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

Consecutive Alternatives
Let switch s be given with k consecutive case alternatives:
```java
switch (e) {
    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:

```c
switch (s) {
    case 0: s0; break;
    ...
    case k-1: sk-1; break;
    default: sk; break;
}
```

Define $\text{code}^i s \rho$ as follows:

$$\text{code}^i s \rho = \text{code}_k^i e \rho$$

$$\text{check}^i 0 k B$$

$$A_0 : \text{code}^i s_0 \rho$$
```
  B : jump A_0
  ...
  C : jump A_k
```

$$A_k : \text{code}^i s_k \rho$$
```
  C :
```

**Translation of the $\text{check}^i$ Macro**

The macro $\text{check}^i l u B$ checks if $l \leq R_i < u$. Let $k = u - l$.

- if $l \leq R_i < u$ it jumps to $B + R_i - l$
- if $R_i < l$ or $R_i \geq u$ it jumps to $B + n = B + k$

```
B : jump A_0
  ...
  C : jump A_k
```

**Translation of the $\text{check}^i$ Macro**

The macro $\text{check}^i l u B$ checks if $l \leq R_i < u$. Let $k = u - l$.

- if $l \leq R_i < u$ it jumps to $B + R_i - l$
- if $R_i < l$ or $R_i \geq u$ it jumps to $C$

we define:

```
\begin{align*}
\text{check}^0 l u B &= \text{load} R_i+1 l \\
&\quad \text{geq} R_i+2 R_i R_i+1 \\
&\quad \text{jumpz} R_i+2 B \\
&\quad \text{sub} R_i R_i R_i+1 \\
&\quad \text{load} R_i+1 u \\
&\quad \text{geq} R_i+2 R_i R_i+1 \\
&\quad \text{jump} A_k \\
\end{align*}
```

```
```
Translation of the \texttt{check} Macro

The macro \texttt{check} \( l \ u B \) checks if \( l \leq R_i < u \). Let \( k = u - l \).
- if \( l \leq R_i < u \) it jumps to \( B + R_i - l \)
- if \( R_i < l \) or \( R_i \geq u \) it jumps to \( C \)
we define:

\[ \begin{align*}
\text{check} & \quad l \ u B \quad = \quad \text{loadc } R_{i+1} \ 0 \\
& \quad \text{geq } R_{i+2} R_i R_{i+1} \\
& \quad \text{jumpz } R_{i+2} E \\
& \quad \text{sub } R_i R_i R_{i+1} \\
& \quad \text{loadc } R_{i+1} u \\
& \quad \text{geq } R_{i+2} R_i R_{i+1} \quad \text{jump } A_k \\
& \quad \text{jumpz } R_{i+2} D \\
& \quad \text{loadc } R_i (i - C) \\
& \quad \text{jump } B
\end{align*} \]

Improvements for Jump Tables

This translation is only suitable for certain \texttt{switch} statement.
- 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 \texttt{check} completely

General translation of switch-Statements

In general, the values of the various cases may be far apart:
- generate an \texttt{if}-ladder, that is, a sequence of \texttt{if}-statements
- for \( n \) cases, an \texttt{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 \texttt{if} cascade can be re-arranged by using information from profiling, so that paths executed more frequently require fewer tests

Chapter 4:
Functions
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:

\[
\text{code: } f_p \rightarrow \text{loadc } R_i.f
\]
with \( f_p \) 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:

```
      main
        /|
       / |
      fac fac
       / |
      fac fac
```

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

