**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**

\[
\begin{align*}
    \text{r-value of } x &= \text{content of } x \\
    \text{l-value of } x &= \text{address of } x
\end{align*}
\]

Compute r- and l-value in register \( R_i \):

<table>
<thead>
<tr>
<th>code( R_i ) e ( \rho )</th>
<th>generates code to compute the r-value of ( e ), given the environment ( \rho )</th>
</tr>
</thead>
<tbody>
<tr>
<td>code( i ) e ( \rho )</td>
<td>analogously for the l-value</td>
</tr>
</tbody>
</table>

**note:**

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

---

**Address Environment**

A variable is stored in four different ways:

- Global: a variable is global
- Local: a variable is stored on the stack frame
- Register: a variable is stored in a local register \( R_i \) or a global register \( R \)
Address Environment

A variable is stored in four different ways:

- **Global**: a variable is global
- **Local**: a variable is stored on the stack frame
- **Register**: a variable is stored in a local register $R_i$ or a global register $R_i$

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

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

**Observe**: a variable $x$ can only have one entry in $\rho$
However:

**Address Environment**

A variable is stored in four different ways:

- **Global**: a variable is global
- **Local**: a variable is stored on the stack frame
- **Register**: a variable is stored in a local register $R_i$ or a global register $R_i$

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

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

**Observe**: a variable $x$ can only have one entry in $\rho$

However:

- $\rho$ may 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 $R_1 \ldots R_n$
- separate compilation becomes difficult, since code of function depends on $n$

Furthermore:

**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:
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., $\rho x = (L, o)$ or $\rho x = (G, o)$

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., $\rho x = (L, o)$ or $\rho x = (G, o)$
- an access to an array is always done through a pointer, hence, it must be stored in memory

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 $\rho(x) = (R,f)$ then:
$$
code_R x = e_2 \rho = \text{code}_R e_2 \rho
$$
move $R_j R_i$
Translation of Statements

Statements such as \(x = 2 \cdot y\) have so far been translated by:
- computing the r-value of \(2 \cdot y\) in register \(R_i\),
- copying the content of \(R_i\) into the register \(\rho(x)\)

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

\[
\text{code}_R x = e_2 \rho = \text{code}_R e_2 \rho
\]

\[
\text{move } R_j, R_i
\]

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

Translation of L-Values

new instruction: \(\text{store } R_i, R_j\) with semantics \(S[R_i] = R_j\)

\[
\text{code}_L e \rho = \text{code}_L e \rho
\]

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

Translation of Statements

Statements such as \(x = 2 \cdot y\) have so far been translated by:
- computing the r-value of \(2 \cdot y\) in register \(R_i\),
- copying the content of \(R_i\) into the register \(\rho(x)\)

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

\[
\text{code}_R x = e_2 \rho = \text{code}_R e_2 \rho
\]

\[
\text{move } R_j, R_i
\]

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

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 \text{store} instruction

Translation of L-Values

new instruction: \(\text{store } R_i, R_j\) with semantics \(S[R_i] = R_j\)

definition for assignments:

\[
\text{code}_L e \rho = \text{code}_L e \rho
\]

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

- Thus, for the case \(e_1 = x\) and \(\rho x = \langle R, f \rangle\) does not hold:

\[
\text{code}_L e_1 = e_2 \rho = \text{code}_L e_2 \rho
\]

\[
\text{code}_L^{i+1} e_1 \rho
\]

\[
\text{store } R_{i+1}, R_i
\]
Translation of L-Values

new instruction: store \( R_i R_j \) with semantics \( S[R_i] = R_j \)

definition for assignments:

\[ \text{code}^i_e \rho = \text{code}^j_e \rho \]

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

- Thus, for the case \( e_1 = x \) and \( \rho x = \langle R_i, j \rangle \) does not hold:

\[ \text{code}^k_e \rho = \text{code}^k_{e_2} \rho \]

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

\[ \text{code}^i_x \rho = \text{loade} R_i a \]

Allocating Memory for Local Variables

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

- \text{alloc} \( k \)
- \text{SP} = \text{SP} + k;

- \text{pop} \( k \)
- \text{SP} = \text{SP} - k;

The instruction \text{alloc} \( k \) reserves space for \( k \) variables on the stack, \text{pop} \( k \) frees this space again.

Access to Local Variables

Accesses to local variables are relative to FP. We therefore modify \( \text{code}^c_L \) to cater for variables in memory.

