Simple Conditional
We first consider \( s \equiv \text{if } (c) \text{ ss} \) ...and present a translation without basic blocks.

Idea:
- emit the code of \( c \) and \( \text{ss} \) in sequence
- insert a jump instruction in-between, so that correct control flow is ensured

\[
\text{code}^t \ s \ \rho = \begin{cases} 
\text{code}^t_2 \ c \ \rho \\
\text{jumpz} \ \rho, \ A \\
\text{code}^t_3 \ \text{ss} \ \rho \\
\text{code}^t \ \text{for ss}
\end{cases}
\]

Example for if-statement
Let \( \rho = \{ x \mapsto 4, y \mapsto 7 \} \) and let \( s \) be the statement

\[
\begin{align*}
\text{if } &\quad x > y; \quad \text{/* (i) */} \\
&\quad x \leftarrow x - y; \quad \text{/* (ii) */} \\
\text{else } &\quad y \leftarrow y - x; \quad \text{/* (iii) */}
\end{align*}
\]

Then \( \text{code}^t \ s \ \rho \) yields:
Example for if-statement

Let $\rho = \{ x \mapsto 4, y \mapsto 7 \}$ and let $s$ be the statement:

```plaintext
if (x > y) { /* (i) */
    x = x - y; /* (iii) */
} else {
    y = y - x; /* (iii) */
}
```

Then $\text{code}^i s \rho$ yields:

- (i) move $R_i R_4$
- (iii) move $R_i R_7$
- move $R_{i+1} R_7$
- move $R_{i+1} R_{i+1}$
- sub $R_i R_i R_{i+1}$
- move $R_i R_{i+1}$
- move $R_{i+1}$
- move $R_i R_i$
- move $R_{i+1}$
- jump $B$

Example: Translation of Loops

Let $\rho = \{ a \mapsto 7, b \mapsto 8, c \mapsto 9 \}$ and let $s$ be the statement:

```plaintext
while (a > 0) { /* (i) */
    c = c + 1; /* (ii) */
    a = a - b; /* (iii) */
}
```

Then $\text{code}^i s \rho$ evaluates to:

- (i) move $R_i R_7$
- load $R_{i+1} 0$
- gr $R_i R_i R_{i+1}$
- move $R_i R_0$
- move $R_i R_9$
- load $R_{i+1} 1$
- add $R_i R_i R_{i+1}$
- sub $R_i R_i R_{i+1}$
- move $R_i R_i$
- move $R_{i+1} R_8$
- move $R_i R_7$
- jump $B$
- jump $A$

Iterating Statements

We only consider the loop $s \equiv \text{while}(e) s'$. For this statement we define:

- code$_A$ while $s \rho = A$
- code$_e$ for $e$
- code$_{s'}$ for $s'$

Example: Translation of Loops

Let $\rho = \{ a \mapsto 7, b \mapsto 8, c \mapsto 9 \}$ and let $s$ be the statement:

```plaintext
while (a > 0) { /* (i) */
    c = c + 1; /* (ii) */
    a = a - b; /* (iii) */
}
```

Then $\text{code}^i s \rho$ evaluates to:

- (i) move $R_i R_7$
- load $R_{i+1} 0$
- gr $R_i R_i R_{i+1}$
- move $R_i R_0$
- move $R_i R_9$
- load $R_{i+1} 1$
- add $R_i R_i R_{i+1}$
- sub $R_i R_i R_{i+1}$
- move $R_i R_i$
- move $R_{i+1} R_8$
- move $R_i R_7$
- jump $B$
- jump $A$
for-Loops

The *for*-loop $s \equiv (e_1; e_2; e_3)$ $s'$ is equivalent to the statement sequence $e_1$; while $(e_2) \{s'; e_3; \}$ — as long as $s'$ does not contain a `continue` statement.

