Script generated by TTT

Title: Simon: Compilerbau (08.07.2013)

Date: Mon Jul 08 14:15:26 CEST 2013

Duration: 89:31 min

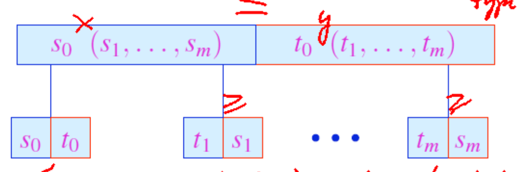
Pages: 62

Rules and Examples for Subtyping

int x; float y;

type(x) ≤ type(y)

y = x; correct



int (*x)(int); float (*y)(float);

y = x; float r = *y(0.7)

Examples:

struct {**int** a; **int** b; }	$\leq$	**struct** {**float** a; }
int (**int**)	$\not\leq$	**float** (**float**)
int (**float**)	$\leq$	**float** (**int**)

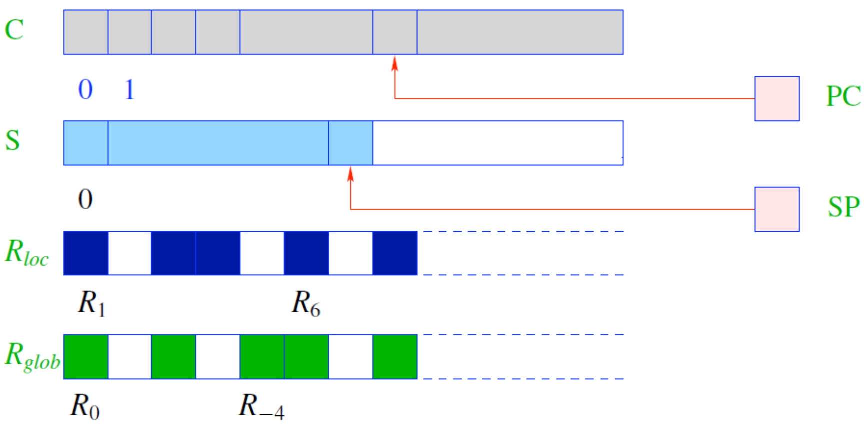
Attention:

- For functions:
- the return types are in normal subtype relationship
- for argument types, the subtype relation reverses

Principle of the Register C-Machine

The R-CMa is composed of a stack, heap and a code segment, just like the JVM; it additionally has register sets:

- *local* registers are $R_1, R_2, \ldots R_i, \ldots$
- *global* register are $R_0, R_{-1}, \ldots R_j, \ldots$



The Register Sets of the R-CMa

The two register sets have the following purpose:

1. the *local* registers R_i
   - save temporary results
   - store the contents of local variables of a function
   - can efficiently be stored and restored from the stack
2. the *global* registers R_i
   - save the parameters of a function
   - store the result of a function

Translation of Simple Expressions

Using variables stored in registers; loading constants:

instruction	semantics	intuition
loadc R_i c	$R_i = c$	load constant
move R_i R_j	$R_i = R_j$	copy R_j to R_i

We define the following translation schema (with $\rho\, x = a$):

$$\begin{aligned} \text{code}^i_R\ c\ \rho &= \text{loadc}\ R_i\ c \\ \text{code}^i_R\ x\ \rho &= \text{move}\ R_i\ R_a \\ \text{code}^i_R\ x = e\ \rho &= \text{code}^i_R\ e\ \rho \\ &\quad\ \text{move}\ R_a\ R_i \end{aligned}$$

Translation of Simple Expressions

Using variables stored in registers; loading constants:

instruction	semantics	intuition
loadc R_i c	$R_i = c$	load constant
move R_i R_j	$R_i = R_j$	copy R_j to R_i

We define the following translation schema (with $\rho\, x = a$):

$$\begin{aligned} \text{code}^i_R\ c\ \rho &= \text{loadc}\ R_i\ c \\ \text{code}^i_R\ x\ \rho &= \text{move}\ R_i\ R_a \\ \text{code}^i_R\ x = e\ \rho &= \text{code}^i_R\ e\ \rho \\ &\quad\ \text{move}\ R_a\ R_i \end{aligned}$$

Note: all instructions use the Intel convention (in contrast to the AT&T convention): op $dst\ src_1\ \ldots src_n$.