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

\[ \text{code}^c_x \rho = \text{loadc} R_i a \text{ if } \rho x = \langle L, a \rangle \]

Instruction \text{loadc} \( R_i k \) computes the sum of FP and \( k \).

General Computation of the L-Value of a Variable

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

\[ \text{code}^c_x \rho = \begin{cases} \text{loade} R_i a & \text{if } \rho x = \langle G, a \rangle \\ \text{loadc} R_i a & \text{if } \rho x = \langle L, a \rangle \end{cases} \]
**General Computation of the L-Value of a Variable**

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

\[
\text{code}_L \ x \ \rho = \begin{cases} 
\text{ldc} \ R, \ a & \text{if } \rho \ x = \langle G, a \rangle \\
\text{ladr} \ R, \ a & \text{if } \rho \ x = \langle L, a \rangle 
\end{cases}
\]

**Note:** for \( \rho \ x = \langle R, j \rangle \) the function \( \text{code}_L \) 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 \( \text{load } R_i, R_j \) sets \( R_i \) to the value at address \( R_j \).

**Thus:** \( \text{load } R_i, R_j; \) \( R_j \) sets \( R_i \) to \( x \) where \( \rho x = (L, a) \).

---

Macro-Command for Accessing Local Variables

**Define:** the command \( \text{load } R_i, R_j \) sets \( R_i \) to the value at address \( R_j \).

**Thus:** \( \text{load } R_i, a; \) \( \text{load } R_j, R_i; \) sets \( R_j \) to \( x \) where \( \rho x = (L, a) \).

**In general:** Load variable \( x \) into register \( R_j \):

\[
\text{code}_L \ x \ \rho = \begin{cases} 
\text{load } R_i, a & \text{if } \rho x = (G, a) \\
\text{load } R_i, a & \text{if } \rho x = (L, a) \\
\text{move } R_i, R_j & \text{if } \rho x = (R, i) 
\end{cases}
\]
Macro-Command for Accessing Local Variables

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

Thus: \texttt{loadrc }R_i;R_j;\texttt{ sets }R_i\texttt{ to }x\texttt{ where }\rho x = (L,a).

In general: Load variable \(x\) into register \(R_i\):

\[
\text{code}_{R_i}^L x \rho = \begin{cases} 
\text{loada } R_i; a & \text{if } \rho x = (G,a) \\
\text{loadr } R_i; a & \text{if } \rho x = (L,a) \\
\text{move } R_i; R_j & \text{if } \rho x = (R, \bar{r}) \end{cases}
\]

Analogously: for write operations we define:

\[
\begin{align*}
\text{storer } a & R_j = \text{loadrc } R_i; a \\
\text{storea } a & R_j = \text{loadr } R_i; a \\
\end{align*}
\]

i.e. \texttt{storea }\(R_j\) is a \textit{macro}. Define special case (where \(\rho x = (G,a)\)):

\[
\text{code}_{R_i}^L x = e_2 \rho = \begin{cases} 
\text{code}_{R_i}^L e_2 \rho & \text{if } \rho x = (G,a) \\
\text{code}_{R_i}^{L+1} x \rho & \text{if } \rho x = (R, \bar{r}) \\
\text{store } R_{i+1}; R_i & \text{if } \rho x = (F[+1], \bar{r}) \\
\end{cases}
\]

Data Transfer Instructions of the R-CMAs

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

<table>
<thead>
<tr>
<th>instruction</th>
<th>semantics</th>
<th>intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load (R_i; R_j)</td>
<td>(R_i \leftarrow S[R_j])</td>
<td>load value from address</td>
</tr>
<tr>
<td>load (R_i; c)</td>
<td>(R_i \leftarrow S[c])</td>
<td>load global variable</td>
</tr>
<tr>
<td>loadr (R_i; c)</td>
<td>(R_i \leftarrow S[F!!P + c])</td>
<td>load local variable</td>
</tr>
<tr>
<td>store (R_i; R_j)</td>
<td>(S[R_j] \leftarrow R_i)</td>
<td>store value at address</td>
</tr>
<tr>
<td>storea (R_i; R_j)</td>
<td>(S[c] \leftarrow R_i)</td>
<td>write global variable</td>
</tr>
<tr>
<td>std (R_i; R_j)</td>
<td>(S[F!!P + c] \leftarrow R_i)</td>
<td>write local variable</td>
</tr>
</tbody>
</table>

instructions for computing addresses:

