- separate compilation becomes difficult, since code of function depends on n

Furthermore:

Necessity of Storing Variables in Memory

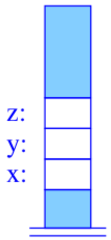


Global variables:

- could be assigned throughout to registers $R_1 \dots R_n$
- separate compilation becomes difficult, since code of function depends on n
- simple solution: store global variables in memory

Furthermore:

Necessity of Storing Variables in Memory



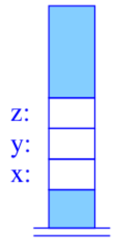
Global variables:

- could be assigned throughout to registers $R_1 \ldots R_n$
- separate compilation becomes difficult, since code of function depends on n
- simple solution: store global variables in memory

Furthermore:

- a variable x (**int** or **struct**) whose address has been taken must be stored in memory, i.e. $\rho\, x = \langle L, o\rangle$ or $\rho\, x = \langle G, o\rangle$

Necessity of Storing Variables in Memory



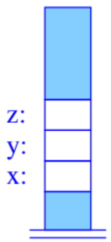
Global variables:

- could be assigned throughout to registers $R_1 \ldots R_n$
- separate compilation becomes difficult, since code of function depends on n
- simple solution: store global variables in memory

Furthermore:

- a variable x (**int** or **struct**) whose address has been taken must be stored in memory, i.e. $\rho\, x = \langle L, o\rangle$ or $\rho\, x = \langle G, o\rangle$
- an access to an array is always done through a pointer, hence, it must be stored in memory

Necessity of Storing Variables in Memory



Global variables:

- could be assigned throughout to registers $R_1 \ldots R_n$
- separate compilation becomes difficult, since code of function depends on n
- simple solution: store global variables in memory

Furthermore:

- a variable x (**int** or **struct**) whose address has been taken must be stored in memory, i.e. $\rho\, x = \langle L, o\rangle$ or $\rho\, x = \langle G, o\rangle$
- an access to an array is always done through a pointer, hence, it must be stored in memory
- optimization: store individual elements of a **struct** in register while no pointer accesses may reach the structure

Translation of Statements

Statements such as `x=2*y` have so far been translated by:

- computing the r-value of `2*y` in register R_i,
- copying the content of R_i into the register $\rho(x)$

formally: let $\rho(x) = \langle R, j\rangle$ then:

$$\begin{array}{rcl} \text{code}^i_{\text{R}}\; x = e_2\; \rho & = & \text{code}^i_{\text{R}}\; e_2\; \rho \\ & & \text{move}\; R_j\; R_i \end{array}$$

Translation of Statements

Statements such as `x=2*y` have so far been translated by:

- computing the r-value of `2*y` in register R_i,
- copying the content of R_i into the register $\rho(x)$

formally: let $\rho(x) = \langle R, j\rangle$ then:

$$\begin{array}{lll} \text{code}_R^i\ x = e_2\ \rho & = & \text{code}_R^i\ e_2\ \rho \\ & & \text{move}\ R_j\ R_i \end{array}$$

but: undefined result if $\rho\ x = \langle L, a\rangle$ or $\rho\ x = \langle G, a\rangle$.

Translation of Statements

Statements such as `x=2*y` have so far been translated by:

- computing the r-value of `2*y` in register R_i,
- copying the content of R_i into the register $\rho(x)$

formally: let $\rho(x) = \langle R, j\rangle$ then:

$$\begin{array}{lll} \text{code}_R^i\ x = e_2\ \rho & = & \text{code}_R^i\ e_2\ \rho \\ & & \text{move}\ R_j\ R_i \end{array}$$

but: undefined result if $\rho\ x = \langle L, a\rangle$ or $\rho\ x = \langle G, a\rangle$.

$e_1 = e_2$

idea:

- compute the r-value of e_2 in register R_i,
- compute the l-value of e_1 in register R_{i+1} and
- write e_2 to address e_1 using a store instruction

Translation of L-Values

new instruction: store R_i R_j with semantics $S[R_i] = R_j$

definition for assignments:

$$\text{code}^i\ e\ \rho = \text{code}_R^i\ e\ \rho$$

So how do we translate $x = e$ (with $\rho\ x = \langle G, a\rangle$)?

Translation of L-Values

new instruction: store R_i R_j with semantics $S[R_i] = R_j$

definition for assignments:

$$\text{code}^i\ e\ \rho = \text{code}_R^i\ e\ \rho$$

So how do we translate $x = e$ (with $\rho\ x = \langle G, a\rangle$)?

- Thus, for the case $e_1 = x$ and $\rho\ x = \langle R, j\rangle$ does *not* hold:

$$\begin{array}{lll} \text{code}_R^i\ e_1 = e_2\ \rho & = & \text{code}_R^i\ e_2\ \rho \\ & & \text{code}_L^{i+1}\ e_1\ \rho \\ & & \text{store}\ R_{i+1}\ R_i \end{array}$$

Translation of L-Values

new instruction: store R_i R_j with semantics $S[R_i] = R_j$

definition for assignments:

$$\text{code}^i\ e\ \rho = \text{code}^i_R\ e\ \rho$$

So how do we translate $x = e$ (with $\rho\ x = \langle G, a\rangle$)?

