Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in \textit{local} registers R_i
- intermediate results also live in \textit{local} registers R_i
- parameters \textit{global} registers R_i (with $i \leq 0$)
- global variables:

Principle of Function Call and Return

actions taken on \textit{entering} $g$:

1. compute the start address of $g$
2. compute actual parameters
3. backup of \textit{caller}-save registers
4. backup of FP, EP
5. set the new FP
6. back up of PC and jump to the beginning of $g$
7. setup new EP
8. allocate space for local variables

actions taken on \textit{leaving} $g$:

1. compute the result
2. restore FP, EP, SP
3. return to the call site in $f$
   that is, restore PC
4. restore the \textit{caller}-save registers
5. clean up stack

Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in \textit{local} registers R_i
- intermediate results also live in \textit{local} registers R_i
- parameters \textit{global} registers R_i (with $i \leq 0$)
- global variables: let's suppose there are none

convention:

- the $i$th argument of a function is passed in register R_i
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:
- automatic variables live in **local** registers \( R_i \)
- intermediate results also live in **local** registers \( R_i \)
- parameters **global** registers \( R_i \) (with \( i \leq 0 \))
- global variables: let’s suppose there are none

convention:
- the \( i \)th argument of a function is passed in register \( R_i \)
- the result of a function is stored in \( R_0 \)
- **local** registers are saved before calling a function

Definition

Let \( f \) be a function that calls \( g \). A register \( R_i \) is called
- **caller-saved** if \( f \) backs up \( R_i \) and \( g \) may overwrite it
- **callee-saved** if \( f \) does not back up \( R_i \) but must restore it before it returns

Translation of Function Calls

A function call \( g(e_1, \ldots, e_n) \) is translated as follows:

\[
\text{code}_e \ g(e_1, \ldots, e_n) \quad \rho = \quad \text{code}_g \ g(e_1, \ldots, e_n) = \quad \text{code}_e(1) \quad \rho
\]

New instructions:
- **saveloc** \( R_l R_j \) pushes the registers \( R_l, R_{l+1}, \ldots R_j \) onto the stack
- **mark** backs up the organizational cells
- **call** \( R_l \) calls the function at the address in \( R_l \)
- **restloc** \( R_l R_j \) pops \( R_l, R_{l-1}, \ldots R_j \) off the stack
Rescuing EP and FP

The instruction \textit{mark} allocates stack space for the return value and the organizational cells and backs up FP and EP.

\[
S[SP+1] = EP;\\
S[SP+2] = FP;\\
SP = SP + 2;
\]

Calling a Function

The instruction \textit{call} rescues the value of PC+1 onto the stack and sets FP and PC.

Result of a Function

The global register set is also used to communicate the result value of a function:

\[
\text{code}\_\text{return } e\rho = \text{code}\_k e\rho \\
\quad \text{move } R_0 R_i \\
\quad \text{return}
\]

\[
\text{alternative without result value:}
\]

\[
\text{code}\_\text{return } \rho = \text{return}
\]
Result of a Function

The global register set is also used to communicate the result value of a function:

\[
\text{code}^d \text{ return } e \rho = \text{code}^r \text{ return } e \rho
\]

\[
\text{move } R_0 R_i
\]

\[
\text{return}
\]

alternative without result value:

\[
\text{code}^d \text{ return } \rho = \text{return}
\]

\[
\text{move } R_0 R_i
\]

\[
\text{return}
\]

\[
\text{global} \text{ registers are otherwise not used inside a function body:}
\]

- advantage: at any point in the body another function can be called without backing up \text{global} \text{ registers}
- disadvantage: on entering a function, all \text{global} \text{ registers must be saved}

Return from a Function

The instruction return relinquishes control of the current stack frame, that is, it restores PC, EP and FP.

Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^d \text{ t_r } \text{ (args) } \{ \text{decls ss} \} \rho = \text{enter } q
\]

\[
\text{move } R_{i+1} R_{i-1}
\]

\[
\vdots
\]

\[
\text{move } R_{i+n} R_{i-n}
\]

\[
\text{code}^{d+n+1} \text{ ss } \rho'
\]

\[
\text{return}
\]

Assumptions:
Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code} \quad \text{def} \quad \psi (\text{args}) \{ \text{decls} \quad s \} \quad \rho = \quad \begin{aligned}
&\text{enter} q \\
&\text{move} \ R_{i+1} \quad R_{-1} \\
&\vdots \\
&\text{move} \ R_{i+n} \quad R_{-n} \\
&\text{code}^{i+n+1} \quad \text{ss} \quad \rho' \\
&\text{return}
\end{aligned}
\]

Assumptions:
- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots, R_i \)