<table>
<thead>
<tr>
<th>instruction</th>
<th>semantics</th>
<th>intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load (R_i; c)</td>
<td>(R_i \leftarrow c)</td>
<td>load constant</td>
</tr>
<tr>
<td>loadrc (R_i; c)</td>
<td>(R_i \leftarrow F!!P + c)</td>
<td>load constant relative to FP</td>
</tr>
</tbody>
</table>

instructions for general data transfer:

<table>
<thead>
<tr>
<th>instruction</th>
<th>semantics</th>
<th>intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>move (R_i; R_j)</td>
<td>(R_i \leftarrow R_j)</td>
<td>transfer value between regs</td>
</tr>
</tbody>
</table>
| move \(R_i; k\) | \([S[SP + i + 1] \leftarrow S[R_j + \bar{r}]_{i=0}^{i+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:

\(\checkmark\) \textbf{global} variables, defined outside of functions (or as \texttt{static}); \(<G,7>)

\(\checkmark\) \textbf{local} (automatic) variables, defined inside functions, accessible by pointers; \(<R,5>)

\(\checkmark\) \textbf{register} (automatic) variables, defined inside functions.

Example:

\begin{verbatim}
    int x, y;
    void f(int v, int w) {
        int a;
        if (a>0) {
            int b;
            g(b);
            } else {
            int c;
            }
        }
    }
\end{verbatim}
Determining the Address-Environment

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

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

Example:

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

<table>
<thead>
<tr>
<th>v</th>
<th>(\rho(v))</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>(G, 0)</td>
</tr>
<tr>
<td>y</td>
<td>(G, 1)</td>
</tr>
<tr>
<td>v</td>
<td>(R, -1)</td>
</tr>
<tr>
<td>w</td>
<td>(R, -2)</td>
</tr>
<tr>
<td>a</td>
<td>(R, 1)</td>
</tr>
<tr>
<td>b</td>
<td>(L, 0)</td>
</tr>
<tr>
<td>c</td>
<td>(R, 2)</td>
</tr>
</tbody>
</table>

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:**

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

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

<table>
<thead>
<tr>
<th>(p)</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>(R, -1)</td>
</tr>
<tr>
<td>&quot;World&quot;</td>
<td>(L, -3)</td>
</tr>
<tr>
<td>5</td>
<td>(L, -4)</td>
</tr>
<tr>
<td>12</td>
<td>(L, -5)</td>
</tr>
</tbody>
</table>

**idea:**

- push **variadic** parameters from right to left onto the stack
- the first parameter lies right below PC, FP, EP
- for a prototype \(f(x_1, \ldots, x_k, \ldots)\) we set:
  \[
  x_1 \mapsto (R, -1) \quad x_k \mapsto (R, -k) \\
  x_{k+1} \text{ at } (L, -3) \quad x_{k+2} \text{ at } (L, -3 - |x_{k+1}| - \ldots - |x_{k+1}|) 
  \]

Translation of Array Accesses

Extend `code` and `code` with indexed array accesses.

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

Chapter 2: Arrays and Pointers
Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let \( t[\alpha] \ a; \) be the declaration of an array \( a. \)
In order to compute the address of \( a[i], \) we need to compute
\[
\rho a + |t| \ast (R{-}\text{Wert von } i).
\]
Thus:

\[
\begin{align*}
\text{code}_L e_2[e_1] \ \rho &= \text{code}_R e_1 \ \rho \\
\text{code}_R e_2 \ \rho \\
\text{loadc } R_{i+2} \ [t] \\
\text{mul } R_i \ R_0 \ R_{i+2} \\
\text{add } R_i \ R_i \ R_{i+1}
\end{align*}
\]

Note:

- An array in C is simply a pointer. The declared array \( a \) is a pointer constant, whose r-value is the address of the first field of \( a. \)
- Formally, we compute the r-value of a field \( e \) as
  \[
  \text{code}_R e \ \rho = \text{code}_L e \ \rho
  \]
- in C the following are equivalent (as l-value, not as types):
  \[
  \]

C structs (Records)

Note:

The same field name may occur in different structs.