- Thus, for the case $e_1 = x$ and $\rho\ x = \langle R, j\rangle$ does *not* hold:

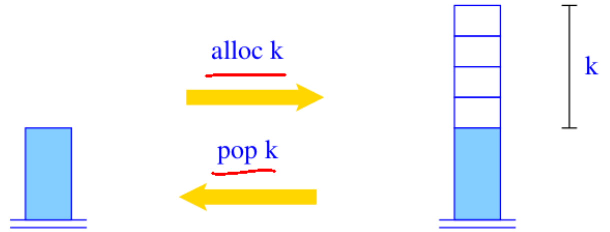
$$\begin{aligned}\text{code}^i_R\ e_1 = e_2\ \rho \quad = \quad & \text{code}^i_R\ e_2\ \rho \\ & \text{code}^{i+1}_L\ e_1\ \rho \\ & \text{store}\ R_{i+1}\ R_i\end{aligned}$$

- The l-value of a variable is computed as follows:

$$\text{code}^i_L\ x\ \rho \quad = \quad \text{loadc}\ R_i\ a$$

Allocating Memory for Local Variables

Given: a function with k local `int` variables that need to be stored in memory.



alloc k SP = SP + k;

pop k SP = SP - k;

The instruction alloc k reserves space for k variables on the stack, pop k frees this space again.

Access to Local Variables

Accesses to local variables are relative to FP. We therefore modify code_L to cater for variables in memory.

f() {

int x

For $\rho\ x = \langle L, a\rangle$ we define

$$\text{code}^i_L\ x\ \rho = \text{loadrc}\ R_i\ a \text{ if } \rho x = \langle L, a\rangle$$

}

Instruction loadrc R_i k computes the sum of FP and k.

$$R_i = FP + k$$

General Computation of the L-Value of a Variable

Computing the address of a variable in R_i is done as follows:

$$\text{code}^i_L\ x\ \rho = \begin{cases} \text{loadc}\ R_i\ a & \text{if } \rho x = \langle G, a\rangle \\ \text{loadrc}\ R_i\ a & \text{if } \rho x = \langle L, a\rangle \end{cases}$$

General Computation of the L-Value of a Variable

Computing the address of a variable in R_i is done as follows:

$$\text{code}_L^i\ x\ \rho = \begin{cases} \text{loadc } R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \text{loadrc } R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \end{cases}$$

Note: for $\rho\, x = \langle R, j\rangle$ the function code_L^i is not defined!

General Computation of the L-Value of a Variable

Computing the address of a variable in R_i is done as follows:

$$\text{code}_L^i\ x\ \rho = \begin{cases} \text{loadc } R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \text{loadrc } R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \end{cases}$$

Note: for $\rho\, x = \langle R, j\rangle$ the function code_L^i is not defined!

Observations:

General Computation of the L-Value of a Variable

Computing the address of a variable in R_i is done as follows:

$$\text{code}_L^i\ x\ \rho = \begin{cases} \text{loadc } R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \text{loadrc } R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \end{cases}$$

Note: for $\rho\, x = \langle R, j\rangle$ the function code_L^i is not defined!

Observations:

- intuitively: a register has no address
- during the compilation the l-value of a register may never be computed
- this requires a case distinction for assignments

Macro-Command for Accessing Local Variables

Define: the command $\text{load } R_i\ R_j$ sets R_i to the value at address R_j.

Macro-Command for Accessing Local Variables

Define: the command $\mathrm{load}\ R_i\ R_j$ sets R_i to the value at address R_j.

Thus: $\mathrm{loadrc}\ R_i\ a; \mathrm{load}\ R_j\ R_i$: sets R_j to x where $\rho\ x = \langle L, a \rangle$.

Macro-Command for Accessing Local Variables

Define: the command $\mathrm{load}\ R_i\ R_j$ sets R_i to the value at address R_j.

Macro-Command for Accessing Local Variables

Define: the command $\mathrm{load}\ R_i\ R_j$ sets R_i to the value at address R_j.

Thus: $\mathrm{loadrc}\ R_i\ a; \mathrm{load}\ R_j\ R_i$: sets R_j to x where $\rho\ x = \langle L, a \rangle$.

Macro-Command for Accessing Local Variables

Define: the command $\mathrm{load}\ R_i\ R_j$ sets R_i to the value at address R_j.

Thus: $\mathrm{loadrc}\ R_i\ a; \mathrm{load}\ R_j\ R_i$: sets R_j to x where $\rho\ x = \langle L, a \rangle$.

In general: Load variable x into register R_i:

$$\mathrm{code}_R^i\ x\ \rho = \begin{cases} \mathrm{loada}\ R_i\ a & \text{if } \rho\, x = \langle G, a \rangle \\ \mathrm{loadr}\ R_i\ a & \text{if } \rho\, x = \langle L, a \rangle \\ \mathrm{move}\ R_i\ R_j & \text{if } \rho\, x = \langle R, i \rangle \end{cases}$$

Macro-Command for Accessing Local Variables

Define: the command load $R_i\ R_j$ sets R_i to the value at address R_j.

Thus: loadrc $R_i\ a$; load $R_j\ R_i$: sets R_j to x where $\rho\ x = \langle L, a\rangle$.