Translation of Expressions

Let op $= \{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or\}$. The R-CMa provides an instruction for each operator op.

$$\text{op}\ \ R_i\ R_j\ R_k$$

where R_i is the target register, R_j the first and R_k the second argument.

Correspondingly, we generate code as follows:

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

i

i+1

i

Translation of Expressions

Let op $= \{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or\}$. The R-CMa provides an instruction for each operator op.

$$\text{op}\ \ R_i\ R_j\ R_k$$

where R_i is the target register, R_j the first and R_k the second argument.

Correspondingly, we generate code as follows:

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

Example: Translate $3 \star 4$ with $i = 4$:

$$\begin{aligned} \text{code}^4_R\ \ 3 \star 4\ \ \rho &= \text{code}^4_R\ \ 3\ \ \rho \\ &\quad\ \text{code}^5_R\ \ 4\ \ \rho \\ &\quad\ \text{mul}\ R_4\ R_4\ R_5 \end{aligned}$$

Applying Translation Schema for Expressions

Suppose the following function is given:

```
void f(void) {
  int x,y,z;
  x = y+z*3;
}
```

- Let $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ be the address environment.
- Let R_4 be the first free register, that is, $i = 4$.

$$\begin{aligned}
\text{code}^4\ \texttt{x=y+z*3}\ \rho &= \text{code}_R^4\ \texttt{y+z*3}\ \rho \\
&\quad \text{move } R_1\ R_4 \\
\text{code}_R^4\ \texttt{y+z*3}\ \rho &= \text{move } R_4\ R_2 \\
&\quad \text{code}_R^5\ \texttt{z*3}\ \rho \\
&\quad \text{add } R_4\ R_4\ R_5
\end{aligned}$$

if() {

};

x = 7;

if() {

}

About Statements and Expressions

General idea for translation:

$\text{code}^i\ s\ \rho$: generate code for statement s

$\text{code}_R^i\ e\ \rho$: generate code for expression e into R_i

Throughout: $i, i+1, \ldots$ are free (unused) registers

For an *expression* $x = e$ with $\rho\ x = a$ we defined:

$$\begin{aligned}
\text{code}_R^i\ x = e\ \rho &= \text{code}_R^i\ e\ \rho \\
&\quad \text{move } R_a\ R_i
\end{aligned}$$

However, $x = e$ is also a *statement*:

- Define:

$$\text{code}^i\ e_1 = e_2\ \rho = \text{code}_R^i\ e_1 = e_2\ \rho$$

The temporary register R_i is ignored here. More general:

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

About Statements and Expressions

General idea for translation:

$\text{code}^i\ s\ \rho$: generate code for statement s

$\text{code}_R^i\ e\ \rho$: generate code for expression e into R_i

Throughout: $i, i+1, \ldots$ are free (unused) registers

For an *expression* $x = e$ with $\rho\ x = a$ we defined:

$$\begin{aligned}
\text{code}_R^i\ x = e\ \rho &= \text{code}_R^i\ e\ \rho \\
&\quad \text{move } R_a\ R_i
\end{aligned}$$

However, $x = e$ is also a *statement*:

if $(e_1 = e_2)$ {

}

- Define:

$$\text{code}^i\ e_1 = e_2\ \rho = \text{code}_R^i\ e_1 = e_2\ \rho$$

The temporary register R_i is ignored here. More general:

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

- Observation: the assignment to e_1 is a side effect of the evaluating the expression $e_1 = e_2$.