```
SP
```

```
FP
```

```
PCold
```

```
FPold
```

local memory
callee

organizational
cells
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
\texttt{f(42, \ldots);}

 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 **callee**-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

actions taken on leaving $g$:

1. compute the result into $R_9$
2. restore FP, SP
3. return to the call site in $f$, that is, restore PC
4. restore the **callee**-save registers

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**:

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:
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 live in \textit{global} registers $R_i$ (with $i \leq 0$)
- global variables: let’s suppose there are none

\[ R_i \]

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

Translation of Function Calls

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

\[
\begin{align*}
\text{code}_R^g & g(e_1, \ldots, e_n) \rho = \\
\text{code}_R^g & g \rho \\
\text{code}_R^g & e_1 \rho \\
\vdots & \\
\text{code}_R^g & e_n \rho \\
\text{move} & R_{i+1} R_i \\
\vdots & \\
\text{move} & R_{i+n} R_i \\
\text{savclrc} & R_i R_{i-1} \\
\text{mark} & \\
\text{call} & R_i \\
\text{restorc} & R_i R_{i-1} \\
\text{move} & R_i R_0
\end{align*}
\]

New instructions:
- \text{savclrc} $R_i R_j$ pushes the registers $R_i$, $R_{i+1} \ldots R_j$ onto the stack
- \text{mark} backs up the organizational cells
- \text{call} $R_i$ calls the function at the address in $R_i$
- \text{restorc} $R_i$ $R_j$ pops $R_j$, $R_{j-1}, \ldots R_i$ off the stack

Definition

Let $f$ be a function that calls $g$. A register $R_i$ is called
- \textit{caller-saved} if $f$ backs up $R_i$ and $g$ may overwrite it
- \textit{callee-saved} if $f$ does not back up $R_i$, and $g$ must restore it before returning
Rescuing EP and FP

The instruction **mark** allocates stack space for the return value and the organizational cells and backs up FP.

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

Calling a Function

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

\[ SP = SP + 1; \]
\[ S[SP] = PC; \]
\[ FP = SP; \]
\[ PC = Ri; \]

Result of a Function

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

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

Result of a Function

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

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

alternative without result value:

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

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

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

move \( R_0 \) \( R_i \)


alternative without result value:

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

\textit{global} registers are otherwise not used inside a function body:

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

Translation of Functions

The translation of a function is thus defined as follows:

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

move \( R_{i+1} \) \( R_{-1} \)

\[
\vdots
\]

move \( R_{i+n} \) \( R_{-n} \)

\[
\text{code}^l t_{i+n+1} \{ \text{ss } \rho' \}
\]

\text{return}

Assumptions:

- the function has \( n \) parameters

Return from a Function

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

\[
\begin{align*}
\text{PC} &= S[\text{FP}] ; \\
\text{EP} &= S[\text{FP} - 2] ; \\
\text{SP} &= \text{FP} - 3 ; \\
\text{FP} &= S[\text{SP} + 2] ;
\end{align*}
\]

Translation of Functions

The translation of a function is thus defined as follows:

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

move \( R_{i+1} \) \( R_{-1} \)

\[
\vdots
\]

move \( R_{i+n} \) \( R_{-n} \)

\[
\text{code}^l t_{i+n+1} \{ \text{ss } \rho' \}
\]

\text{return}

Assumptions:

- the function has \( n \) parameters
Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^1 \ 1 \ 1 \ 1 \ 1 \ {\text{t}.} \ (a) \ {\text{f.}} \ (a) \ (\text{args}) \ {\text{decls}} \ \{ss\} \ \rho = \begin{cases} & \text{enter } q \\
& \text{move } R_{t+1} \ R_{t-1} \\
& \vdots \\
& \text{move } R_{t+n} \ R_{n-1} \\
& \text{code}^1 \ {\text{f.}} \ (a) \ {\text{decls}} \ \{ss\} \ \rho' \\
& \text{return} \end{cases}
\]

Assumptions:
- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots, R_n \)
- the parameters of the function are in \( R_{-1}, \ldots, R_{-n} \)
- \( \rho' \) is obtained by extending \( \rho \) with the bindings in \( \text{decls} \) and the function parameters \( \text{args} \)

Translation of the \( \text{fac} \)-function

Consider:

\[
\text{int} \ \text{fac} (\text{int} \ x) \ {\text{f.}} \ {\text{if}} \ (x \leq 0) \ {\text{then}} \\
\quad \text{return} \ 1; \\
\text{else} \\
\quad \text{return} \ x \ \text{fac} (x - 1); \\
\]

\[
\text{fac}: \begin{cases} & \text{enter} \\
& \text{move } R_1 \ R_{-1} \\
& \text{save param.} \\
& i = 2 \\
& \text{move } R_2 \ R_1 \\
& \text{load } R_3 \ 0 \\
& \text{leq } R_2 \ R_2 \ R_3 \\
& \text{jumpz } R_2 \ \text{-A} \\
& \text{load } R_0 \ 1 \\
& \text{move } R_0 \ R_2 \\
& \text{return} \\
& \text{jump } \text{-B} \end{cases}
\]

\[
\begin{align*}
\text{A:} & \quad \text{move } R_2 \ R_1 \\
& \quad x \ \text{fac} (x - 1) \\
& \quad i = 3 \\
& \quad \text{move } R_3 \ R_1 \\
& \quad \text{x -1} \\
& \quad i = 4 \\
& \quad \text{load } R_4 \ 1 \\
& \quad \text{sub } R_5 \ R_3 \ R_4 \\
& \quad \text{move } R_1 \ R_3 \ \text{fac} (x - 1) \\
& \quad \text{load } R_3 \ \text{fac} \\
& \quad \text{saveloc } R_1 \ R_2 \\
& \quad \text{mark} \\
& \quad \text{call } R_3 \\
& \quad \text{restoreloc } R_3 \ R_2 \\
& \quad \text{move } R_3 \ R_5 \\
& \quad \text{mul } R_2 \ R_2 \ R_5 \\
& \quad \text{move } R_0 \ R_2 \\
& \quad \text{return} \\
\end{align*}
\]

\[
\begin{align*}
\text{B:} & \quad \text{code is dead}
\end{align*}
\]

Translation of Whole Programs

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

\[
\text{code}^1 P \ \rho = \begin{cases} & \text{load } R_1 \ \text{main} \\
& \text{mark} \\
& \text{call } R_3 \\
& \text{halt} \\
& \_f_1: \text{code}^1 F_1 \ \rho \oplus \ \rho_{f_1} \\
& \quad \vdots \\
& \_f_n: \text{code}^1 F_n \ \rho \oplus \ \rho_{f_n}
\end{cases}
\]

End of presentation. Click to exit.