In general: Load variable x into register R_i:

$$\mathrm{code}^i_{\mathrm{R}}\ x\ \rho = \begin{cases} \mathrm{loada}\ R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \mathrm{loadr}\ R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \\ \mathrm{move}\ R_i\ R_j & \text{if } \rho\, x = \langle R, i\rangle \end{cases}$$

Analogously: for write operations we define:

$$\begin{array}{lcl} \mathrm{storer}\ a\ R_j & \equiv & \mathrm{loadrc}\ R_i\ a \\ & & \mathrm{store}\ R_i\ R_j \\ \mathrm{storea}\ a\ R_j & \equiv & \mathrm{loadc}\ R_i\ a \\ & & \mathrm{store}\ R_i\ R_j \end{array}$$

loadr ≡ loadrc
load
loada ≡ loadc
load

i.e. storea $a\ R_j$ is a *macro*. Define special case (where $\rho\ x = \langle G, a\rangle$):

$$\begin{array}{lcl} \mathrm{code}^i_{\mathrm{R}}\ x = e_2\ \rho & = & \mathrm{code}^i_{\mathrm{R}}\ e_2\ \rho \\ & & \mathrm{code}^{i+1}_{\mathrm{L}}\ x\ \rho \\ & & \mathrm{store}\ R_{i+1}\ R_i \end{array}$$

Macro-Command for Accessing Local Variables

Define: the command load $R_i\ R_j$ sets R_i to the value at address R_j.

Thus: loadrc $R_i\ a$; load $R_j\ R_i$: sets R_j to x where $\rho\ x = \langle L, a\rangle$.

In general: Load variable x into register R_i:

$$\mathrm{code}^i_{\mathrm{R}}\ x\ \rho = \begin{cases} \mathrm{loada}\ R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \mathrm{loadr}\ R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \\ \mathrm{move}\ R_i\ R_j & \text{if } \rho\, x = \langle R, i\rangle \end{cases}$$

Analogously: for write operations we define:

$$\begin{array}{lcl} \mathrm{storer}\ a\ R_j & \equiv & \mathrm{loadrc}\ R_i\ a \\ & & \mathrm{store}\ R_i\ R_j \\ \mathrm{storea}\ a\ R_j & \equiv & \mathrm{loadc}\ R_i\ a \\ & & \mathrm{store}\ R_i\ R_j \end{array}$$

i.e. storea $a\ R_j$ is a *macro*. Define special case (where $\rho\ x = \langle G, a\rangle$):

$$\begin{array}{lcl} \mathrm{code}^i_{\mathrm{R}}\ x = e_2\ \rho & = & \mathrm{code}^i_{\mathrm{R}}\ e_2\ \rho \\ & & \mathrm{loadc}\ R_{i+1}\ a \\ & & \mathrm{store}\ R_{i+1}\ R_i \end{array}$$

Data Transfer Instructions of the R-CMa

read- and write accesses of the R-CMa are as follows:

instruction	semantics	intuition
load $R_i\ R_j$	$R_i \leftarrow S[R_j]$	load value from address
loada $R_i\ c$	$R_i \leftarrow S[c]$	load global variable
loadr $R_i\ c$	$R_i \leftarrow S[FP + c]$	load local variable
store $R_i\ R_j$	$S[R_i] \leftarrow R_j$	store value at address
storea $c\ R_i$	$S[c] \leftarrow R_i$	write global variable
storer $c\ R_i$	$S[FP + c] \leftarrow R_i$	write local variable

instructions for computing addresses:

instruction	semantics	intuition
loadc $R_i\ c$	$R_i \leftarrow c$	load constant
loadrc $R_i\ c$	$R_i \leftarrow FP + c$	load constant relative to FP

instructions for general data transfer:

instruction	semantics	intuition
move $R_i\ R_j$	$R_i \leftarrow R_j$	transfer value between regs
move $R_i\ k\ R_j$	$[S[SP+i+1] \leftarrow S[R_j+i]]_{i=0}^{k-1}$ $R_i \leftarrow SP+1; SP \leftarrow SP+k$	copy k values onto stack

Determining the Address-Environment

variables in the symbol table are tagged in one of three ways:

1. global variables, defined outside of functions (or as `static`); ⟨G, 7⟩
2. local (automatic) variables, defined inside functions, accessible by pointers; ⟨R, -5⟩
3. register (automatic) variables, defined inside functions.

Example: ~~*(&x+1) = 5~~

```
int x, y;
void f(int v, int w) {
  int a;
  if (a>0) {
    int b;
    g(&b);
  } else {
    int c;
  }
}
```

v	$\rho(v)$
x	⟨G, 42⟩
y	⟨G, 40⟩
v	⟨R, -1⟩
w	⟨R, -2⟩
a	⟨R, 1⟩
b	⟨R, 2⟩
c	⟨R, 2⟩

Determining the Address-Environment

variables in the symbol table are tagged in one of three ways:

1. global variables, defined outside of functions (or as `static`);
2. local (automatic) variables, defined inside functions, accessible by pointers;
3. register (automatic) variables, defined inside functions.