Translation of Whole Programs

A program \( P = F_1; \ldots; F_n \) must have a single \texttt{main} function.

\[
\text{code} \quad P \quad \rho = \quad \begin{aligned}
&\text{load} \quad R_i \quad \text{\_main} \\
&\text{mark} \\
&\text{call} \quad R_i \\
&\text{halt} \\
&f_1 : \quad \text{code} \quad F_1 \quad \rho \oplus \rho_{f_1} \\
&\vdots \\
&f_n : \quad \text{code} \quad F_n \quad \rho \oplus \rho_{f_n}
\end{aligned}
\]
Translation of Whole Programs

A program \( P = F_1; \ldots; F_n \) must have a single main function.

\[
\begin{align*}
\text{code}^1 \ P \ p & = \ \text{load} \ R_1 \ \text{main} \\
& \downarrow \text{mark} \\
& \text{call} \ R_1 \\
& \text{halt} \\
\_f_1 : \ & \text{code}^1 \ F_1 \ p \oplus p_f \\
\vdots \\
\_f_n : \ & \text{code}^1 \ F_n \ p \oplus p_f
\end{align*}
\]

Assumptions:
- \( p = 0 \) assuming that we have no global variables
- \( p_f \) contain the addresses the local variables
- \( p_1 \oplus p_2 = \lambda x. \ \{ \ \frac{p_2(x)}{p_1(x)} \ \text{if} \ x \in \text{dom}(p_2) \ \text{otherwise} \)
Register versus Memory

so far:
- all variables are stored in registers
- all function parameters and the return value are stored in registers

limitations:
- a real machine has only a finite number of registers
- in C it is possible to take the address of a variable
- arrays cannot be translated due to indexing

idea: store variables on the stack

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 \( R_i \):

\[
\begin{array}{ll}
code^r_R e \rho & \text{generates code to compute the r-value of } e, \text{ given} \\
& \text{the environment } \rho \\
code^l_L e \rho & \text{analogously for the l-value}
\end{array}
\]

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

Address Environment

A variable can be 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 \( R_j \)

Address Environment

A variable can be 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 \( R_j \)

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 \( R_a \)
Address Environment

A variable can be 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 \( R_g \)

Accordingly, we define \( \rho : \text{Var} \rightarrow \{ G, L, R \} \times Z \) 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 \( \text{FP} + a \)
- \( \rho x = \langle R, a \rangle \): variable \( x \) is stored in register \( R_a \)

Observe: a variable \( x \) can only have one entry in \( \rho \)

However:

- \( \rho \) may be change with the 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:

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

- 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 $\rho(x) = (R_i, j)$ then:

$$\text{code}^R_x e_2 \rho = \text{code}^R_{R_i} e_2 \rho$$

move $R_j R_i$

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

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 $\rho(x) = (R_i, j)$ then:

$$\text{code}^R_x e_2 \rho = \text{code}^R_{R_i} e_2 \rho$$

move $R_j R_i$

Translation of L-Values

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

definition for assignments:

$$\text{code}^L e \rho = \text{code}^L_{e} e \rho$$

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

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 $S[R_i] = R_j$

| $R_i$ | $13$ | $R_j$ |

definition for assignments:

\[
\text{code}_e^i e \rho = \text{code}_e^j e \rho
\]

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

- Thus, for the case $e_1 = x$ and $\rho, x = (R, i)$ does not hold:

\[
\text{code}_e^i e_1 e_2 \rho = \text{code}_e^i e_2 \rho
\]

| $R_i + 1$ | $R_i$ |

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

\[
\text{code}_e^L x \rho = \text{loadc} R_i a
\]

Allocating Memory for Local Variables

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

- alloc $k$
- pop $k$

| $k$ |

- alloc $k$ \quad $\text{SP} = \text{SP} + k$
- pop $k$ \quad $\text{SP} = \text{SP} - k$

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.

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

\[
\text{code}_e^L x \rho = \text{loadc} R_i a \quad \text{if } \rho, x = (L, a)
\]

Instruction loadc $R_i k$ computes the sum of FP and $k$.

| $k$ |

\[
R_i = \text{FP} + k
\]
General Computation of the L-Value of a Variable

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

$$\text{code}_{L,x}^i \rho = \begin{cases} \text{loade} R_i a & \text{if } \rho x = (G, a) \\ \text{loadrc} R_i a & \text{if } \rho x = (L, a) \end{cases}$$

$\text{Note:}$ for $\rho x = (R, \ell)$ the function $\text{code}_{L,x}^i$ 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:

$$\text{code}_{L,x}^i \rho = \begin{cases} \text{loade} R_i a & \text{if } \rho x = (G, a) \\ \text{loadrc} R_i a & \text{if } \rho x = (L, a) \\ \text{load} R_i a & \text{if } \rho x = (R, \ell) \end{cases}$$

$\text{Note:}$ for $\rho x = (R, \ell)$ the function $\text{code}_{L,x}^i$ is not defined!