Thus, we translate:

```
\text{code}_1 \text{for}(e_1; e_2; e_3) \ s \ \rho = \ \text{code}_2 \ e_1 \ \rho \\
A : \ \text{code}_2 \ e_2 \ \rho \\
jumpi \ R_i \ B \\
\text{code}_3 \ s \ \rho \\
\text{code}_2 \ e_3 \ \rho \\
jumpi \ A \\
B : 
```

The switch-Statement

**Idea:**

- Suppose choosing from multiple options in *constant time* if possible
- use a *jump table* that, at the $i$th position, holds a jump to the $i$th alternative
- in order to realize this idea, we need an *indirect jump* instruction

```
switch (c) {
    case 0: s_0; break;
    ...;
    case k-1: s_{k-1}; break;
    default: s_k; break;
}
```

Consecutive Alternatives

Let `switch` $s$ be given with $k$ consecutive case alternatives:

```
PC = A + R_i;
```
Consecutive Alternatives

Let `switch s` be given with `k` consecutive `case` alternatives:

```java
switch (e) {
    case 0: s₀; break;
    ...
    case k-1: s_{k-1}; break;
    default: s_k; break;
}
```

Define `code^i s ρ` as follows:

```latex
\text{code}^i s \rho = \left\{ \begin{array}{ll}
\text{code}^i 0 \ B & B: \text{jump } A_0 \\
\text{check}^i 0 \ k \ B & A_0: \text{code}^i s_0 \rho \\
\vdots & \vdots \\
\text{check}^i k \ B & A_k: \text{code}^i s_k \rho \\
\text{jump } C & \vdots \\
\text{jump } C & \vdots \\
\end{array} \right.
```

Translation of the `check^i` Macro

The macro `check^i l u B` checks if `l ≤ R_i < u`. Let `k = u - l`.

- if `l ≤ R_i < u` it jumps to `B + R_i - l`
- if `R_i < l` or `R_i ≥ u` it jumps to `A_k`

```latex
\text{B: jump } A_0 \\
\vdots \\
\text{jump } A_k
```

Translation of the `check^i` Macro

The macro `check^i l u B` checks if `l ≤ R_i < u`. Let `k = u - l`.

- if `l ≤ R_i < u` it jumps to `B + R_i - l`
- if `R_i < l` or `R_i ≥ u` it jumps to `A_k`

we define:

```latex
\text{check}^i l u B = \left\{ \begin{array}{ll}
\text{load } R_{i+1} l & \vdots \\
\text{get } R_{i+2} R_i R_{i+1} & B: \text{jump } A_0 \\
\text{jumpz } R_{i+2} E & A_0: \text{code}^i s_0 \rho \\
\text{sub } R_i R_i R_{i+1} & \vdots \\
\text{load } R_{i+1} u & \vdots \\
\text{get } R_{i+2} R_i R_{i+1} & \text{jump } A_k \\
\text{jumpz } R_{i+2} D & \vdots \\
\text{load } R_i u - l & \text{jump } A_k \\
\text{jump } B & \vdots \\
\end{array} \right.
```
Improvements for Jump Tables

This translation is only suitable for certain switch-statements.
- In case the table starts with 0 instead of u we don’t need to subtract it from c before we use it as index
- if the value of c is guaranteed to be in the interval [l, u], we can omit check

General translation of switch-Statements

In general, the values of the various cases may be far apart:
- generate an if-ladder, that is, a sequence of if-statements
- for n cases, an if-cascade (tree of conditionals) can be generated $\sim O(\log n)$ tests
- if the sequence of numbers has small gaps ($\leq 3$), a jump table may be smaller and faster
- one could generate several jump tables, one for each sets of consecutive cases
- an if cascade can be re-arranged by using information from profiling, so that paths executed more frequently require fewer tests

Ingredients of a Function

The definition of a function consists of
- a name with which it can be called;
- a specification of its formal parameters;
- possibly a result type;
- a sequence of statements.

In C we have:

```
code: $f$  $\rho$ = load $R_{\_f}$ with $\_f$ starting address of $f$
```

Observe:
- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later

Memory Management in Functions

```c
int fac(int x) {
    if (x<=0) return 1;
    else return x*fac(x-1);  
}
```
```c
int main(void) {
    int n;
    n = fac(2) + fac(1);
    printf("%d", n);
}
```

At run-time several instances may be active, that is, the function has been called but has not yet returned.

The recursion tree in the example:
Memory Management in Function Variables

The formal parameters and the local variables of the various instances of a function must be kept separate.