Example:

```
int x, y;
void f(int v, int w) {
  int a;
  if (a>0) {
    int b;
    g(&b);
  } else {
    int c;
  }
}
```

v	$\rho(v)$
x	⟨ G , 0 ⟩
y	⟨ G , 1 ⟩
v	⟨ R , -1 ⟩
w	⟨ R , -2 ⟩
a	⟨ R , 1 ⟩
b	⟨ L , 0 ⟩
c	⟨ R , 2 ⟩

Function Arguments on the Stack

- C allows for so-called *variadic functions*
- an unknown number of parameters: $R_{-1}, R_{-2}, \ldots$
- problem: callee cannot index into global registers

example:

```
int printf(const char * format, ...);
char *s =
  "Hello␣%s!\nIt's␣%i␣to␣%i!\n";

int main(void) {
  printf(s ,"World", 5, 12);
  return 0;
}
```

value	$\rho(p_i)$
s	$\langle R, -1\rangle$
"World"	$\langle L, -3\rangle$
5	$\langle L, -4\rangle$
12	$\langle L, -5\rangle$

idea:

- push *variadic* parameters from *right to left* onto the stack
- the first parameter lies right below PC, FP, EP
- for a prototype $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k,\ \ldots)$ we set:

$$x_1 \mapsto \langle R, -1\rangle \qquad x_k \mapsto \langle R, -k\rangle$$
$$x_{k+1} \text{ at } \langle L, -3\rangle \qquad x_{k+i} \text{ at } \langle L, -3 - |\tau_{k+1}| - \ldots - |\tau_{k+i-1}|\rangle$$

Chapter 2:
Arrays and Pointers

int float

Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let `t[c] a;` be the declaration of an array a.

Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let `t[c] a;` be the declaration of an array a.
In order to compute the address of `a[i]`, we need to compute $\rho\, a + |t| * $ *(R-Wert von i)*. Thus:

$$\begin{array}{lll} \text{code}_\text{L}^i\ e_2[e_1]\ \rho & = & \text{code}_\text{R}^i\ e_1\ \rho \\ & & \text{code}_\text{R}^{i+1}\ e_2\ \rho \\ & & \text{loadc}\ R_{i+2}\ |t| \\ & & \text{mul}\ R_{i+1}\ R_{i+1}\ R_{i+2} \\ & & \text{add}\ R_i\ R_i\ R_{i+1} \end{array}$$

Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let `t[c] a;` be the declaration of an array a.
In order to compute the address of `a[i]`, we need to compute $\rho\, a + |t| * $ *(R-Wert von i)*. Thus:

$$\begin{array}{lll} \text{code}_\text{L}^i\ e_2[e_1]\ \rho & = & \text{code}_\text{R}^i\ e_1\ \rho \\ & & \text{code}_\text{R}^{i+1}\ e_2\ \rho \\ & & \text{loadc}\ R_{i+2}\ |t| \\ & & \text{mul}\ R_{i+1}\ R_{i+1}\ R_{i+2} \\ & & \text{add}\ R_i\ R_i\ R_{i+1} \end{array}$$

Note:

- An array in C is simply a *pointer*. The declared array `a` is a *pointer constant*, whose r-value is address of the first field of `a`.
- Formally, we compute the r-value of a field `e` as $\text{code}_\text{R}^i\ e\ \rho = \text{code}_\text{L}^i\ e\ \rho$
- in C the following are equivalent (as l-value, not as types):

  `2[a]`   `a[2]`   `a+2`

C `structs` (Records)

Note:
The same field name may occur in different `structs`
Here: The component environment ρ_{st} relates to the currently translated structure st.
Let **struct** `{` **int** `a;` **int** `b; } x;` be part of a declaration list.

- `x` is a variable of the size of (at least) the sum of the sizes of its fields
- we populate ρ_{st} with addresses of fields that are *relative* to the beginning of `x`, here $a \mapsto 0$, $b \mapsto 1$.

C `structs` (Records)

Note:
The same field name may occur in different `structs`
Here: The component environment ρ_{st} relates to the currently translated structure st.
Let **struct** `{` **int** `a;` **int** `b; } x;` be part of a declaration list.

- `x` is a variable of the size of (at least) the sum of the sizes of its fields
- we populate ρ_{st} with addresses of fields that are *relative* to the beginning of `x`, here $a \mapsto 0$, $b \mapsto 1$.

In general, let $t \equiv$ **struct** `{` $t_1\ v_1$`;` ... `;` $t_k\ v_k$ `}`, then

$$|t| := \sum_{i=1}^{k} |t_i| \qquad \rho_{st}\ v_1 := 0 \qquad \rho_{st}\ v_i := \rho_{st}\ v_{i-1} + |t_{i-1}| \quad \text{für } i > 1$$

We obtain:

e.c = _

$$\begin{array}{lll} \text{code}_\text{L}^i\ (e.c)\ \rho & = & \text{code}_\text{L}^i\ e\ \rho \\ & & \text{loadc}\ R_{i+1}\ (\rho_{st}\ c) \\ & & \text{add}\ R_i\ R_i\ R_{i+1} \end{array}$$

Pointer in C

Computing with pointers means

1. to create pointers, that is, to obtain the address of a variable;
2. to dereference pointers, that is, to access the pointed-to memory

&x

Creating pointers:

- through the use of the address-of operator: `&` yields a pointer to a variable, that is, its ($\widehat{=}$l-value). Thus define:

$$\text{code}_R^i\ \&e\ \rho \quad = \quad \text{code}_L^i\ e\ \rho$$

Example:
Let `struct { int a; int b; } x;` with $\rho = \{x \mapsto 13\}$ and $\rho_{st} = \{a \mapsto 0, b \mapsto 1\}$.
Then

$$\begin{array}{lll} \text{code}_L^i\ (x.b)\ \rho & = & \text{loadc } R_{i+1}\ 13 \\ & & \text{loadc } R_i\ 1 \\ & & \text{add } R_i\ R_i\ R_{i+1} \end{array}$$