Here: The component environment \( \rho_M \) relates to the currently translated structure \( s. \)

Let \( \text{struct } \{ \text{int } a; \text{int } b; \} \ x; \) 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 \( \rho_M \) with addresses of fields that are relative to the beginning of \( x, \) here \( a \to 0, \ b \to 1. \)

In general, let \( t \equiv \text{struct } \{ t_1 \ v_1; \ldots; t_k \ v_k \}, \) then

\[
|t| := \sum_{i=1}^{k} |t_i| \\
\rho_M v_i := 0 \\
\rho_M v_i := \rho_M v_{i-1} + |t_{i-1}| \quad \text{für } i > 1
\]

We obtain:

\[
\begin{align*}
\text{code}_L (e, c) \ \rho &= \text{code}_L e \ \rho \\
\text{loadc } R_{i+1} (\rho_M e) \\
\text{add } R_i \ R_i \ R_{i+1}
\end{align*}
\]
**Pointer in C**

Computing with pointers means

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

Creating pointers:

- through the use of the address-of operator: $\&$ yields a pointer to a variable, that is, its (≡-value). Thus define:

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

**Example:**

Let

$$\text{struct} \{ \text{int } a; \text{ int } b; \} \ x; \text{ with } \rho = \{x \mapsto 13\} \text{ and } \rho_R = \{a \mapsto 0, b \mapsto 1\}.$$  

Then

$$\begin{align*} \text{code}_R^e (x, b) \rho &= \text{loade } R_{i+1} 13 \\
&\quad \text{loade } R_i 1 \\
&\quad \text{add } R_i R_i R_{i+1} \end{align*}$$

**Translation of Dereferencing (I)**

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

$$\begin{align*} \text{struct } t \{ \text{int } a[7]; \text{ struct } *b; \}; \\
\text{int } i,j; \\
\text{struct } t *pt; \end{align*}$$

Translate $e \equiv (\text{pt } \rightarrow \text{ b }) \rightarrow \text{ a}\}[i+1]$

Then we have:

$$\begin{align*} \text{code}_L^e e \rho &= \text{code}_L^e ((\text{pt } \rightarrow \text{ b }) \rightarrow \text{ a}) \rho \\
&= \text{loade } R_{i+1} 1 \\
&\quad \text{loade } R_i 1 \\
&\quad \text{mul } R_{i+1} R_{i+1} R_{i+2} \\
&\quad \text{add } R_i R_i R_{i+1} \end{align*}$$

**Dereferencing Pointers**

Applying the $\ast$ operator to an expression $e$ yields the content of the cell whose l-value is stored in $e$:

$$\text{code}_R^e \ast e \rho = \text{code}_L^e e \rho$$

**Example:** Consider

$$\begin{align*} \text{struct } t \{ \text{int } a[7]; \text{ struct } *b; \}; \\
\text{int } i,j; \\
\text{struct } t *pt; \end{align*}$$

and the expression $e \equiv ((\text{pt } \rightarrow \text{ b }) \rightarrow \text{ a})[i+1]$

Since $e \rightarrow a \equiv (*e).a$, we get:

$$\begin{align*} \text{code}_L^e (e \rightarrow a) \rho &= \text{code}_L^e e \rho \\
&= \text{loade } R_{i+1} (\text{pt } \rightarrow \text{ a}) \\
&\quad \text{add } R_i R_i R_{i+1} \end{align*}$$

**Translation of Dereferencing (II)**

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

$$\text{code}_L^e ((\text{pt } \rightarrow \text{ b }) \rightarrow \text{ a}) \rho = \text{loade } R_i 3$$

$$\begin{align*} \text{loade } R_{i+1} 1 \\
&\quad \text{loade } R_i 1 \\
&\quad \text{mul } R_{i+1} R_{i+1} R_{i+2} \\
&\quad \text{add } R_i R_i R_{i+1} \end{align*}$$

**Overall, we obtain the sequence:**

$$\begin{align*} \text{loade } R_i 3 \\
\text{loade } R_{i+1} 7 \\
\text{add } R_i R_i R_{i+1} \\
\text{mul } R_{i+1} R_{i+1} R_{i+2} \\
\text{add } R_i R_i R_{i+1} \end{align*}$$
Passing Compound Parameters

Consider the following declarations:

```c
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

```c
int f(point_t p)
{
  // Use p instead of *p
}
```
Passing Compound Parameters

Consider the following declarations:

```c
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

\[
\text{code}^l e \rho = \text{code}^{l+1} e \rho \\
\text{move } R_i k R_{i+1} \quad \text{a structure of size } k
\]

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**:

```
S               H
0
SP  EP  NP
```