Idea for implementing functions:
- set up a region of memory each time it is called
- in sequential programs this memory region can be allocated on the stack
- thus, each instance of a function has its own region on the stack
- these regions are called stack frames

Organization of a Stack Frame

- stack representation: grows upwards
- SP points to the last used stack cell

Split of Obligations

Definition
Let \( f \) be the current function that calls a function \( g \).
- \( f \) is dubbed caller
- \( g \) is dubbed callee

The code for managing function calls has to be split between caller and callee. This split cannot be done arbitrarily since some information is only known in that caller or only in the callee.

Observation:
The space requirement for parameters is only known by the caller.
Example: printf

Principle of Function Call and Return

actions taken on entering \( g \):
1. compute the start address of \( g \)
2. compute actual parameters in globals
3. backup of caller-save registers
4. backup of FP
5. set the new FP
6. back up of PC and jump to the beginning of \( g \)
7. copy actual params to locals

... is in \( g \)

Actions taken on leaving \( g \):
1. compute the result into \( R_0 \)
2. restore FP, SP
3. return to the call site in \( f \), that is, restore PC
4. restore the caller-save registers

... is in \( f \)
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 live in 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 $R_i$, and $g$ must restore it before returning

Translation of Function Calls

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

\[
\text{code}^R_g(e_1, \ldots, e_n) = \text{code}^{R_i}_R g \\quad \text{code}^{R_{-1}}_R e_1 \quad \text{code}^{R_{-i+n}}_R e_n
\]

\[
\begin{align*}
\text{move} & \quad R_{-1} R_{i+1} \\
\text{move} & \quad R_{-n} R_{i+n} \\
\text{saveloc} & \quad R_i R_{i-1} \\
\text{mark} & \\
\text{call} & \quad R_i \\
\text{restoreloc} & \quad R_{i-1} R_{i-1} \\
\text{move} & \quad R_i R_0
\end{align*}
\]

Calling a Function

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

\[
\begin{align*}
\text{SP} & = \text{SP} + 1; \\
\text{S[SP]} & = \text{PC}; \\
\text{FP} & = \text{SP}; \\
\text{PC} & = R_i;
\end{align*}
\]
Result of a Function

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

\[
\text{code}^i \text{return } e \rho = \begin{cases} 
\text{move } R_0 R_4 \\
\text{return}
\end{cases}
\]

alternative without result value:

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

Return from a Function

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

Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^i \ t_r \{\text{args}\} \{\text{decls} \ ss\} \rho = \begin{cases} 
\text{move } R_{i+1} R_{i-1} \\
\vdots \\
\text{move } R_{i+n} R_{i-1} \\
\text{code}_{i+n-1} \ ss \ \rho' \\
\text{return}
\end{cases}
\]

Assumptions:

\[
\begin{align*}
\text{PC} &= S[FP] \\
\text{SP} &= \text{FP-2} \\
\text{FP} &= S[\text{SP+1}]
\end{align*}
\]
Translation of Whole Programs

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

code\(^1\) $P \rho = \frac{\text{loadc } R_1 \_\text{main}}{\text{mark}}$

\[ f_1: \text{code} P \rho \oplus \rho_{f_1}, \]

\[ f_n: \text{code} P \rho \oplus \rho_{f_n}. \]

Translation of the fac-function

Consider:

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

A: move $R_2 R_3$

B: $x \cdot \text{fac}(x - 1)$

\[ i = 3 \]

\[ \text{move } R_3 R_1 \]

\[ \text{load } R_4 1 \]

\[ \text{sub } R_2 R_3 R_4 \]

\[ \text{mov } R_4 R_1 \]

\[ \text{save } \text{param.} \]

\[ i = 4 \]

\[ \text{divide } R_2 R_1 \]

\[ \text{load } R_2 0 \]

\[ \text{loq } R_2 R_3 R_4 \]

\[ \text{jump } R_2 \_A \]

\[ \text{load } R_2 1 \]

\[ \text{move } R_0 R_2 \]

\[ \text{return } \]

\[ \text{jump } \_B \]

\[ \text{code is dead} \]

\[ \text{return } \]

\[ x \cdot \ldots \]

End of presentation. Click to exit.