$\text{code}_R^i\ \&x.b$

Dereferencing Pointers

Applying the `*` operator to an expression e yields the content of the cell whose l-value is stored in e:

$$\begin{array}{lll} \text{code}_R^i\ {*e}\ \rho & = & \text{code}_R^i\ e\ \rho \\ & & \text{load } R_i\ R_i \end{array}$$

Example: Consider

```
struct t { int a[7]; struct t *b; };
int i,j;
struct t *pt;
```

and the expression $e \equiv$ `((pt -> b) -> a)[i+1]`

Since `e->a` $\equiv$ `(*e).a` we get:

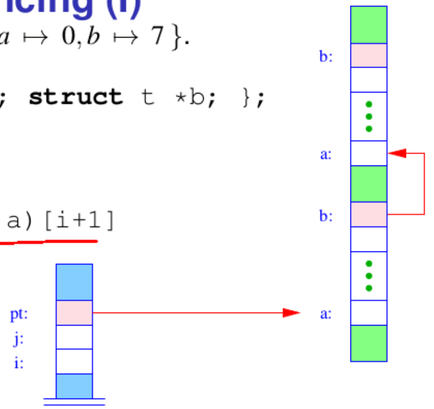
$$\begin{array}{lll} \text{code}_L^i\ (e \to a)\ \rho & = & \text{code}_L^i\ e\ \rho \\ & & \text{loadc } R_{i+1}\ (\rho\ a) \\ & & \text{add } R_i\ R_i\ R_{i+1} \end{array}$$

Translation of Dereferencing (I)

Let $\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$.

```
struct t { int a[7]; struct t *b; };
int i,j;
struct t *pt;
```

Translate $e \equiv$ `((pt -> b) -> a)[i+1]`



Then we have:

$$\begin{array}{lllll} \text{code}_L^i\ e\ \rho & = & \text{code}_L^i\ ((pt \to b) \to a)\ \rho & = & \text{code}_L^i\ ((pt \to b) \to a)\ \rho \\ & & \text{code}_R^{i+1}\ (i+1)\ \rho & & \text{loada } R_{i+1}\ 1 \\ & & \text{loadc } R_{i+2}\ 1 & & \text{loadc } R_{i+2}\ 1 \\ & & \text{mul } R_{i+1}\ R_{i+1}\ R_{i+2} & & \text{add } R_{i+1}\ R_{i+1}\ R_{i+2} \\ & & \text{add } R_i\ R_i\ R_{i+1} & & \text{loadc } R_{i+2}\ 1 \\ & & & & \text{mul } R_{i+1}\ R_{i+1}\ R_{i+2} \\ & & & & \text{add } R_i\ R_i\ R_{i+1} \end{array}$$

Translation of Dereferencing (II)

For dereferences of the form `(*e).a` the r-value is equal to the dereferencing of the l-value of `e` plus the offset of `a`. Thus, we define:

$$\begin{array}{lllll} \text{code}_L^i\ ((pt \to b) \to a)\ \rho & = & \text{code}_L^i\ (pt \to b)\ \rho & = & \text{loada } R_i\ 3 \\ & & \text{loadc } R_{i+1}\ 0 & & \text{loadc } R_{i+1}\ 7 \\ & & \text{add } R_i\ R_i\ R_{i+1} & & \text{add } R_i\ R_i\ R_{i+1} \\ & & & & \text{load } R_i\ R_i \\ & & & & \text{loadc } R_{i+1}\ 0 \\ & & & & \text{add } R_i\ R_i\ R_{i+1} \end{array}$$

Overall, we obtain the sequence:

loada R_i 3	load R_i R_i	loada R_{i+1} 1	loadc R_{i+2} 1
loadc R_{i+1} 7	loadc R_{i+1} 0	loadc R_{i+2} 1	mul R_{i+1} R_{i+1} R_{i+2}
add R_i R_i R_{i+1}	add R_i R_i R_{i+1}	add R_{i+1} R_{i+1} R_{i+2}	add R_i R_i R_{i+1}

Passing Compound Parameters

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

⇝ How do we pass parameters that are not basis types?

Passing Compound Parameters

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

⇝ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure

Passing Compound Parameters

f() {

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

~~f(point_t)~~

⇝ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure
- problem: *callee* could modify the structure

Passing Compound Parameters

Consider the following declarations:

f() {

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

}

⇝ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure
- problem: *callee* could modify the structure
- solution: *caller* passes a pointer to a copy

f() {

p' = point_t;

distToOrigin(&p')

}

Passing Compound Parameters

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

⇝ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure
- problem: *callee* could modify the structure
- solution: *caller* passes a pointer to a copy

$$\begin{aligned} \text{code}_R^i\ e\ \rho \quad = \quad & \text{code}_L^{i+1}\ e\ \rho \\ & \text{move}\ R_i\ k\ R_{i+1} \qquad e \text{ a structure of size } k \end{aligned}$$

Passing Compound Parameters

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

⇝ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure
- problem: *callee* could modify the structure
- solution: *caller* passes a pointer to a copy