Translation of Statement Sequences

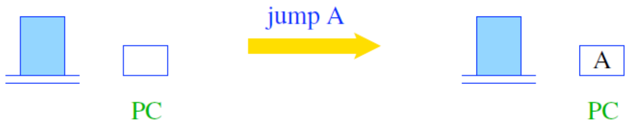
The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

$$\begin{aligned}
\text{code}^i\ (s\ ss)\ \rho &= \text{code}^i\ s\ \rho \\
&\quad \text{code}^i\ ss\ \rho \\
\text{code}^i\ \varepsilon\ \rho &= \quad //\ \textit{empty sequence of instructions}
\end{aligned}$$

Note here: s is a statement, ss is a sequence of statements

Jumps

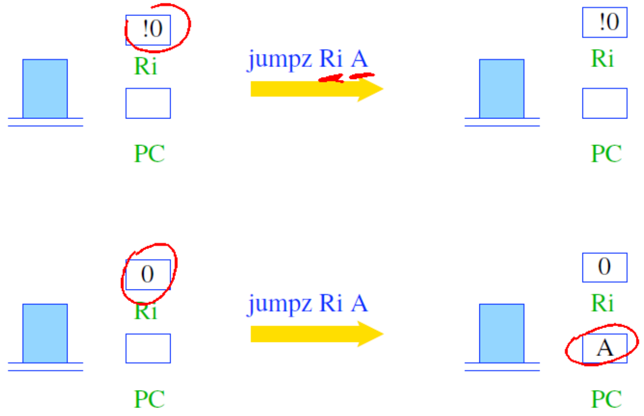
In order to diverge from the linear sequence of execution, we need *jumps*:



PC = A;

Conditional Jumps

A conditional jump branches depending on the value in R_i:



if (R_i == 0) PC = A;

Management of Control Flow

In order to translate statements with control flow, we need to emit jump instructions.

- during the translation of an **if** `(c)` construct, it is not yet clear where to jump to in case that `c` is false

Management of Control Flow

In order to translate statements with control flow, we need to emit jump instructions.

- during the translation of an **if** `(c)` construct, it is not yet clear where to jump to in case that `c` is false
- instruction sequences may be arranged in a different order
  - minimize the number of *unconditional* jumps
  - minimize in a way so that fewer jumps are executed inside loops
  - replace *far jumps* through *near jumps* (if applicable)

Management of Control Flow

In order to translate statements with control flow, we need to emit jump instructions.

- during the translation of an **`if (c)`** construct, it is not yet clear where to jump to in case that `c` is false
- instruction sequences may be arranged in a different order
  - minimize the number of *unconditional* jumps
  - minimize in a way so that fewer jumps are executed inside loops
  - replace *far jumps* through *near jumps* (if applicable)
- organize instruction sequence into blocks without jumps

Management of Control Flow

In order to translate statements with control flow, we need to emit jump instructions.

- during the translation of an **`if (c)`** construct, it is not yet clear where to jump to in case that `c` is false
- instruction sequences may be arranged in a different order
  - minimize the number of *unconditional* jumps
  - minimize in a way so that fewer jumps are executed inside loops
  - replace *far jumps* through *near jumps* (if applicable)
- organize instruction sequence into blocks without jumps

To this end, we define:

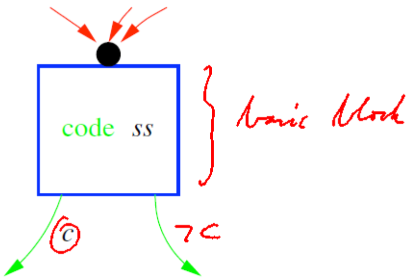
Definition

A basic block consists of

- a sequence of statements ss that does not contain a jump
- a set of outgoing edges to other basic blocks
- where each edge may be labelled with a condition

Basic Blocks and the Register C-Machine

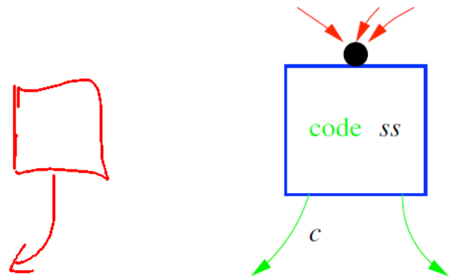
The R-CMa features only a single conditional jump, namely jumpz.