NP \( \triangleq \) *new pointer*: points to the first unused heap cell

EP \( \triangleq \) *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
- 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 > 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:

\[
\text{code}_R \text{ malloc}(e) \rho = \text{code}_K e \rho
\]

Possible Implementations of `free`

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

\[
\text{code}_K \text{ free}(e) \rho = \text{code}_K e \rho
\]

~ simple and efficient, but not for reactive programs
- 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`

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).
Instructions for Starting a Program

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

\[
\text{code}^\dagger P \rho = \begin{align*}
&\text{enter } (k + 3) \\
&\text{alloc } k \\
&\text{load } R_1_{\text{main}} \\
&\text{saveloc } R_1 R_0 \\
&\text{mark} \\
&\text{call } R_1 \\
&\text{restoreloc } R_1 R_0 \\
&\text{halt} \\
&f_1 : \text{code}^\dagger F_1 \rho \oplus \rho_f \\
&\vdots \\
&f_n : \text{code}^\dagger F_n \rho \oplus \rho_f
\end{align*}
\]

Translation of Functions

The translation of a function is modified as follows:

\[
\text{code}^\dagger t_r \in (\text{args}) \{ \text{decls } ss \} \rho = \begin{align*}
&\text{enter } q \\
&\text{alloc } k \\
&\text{move } R_{i+1} R_{i-1} \\
&\vdots \\
&\text{move } R_{i+n} R_{i-n} \\
&\text{code}^\dagger_{i+n+1} ss \rho' \\
&\text{return}
\end{align*}
\]

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)
- $\rho$ contains the address of all functions and global variable

Translation of Functions

The translation of a function is modified as follows:

\[
\text{code}^\dagger t_r \in (\text{args}) \{ \text{decls } ss \} \rho = \begin{align*}
&\text{enter } q \\
&\text{alloc } k \\
&\text{move } R_{i+1} R_{i-1} \\
&\vdots \\
&\text{move } R_{i+n} R_{i-n} \\
&\text{code}^\dagger_{i+n+1} ss \rho' \\
&\text{return}
\end{align*}
\]

Randbedingungen:

- 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:

\[
\text{code}^+ t_i \in (\text{args}) \{ \text{decls ss} \} \rho = \begin{array}{l}
\text{enter } g \\
\text{alloc } k \\
\text{move } R_{i+1} R_{i-1} \\
\vdots \\
\text{move } R_{i+n} R_{i-n} \\
\text{code}^{i+n+1} s s' \rho' \\
\text{return}
\end{array}
\]

Randbedingungen:

- 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 \(g(e_1, \ldots, e_n)\) is translated as follows:

\[
\text{code}^+ g(e_1, \ldots, e_n) \rho = \begin{array}{l}
\text{code}^+ g \rho \\
\text{code}^{i+1} e_1 \rho \\
\vdots \\
\text{code}^{i+n} e_n \rho \\
\text{move } R_{i-1} R_{i+1} \\
\vdots \\
\text{move } R_{i-n} R_{i+n} \\
\text{saveloc } R_{i} R_{i-1} \\
\text{mark} \\
\text{call } R_i \\
\text{restoreloc } R_{i} R_{i-1} \\
\text{pop } k \\
\text{move } R_i R_0
\end{array}
\]

Peephole Optimization

The generated code contains many redundancies, such as:

- \(\text{move } R_3 R_7\)
- \(\text{pop } 0\)
- \(\text{move } R_5 R_9\)
- \(\text{mul } R_4 R_3 R_7\)

Peephole optimization matches certain patterns and replaces them by simpler patterns.
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 coloring

```c
int fac(int x) {
    if (x<=0) then
        _A: move R_2 R_1
            move R_3 R_1
            loadc R_4 1
            sub R_3 R_3 R_4
            move R_2 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
            load R_2 R_3
            jumpz R_2 _A
            load R_0 R_2
            move R_0 R_2
            return
            _B: return
    else
        return x*fac(x-1);}
```

Outlook

Register allocation has several other uses:
- remove unnecessary `move` instructions
- decide which variable to spill onto the stack
  - `~` 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)

`~` lecture on *Program Optimization* schematically presented liveness-analysis can be improved:
- `x` is only live after `x ← y + 1` if `y` was live
- `saveloc` keeps registers unnecessarily alive `~` intermediate representation
Outlook

register allocation has several other uses:

- remove unnecessary move instructions
- decide which variable to spill onto the stack
  - 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)

→ 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 → intermediate representation
- are there optimal rules for the liveness-analysis?