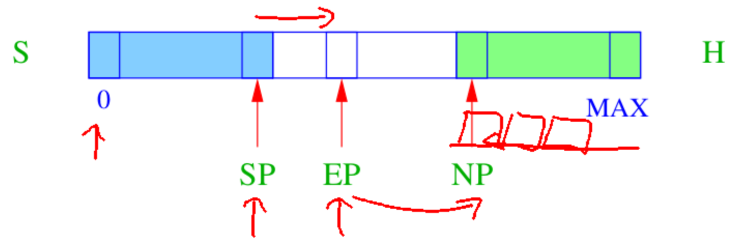
$$\begin{aligned} \text{code}_R^i\ e\ \rho \quad = \quad & \text{code}_L^{i+1}\ e\ \rho \\ & \text{move}\ R_i\ k\ R_{i+1} \qquad e \text{ a structure of size } k \end{aligned}$$

new instruction: move

Chapter 3:
The Heap

The Heap

Pointer all the use dynamic data structure that are allocated on the heap and whose life-time does not have to follow the LIFO-allocation scheme of the stack.
⇝ we need an arbitrary large memory area H, called the heap; implementation:



NP	$\widehat{=}$	new pointer; points to the first unused heap cell
EP	$\widehat{=}$	extreme pointer; points to the cell that SP may maximally point (changes during function call/return).

Invariant of Heap and Stack

- the stack and the heap may not overlap

Invariant of Heap and Stack

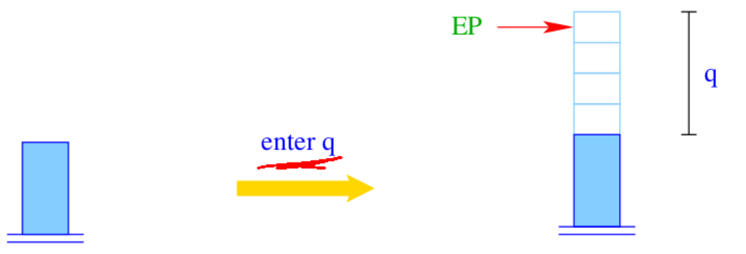
- the stack and the heap may not overlap
- an overlap may only happen when SP is incremented (stack overflow) or
- when NP is decremented (out of memory)
  - in contrast to a stack overflow, an out of memory error can be communicated to the programmer
  - `malloc` returns `NULL` in this case which is defined as `(void*) 0`

Invariant of Heap and Stack

- the stack and the heap may not overlap
- an overlap may only happen when SP is incremented (stack overflow) or
- when NP is decremented (out of memory)
  - in contrast to a stack overflow, an out of memory error can be communicated to the programmer
  - `malloc` returns `NULL` in this case which is defined as `(void*) 0`
- EP reduces the necessary check to a single check upon entering a function
- the check for each heap allocation remains necessary

Reserving Memory on the Stack

The instruction enter q sets EP to the last stack cell that this function will use.



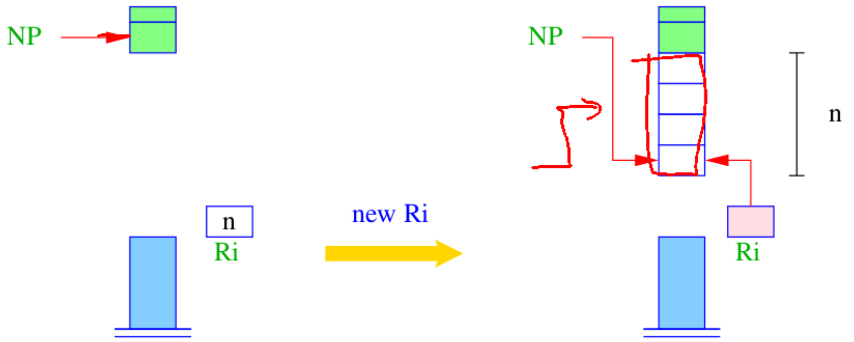
EP = SP + q;
if (EP $\geq$ NP)
    error ("stack overflow");

Dynamically Allocated Memory

In order to implement `malloc`, its use is directly translated into instructions:

- a call to `malloc` must return a pointer to a heap cell:

$$\begin{aligned}\text{code}^i_{\mathrm{R}}\ \textbf{malloc}\,(e)\ \rho \quad &= \quad \text{code}^i_{\mathrm{R}}\ e\ \rho\\ &\quad\ \ \text{new}\ R_i\end{aligned}$$

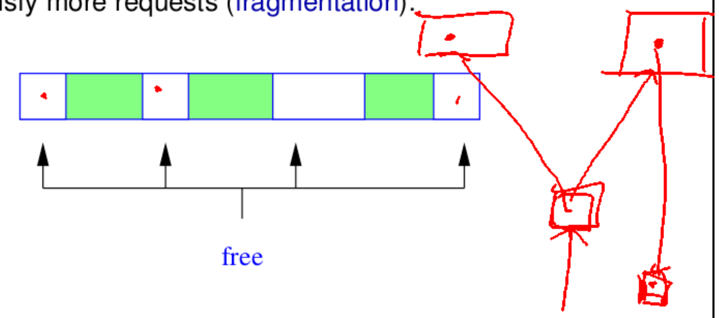



```
if (NP - R[i] <= EP) R[i] = NULL; else {
  NP = NP - R[i];
  R[i] = NP;
}
```