Outgoing edges must have the following form:

Basic Blocks and the Register C-Machine

The R-CMa features only a single conditional jump, namely jumpz.

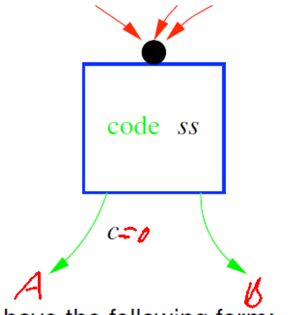


Outgoing edges must have the following form:

1. a single edge (unconditional jump), translated with jump

Basic Blocks and the Register C-Machine

The R-CMa features only a single conditional jump, namely jumpz.



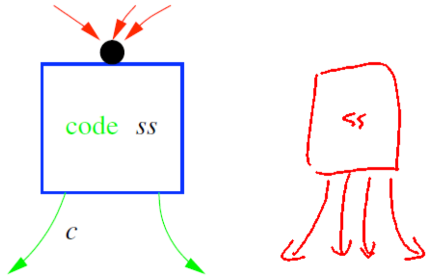
jumpz c A
jump B

Outgoing edges must have the following form:

1. a single edge (unconditional jump), translated with jump
2. two edges, one with $c = 0$ as condition and one without condition, translated with jumpz and jump, respectively

Basic Blocks and the Register C-Machine

The R-CMa features only a single conditional jump, namely jumpz.



Outgoing edges must have the following form:

1. a single edge (unconditional jump), translated with jump
2. two edges, one with $c = 0$ as condition and one without condition, translated with jumpz and jump, respectively
3. a set of edges and one default edge, used for **`switch`** statement, translated with jumpi and jump (to be discussed later)

Formalizing the Translation Involving Control Flow

For simplicity of defining translations of instructions involving control flow, we use *symbolic jump targets*.

- This translation can be used in practice, but a second run through the emitted instructions is necessary to *resolve* the symbolic addresses to actual addresses.

jump A

A: mov

Formalizing the Translation Involving Control Flow

For simplicity of defining translations of instructions involving control flow, we use *symbolic jump targets*.

- This translation can be used in practice, but a second run through the emitted instructions is necessary to *resolve* the symbolic addresses to actual addresses.

Alternatively, we can emit *relative* jumps without a second pass:

- relative jumps have targets that are offsets to the current PC
- sometime relative jumps only possible for small offsets ($\leadsto$ near jumps)
- if all jumps are relative: the code becomes position independent (PIC), that is, it can be moved to a different address
- the generated code can be loaded without relocating absolute jumps

Formalizing the Translation Involving Control Flow

For simplicity of defining translations of instructions involving control flow, we use *symbolic jump targets*.

- This translation can be used in practice, but a second run through the emitted instructions is necessary to *resolve* the symbolic addresses to actual addresses.

Alternatively, we can emit *relative* jumps without a second pass:

- relative jumps have targets that are offsets to the current PC
- sometime relative jumps only possible for small offsets (⇝ near jumps)
- if all jumps are relative: the code becomes position independent (PIC), that is, it can be moved to a different address
- the generated code can be loaded without relocating absolute jumps

generating a graph of basic blocks is useful for *program optimization* where the statements inside basic blocks are simplified

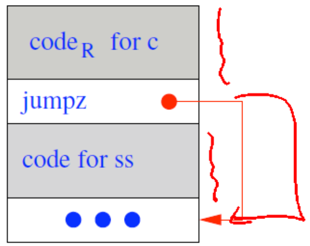
Simple Conditional

We first consider $s \equiv$ `if ( c )` ss.
...and present a translation without basic blocks.