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** $R_i$ $R_j$; **load** $R_j$ $R_i$; sets $R_j$ to $x$ where $\rho x = (L, a)$.

**In general:** Load variable $x$ into register $R_i$:

$$\text{code}_{R,x}^i \rho = \begin{cases} \text{loade} R_i a & \text{if } \rho x = (G, a) \\ \text{loadrc} R_i a & \text{if } \rho x = (L, a) \\ \text{move} R_i R_j & \text{if } \rho x = (R, \ell) \end{cases}$$

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 load $R_i, R_j$ sets $R_i$ to the value at address $R_j$.

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

In general: Load variable $x$ into register $R_i$:

$$\text{code}_R x \rho = \begin{cases} \text{load} R_i a & \text{if } \rho x = (G, a) \\ \text{loadr} R_i a & \text{if } \rho x = (L, a) \\ \text{move} R_i R_j & \text{if } \rho x = (R, i) \end{cases}$$

Analogously: for write operations we define:

$$\text{storer } a R_j \equiv \begin{cases} \text{loadc} R_i a & \text{store } R_i R_j \\ \text{load} R_i a & \text{store } R_i R_j \end{cases}$$

i.e. $\text{storer } a R_j$ is a macro. Define special case (where $\rho x = (G, a)$):

$$\text{loadc}_K x = e_2 \rho = \begin{cases} \text{loadc}_K e_2 \rho & \text{if } \rho x = (G, a) \\ \text{loadr}_K e_2 \rho & \text{if } \rho x = (L, a) \\ \text{move} R_i R_j & \text{if } \rho x = (R, i) \end{cases} \quad \text{store } R_{i+1} R_i$$

Data Transfer Instructions of the R-CMAs

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

- load $R_i, R_j$: load value from address
- load $R_i, c$: load global variable
- load $R_i, R_j$: load local variable
- store $R_i, R_j$: store value at address
- store $c R_i$: write global variable
- store $c R_i$: write local variable

instructions for computing addresses:

- load $R_i, c$: load constant
- loadc $R_i, R_j$: load constant relative to FP

instructions for general data transfer:

- move $R_i, R_j$: transfer value between registers
- move $R_i, k R_j$: copy $k$ values onto the stack

Determining the Address-Environment

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

$G$ - global variables, defined outside of functions (or as static);
$L$ - local (automatic) variables, defined inside functions, accessible by pointers;
$R$ - register (automatic) variables, defined inside functions.

Example:

```c
int x, y;
void f(int v, int w) {
    int a;
    if (a > 0) {
        int b;
        g(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>$(R, 0)$</td>
</tr>
<tr>
<td>$b$</td>
<td>$(G, 2)$</td>
</tr>
<tr>
<td>$c$</td>
<td>$(R, 2)$</td>
</tr>
</tbody>
</table>
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:

```c
int x, y;
void f(int x, int y) {
    int a;
    if (a > 0) {
        int b;
        g(x, 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\n\nIt\'s\n\nto\n\n!\n"

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

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\n\nIt\'s\n\nto\n\n!\n"

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

Variables in Memory

Chapter 2: Arrays and Pointers
Arrays


- the array a contains 11 elements and therefore requires 11 cells.
- \( \rho a \) is the address of \( a[0] \).

Define the function \( |\cdot| \) to compute the required space of a type:

\[
|t| = \begin{cases} 
\frac{1}{k} \cdot |t'| & \text{if } t \text{ is base type} \\
|t| & \text{if } t = t'[k] 
\end{cases}
\]

For a sequence of declarations \( d \equiv x_1 x_1; \ldots; x_k x_k \); we have:

\[
\begin{align*}
\rho x_1 &= 1 \\
\rho x_i &= \rho x_{i-1} + |x_{i-1}| & \text{for } i > 1
\end{align*}
\]

|\cdot| can be computed at compile time and, hence, \( \rho \) too.

Note: \( |\cdot| \) is required to translate the sizeof operator in C

Translation of Array Accesses

Extend codeL and codeR 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 \( \rho a + |t| \times (R\text{-Wert von } i) \). Thus:

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

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 \( \text{code}_L e \rho = \text{code}_L e \rho \).
- In C the following are equivalent (as \( l \)-value, not as types):
  \[
  2[a] \quad a[2] \quad a + 2
  \]
**C structs (Records)**

Note:
The same field name may occur in different structs.
Here: The component environment $\rho_{st}$ relates to the currently translated structure $st$.
Let $\textbf{struct} \{ \text{int } a; \text{ int } b; \} \ x; \text{ be part of a declaration list.}$
- $x$ is a variable of the size of (at least) the sum of the sizes of its fields.
- We populate $\rho_{st}$ with addresses of fields that are relative to the beginning of $x$, here $a \mapsto 0$, $b \mapsto 1$.