Freeing Memory

A region allocated with `malloc` may be deallocated using `free`.
Problems:

- the freed memory could still be accessed, thereby accessing memory that may have a new owner (dangling references).
- interleaving `malloc` and `free` may not leave a larger enough block to satisfy more requests (fragmentation):



Possible Implementations of `free`

1. Leave the problem of dangling pointers to the programmer. Use a data structure to manage allocated and free memory. ⇝ `malloc` becomes expensive
2. Do nothing:

$$\text{code}^i\ \textbf{free}(e)\ \rho \quad = \quad \text{code}^i_{\mathrm{R}}\ e\ \rho$$

⇝ simple and efficient, but not for reactive programs

3. Use an automatic, possibly "conservative" garbage collection, that occasionally runs to reclaim memory that certainly is not in use anymore. Make this re-claimed memory available again to `malloc`.

Chapter 4:

Translating Functions and Programs with Heap

+1

6

Instructions for Starting a Program

A program $P = F_1; \ldots F_n$ has to have one `main` function.

$$
\begin{array}{rcl}
\text{code}^1\ P\ \rho & = & \text{enter}\ (k+3) \\
 & & \text{alloc}\ k \\
 & & \text{loadc}\ R_1\ \texttt{\_main} \\
 & & \cancel{\text{saveloc}\ R_1\ R_0} \\
 & & \text{mark} \\
 & & \text{call}\ R_1 \\
 & & \cancel{\text{restoreloc}\ R_1\ R_0} \\
 & & \text{halt} \\
\_f_1: & & \text{code}^i\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
\_f_n: & & \text{code}^i\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Instructions for Starting a Program

A program $P = F_1; \ldots F_n$ has to have one `main` function.

$$
\begin{array}{rcl}
\text{code}^1\ P\ \rho & = & \text{enter}\ (k+3) \\
 & & \text{alloc}\ k \\
 & & \text{loadc}\ R_1\ \texttt{\_main} \\
 & & \text{saveloc}\ R_1\ R_0 \\
 & & \text{mark} \\
 & & \text{call}\ R_1 \\
 & & \text{restoreloc}\ R_1\ R_0 \\
 & & \text{halt} \\
\_f_1: & & \text{code}^i\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
\_f_n: & & \text{code}^i\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

assumptions:

- k are the number of stack location set aside for global variables
- saveloc $R_1\ R_0$ has no effect (i.e. it backs up no register)
- ρ contains the address of all functions and global variable

Translation of Functions

The translation of a function is modified as follows:

$$
\begin{array}{rcl}
\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho & = & \text{enter}\ q \\
 & & \text{alloc}\ k \\
 & & \text{move}\ R_{l+1}\ R_{-1} \\
 & & \vdots \\
 & & \text{move}\ R_{l+n}\ R_{-n} \\
 & & \text{code}^{l+n+1}\ ss\ \rho' \\
 & & \text{return}
\end{array}
$$

side conditions:

Randbedinungen:

Translation of Functions

The translation of a function is modified as follows:

$$
\begin{array}{rcl}
\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho & = & \text{enter}\ q \\
 & & \text{alloc}\ k \\
 & & \text{move}\ R_{l+1}\ R_{-1} \\
 & & \vdots \\
 & & \text{move}\ R_{l+n}\ R_{-n} \\
 & & \text{code}^{l+n+1}\ ss\ \rho' \\
 & & \text{return}
\end{array}
$$

Randbedinungen:

- enter ensures that enough stack space is available (q: number of required stack cells)

Translation of Functions

The translation of a function is modified as follows:

$$\begin{array}{lll} \text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho & = & \text{enter}\ q \\ & & \text{alloc}\ k \\ & & \text{move}\ R_{l+1}\ R_{-1} \\ & & \vdots \\ & & \text{move}\ R_{l+n}\ R_{-n} \\ & & \text{code}^{l+n+1}\ ss\ \rho' \\ & & \text{return} \end{array}$$

Randbedinungen:

- enter ensures that enough stack space is available (q: number of required stack cells)
- alloc reserves space on the stack for local variables ($k < q$)

Translation of Function Calls

The function call $\texttt{g}(e_1, \ldots e_n)$ is translated as folllows:

$$\begin{array}{lll} \text{code}^i_{\text{R}}\ \texttt{g}(e_1, \ldots e_n)\ \rho & = & \text{code}^i_{\text{R}}\ \texttt{g}\ \rho \\ & & \text{code}^{i+1}_{\text{R}}\ e_1\ \rho \\ & & \vdots \\ & & \text{code}^{i+n}_{\text{R}}\ e_n\ \rho \\ & & \text{move}\ R_{-1}\ R_{i+1} \\ & & \vdots \\ & & \text{move}\ R_{-n}\ R_{i+n} \\ & & \text{saveloc}\ R_1\ R_{i-1} \\ & & \text{mark} \\ & & \text{call}\ R_i \\ & & \text{restoreloc}\ R_1\ R_{i-1} \\ & & \text{pop}\ k \\ & & \text{move}\ R_i\ R_0 \end{array}$$

Translation of Function Calls

The function call $\texttt{g}(e_1, \ldots e_n)$ is translated as folllows:

$$\begin{array}{lll} \text{code}^i_{\text{R}}\ \texttt{g}(e_1, \ldots e_n)\ \rho & = & \text{code}^i_{\text{R}}\ \texttt{g}\ \rho \\ & & \text{code}^{i+1}_{\text{R}}\ e_1\ \rho \\ & & \vdots \\ & & \text{code}^{i+n}_{\text{R}}\ e_n\ \rho \\ & & \text{move}\ R_{-1}\ R_{i+1} \\ & & \vdots \\ & & \text{move}\ R_{-n}\ R_{i+n} \\ & & \text{saveloc}\ R_1\ R_{i-1} \\ & & \text{mark} \\ & & \text{call}\ R_i \\ & & \text{restoreloc}\ R_1\ R_{i-1} \\ & & \text{pop}\ k \\ & & \text{move}\ R_i\ R_0 \end{array}$$

Difference to previous scheme:

- we assume that g has n arguments, that is, it is not *variadic*
- new instruction pop : removes stack cells which have been allocated in $\text{code}^{i+j}_{\text{R}}\ e_j\ \rho$

Peephole Optimization

The generated code contains many redundancies, such as:

$\text{move}\ R_7\ R_7$

$\text{pop}\ 0$

$\text{move}\ R_5\ R_7$
$\text{mul}\ R_4\ R_4\ R_7$

Peephole optimization matches certain patterns and replaces them by simpler patterns

Realistic Register Machiens

The R-CMa is a virtual machine that makes it easy to generate code.

Realistic Register Machiens

The R-CMa is a virtual machine that makes it easy to generate code.

- real processors have a fixed number of registers

Realistic Register Machiens

The R-CMa is a virtual machine that makes it easy to generate code.

- real processors have a fixed number of registers
- the infinite set of *virtual* registers of the R-CMa must be mapped onto a finite set of processor registers
- idea: use a register R_i that is currently not in use for the content of R_j
- in case the program needs more register at one time than available, we need to spill registers onto the stack

We thus require solutions to the following problems:

Realistic Register Machiens

The R-CMa is a virtual machine that makes it easy to generate code.

- real processors have a fixed number of registers
- the infinite set of *virtual* registers of the R-CMa must be mapped onto a finite set of processor registers
- idea: use a register R_i that is currently not in use for the content of R_j
- in case the program needs more register at one time than available, we need to spill registers onto the stack

We thus require solutions to the following problems:

- determine when a register is not live (in use)
- map several virtual registers to the same processor register if they are not live at the same time

these problems are addressed in the lecture on *Program Optimization*.

Register Coloring for the `fac`-Function

Note: def-use liveness

```
int fac(int x) {
 if (x<=0) then
   return 1;
 else
   return x*fac(x-1);
}
```

−1 0 1 2 3 4

_fac: enter 5
move R_1 R_{-1}
move R_2 R_1
loadc R_3 0
leq R_2 R_2 R_3
jumpz R_2 _A
loadc R_2 1
move R_0 R_2
return
jump _B

−1 0 1 2 3 4

_A: move R_2 R_1
move R_3 R_1
loadc R_4 1
sub R_3 R_3 R_4
move R_{-1} R_3
loadc R_3 _fac
saveloc R_1 R_2
mark
call R_3
restoreloc R_1 R_2
move R_3 R_0
mul R_2 R_2 R_3
move R_0 R_2
return
_B: return

Register Coloring for the `fac`-Function

Note: def-use liveness coloring

```
int fac(int x) {
 if (x<=0) then
   return 1;
 else
   return x*fac(x-1);
}
```

−1 0 1 2 3 4

_fac: enter 5
move R_1 R_{-1}
move R_0 R_1
loadc R_{-1} 0
leq R_0 R_0 R_{-1}
jumpz R_2 _A
loadc R_2 1
move R_0 R_2
return
jump _B

−1 0 1 2 3 4

_A: move R_2 R_1
move R_0 R_1
loadc R_{-1} 1
sub R_0 R_0 R_{-1}
move R_{-1} R_0
loadc R_3 _fac
saveloc R_1 R_2
mark
call R_3
restoreloc R_1 R_2
move R_0 R_0
mul R_2 R_2 R_0
move R_0 R_2
return
_B: return

Outlook

register allocation has several other uses:

- remove unnecessary move instructions

Outlook

register allocation has several other uses:

- remove unnecessary move instructions
- decide which variable to spill onto the stack
  - $\leadsto$ this might in turn require more registers
- translation into single static assignment form simplifies analysis
- optimal register allocation possible (but registers might need to be permuted at the end of basic blocks)

$\leadsto$ lecture on *Program Optimization*

schematically presented liveness-analysis can be improved:

- x is only live after $x \leftarrow y + 1$ if y was live
- saveloc keeps registers unnecessarily alive $\leadsto$ intermediate representation

Outlook

register allocation has several other uses:

- remove unnecessary move instructions
- decide which variable to spill onto the stack
  - $\rightsquigarrow$ this might in turn require more registers
- translation into single static assignment form simplifies analysis
- optimal register allocation possible (but registers might need to be permuted at the end of basic blocks)

$\rightsquigarrow$ lecture on *Program Optimization*

schematically presented liveness-analysis can be improved:

- x is only live after $x \leftarrow y + 1$ if y was live
- saveloc keeps registers unnecessarily alive $\rightsquigarrow$ intermediate representation
- are there *optimal* rules for the liveness-analysis?