**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 (≡ i-value). Thus define:
  $$\text{code}_k \& e \ \rho = \text{code}_{k+1} e \ \rho$$

Example:
Let $\textbf{struct} \{ \text{int } a; \text{ int } b; \} \ x; \text{ with } \rho = \{ a \mapsto 13 \}$ and $\rho_{st} = \{ a \mapsto 0, b \mapsto 1 \}$.
Then
  $$\text{code}_k (x.b) \ \rho = \text{load } R_1 13$$
  $$\text{load } R_1 1$$
  $$\text{add } R_1, R_i, R_{i+1}$$

**Dereferencing Pointers**

Applying the $*$ operator to an expression $e$ yields the content of the cell whose l-value is stored in $e$:
  $$\text{code}_k \* e \ \rho = \text{code}_{k+1} e \ \rho$$

Example: Consider

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

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

Since $e \rightarrow a \equiv (\*e).a$ we get:
  $$\text{code}_k (e \rightarrow a) \ \rho = \text{code}_{k+1} e \ \rho$$
  $$\text{load } R_1 R_i R_{i+1}$$
Translation of Array Accesses

Extend $\text{code}_{e}$ and $\text{code}_{e'}$ with indexed array accesses.

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

$$\text{code}_{e}^{t[a]_i} \rho = \text{code}_{e'}^{t[a]_i} \rho \quad \text{where} \quad a[i] \equiv a$$

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 $\text{code}_{e}^{t[a]_i} \rho$
- In C, the following are equivalent (as l-value, not as types):
  $$2[a] \quad a[2] \quad a+2$$

C structs (Records)

Note:
The same field name may occur in different structs.
Here: The component environment $\rho_{\alpha}$ 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 $\rho_{\alpha}$ with addresses of fields that are relative to the beginning of $x$, here $a \rightarrow 0$, $b \rightarrow 1$.

In general, let $i \equiv \text{struct} \{ t_1 v_1; \ldots ; t_k v_k \}$, then

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

We obtain:

$$\text{code}_{e}^{(e.c)} \rho = \text{code}_{e'}^{(e.c)} \rho \quad \text{load } R_{\alpha + 1} (\rho_{\alpha} c) \quad \text{add } R_i R_i R_{i+1}$$

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

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

$$\text{code}_{e}^{(e.c)} \rho = \text{code}_{e'}^{(e.c)} \rho \quad \text{move } R_{\alpha + 1} R_{i+1} \quad e \text{ a structure of size } k$$
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

\[
\text{new instruction: } \text{move} \quad R_i k \quad R_{i+1} \quad e \quad \text{a structure of size } k
\]

Invariant of Heap and Stack

- the stack and the heap may not overlap

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:

\[
\text{code}_k \ e \rho = \text{code}_{k+1} \ e \rho
\]

⇒ 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.

Translation of Programs

Before the execution of a program, the runtime sets:

\[
SP = -1 \quad FP = \text{EP} = 0 \quad PC = 0 \quad NP = \text{MAX}
\]

Let \( p = V \_\text{defs} \quad F_\text{-def}_1 \ldots F_\text{-def}_n \) be a program where \( F_\text{-def}_i \) defines a function \( f_i \) of which one is called \( \text{main} \).

The code for the program \( p \) is comprised of:

- code for each function definition \( F_\text{-def}_i \);
- code to initialize global variables
- code that calls \( \text{main}() \)
- an instruction \( \text{halt} \).
Instructions for Starting a Program

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

```
code^1 P \rho = enter (k + 3)
alloc k
load R_1_main
saveloc R_1 R_0
mark
call R_1
resaveloc R_1 R_0
halt
```

```
\_f_1: code^1 F_1 \rho \oplus \rho_\mathbb{F}
\vdots
\_f_n: code^1 F_n \rho \oplus \rho_\mathbb{F}
```

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:

```
code^1 t_r \x{L}(args)\{decls ss\} \rho = enter q
alloc k
move R_{i+1} R_{-1}
\vdots
move R_{i+n} R_{-n}
code^{i+n+1} ss \rho'
return
```

Randbedingungen:

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

Note: def-use liveness

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

Outlook

- register allocation has several other uses:
  - remove unnecessary move instructions