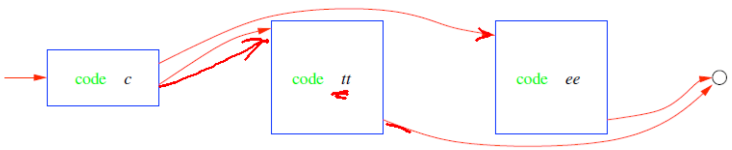
Idea:

- emit the code of c and ss in sequence
- insert a jump instruction in-between, so that correct control flow is ensured

$$\begin{array}{rrl} \text{code}^i\ s\ \rho & = & \text{code}^i_R\ c\ \rho \\ & & \text{jumpz}\ R_i\ A \\ & & \text{code}^i\ ss\ \rho \\ A: & & \ldots \end{array}$$

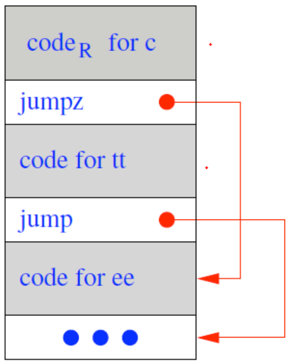


General Conditional



Translation of `if ( c )` tt **else** ee.

$$\begin{array}{rrl} \text{code}^i\ \text{if}(c)\ tt\ \text{else}\ ee\ \rho & = & \\ & & \text{code}^i_R\ c\ \rho \\ & & \text{jumpz}\ R_i\ A \\ & & \text{code}^i\ tt\ \rho \\ & & \text{jump}\ B \\ & A: & \text{code}^i\ ee\ \rho \\ & B: & \end{array}$$



Example for if-statement

$i = 8$

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

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

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

$$\begin{array}{rl} \text{code}^8_R\ x>y\ \rho = & \text{mov}\ R_8\ R_4 \\ & \text{mov}\ R_9\ R_7 \\ & \text{gr}\ R_8\ R_8\ R_9 \\ & \text{jumpz}\ R_8\ A \\ & \text{code}^8\ x=x-y\ \rho \\ & \text{jump}\ B \\ A: & \text{code}^8\ y=y-x\ \rho \\ B: & \end{array}$$

Example for if-statement

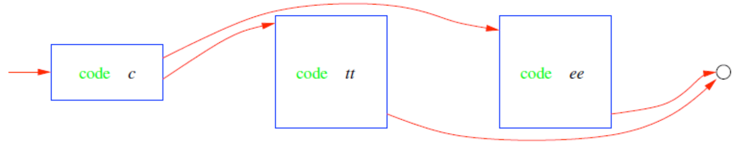
Let $\rho = \{x \mapsto 4, y \mapsto 7\}$ and let s be the statement

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

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

(i)	(ii)	(iii)
move R_i R_4	move R_i R_4	A: move R_i R_7
move R_{i+1} R_7	move R_{i+1} R_7	move R_{i+1} R_4
gr R_i R_i R_{i+1}	sub R_i R_i R_{i+1}	sub R_i R_i R_{i+1}
jumpz R_i A	move R_4 R_i	move R_7 R_i
	jump B	B:

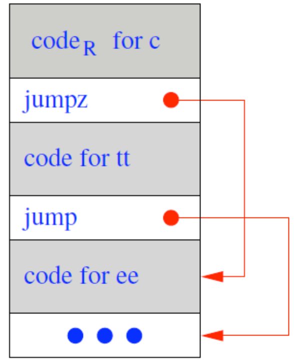
General Conditional



Translation of **if** (c) tt **else** ee.

$$\text{code}^i\ \texttt{if}(c)\ tt\ \texttt{else}\ ee\ \rho \quad =$$

$\text{code}^i_R\ c\ \rho$
jumpz R_i A
$\text{code}^i\ tt\ \rho$
jump B
A: $\text{code}^i\ ee\ \rho$
B:



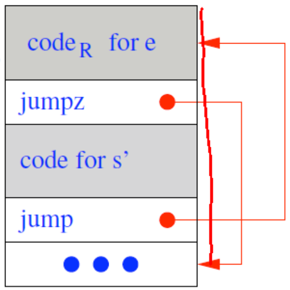
Iterating Statements

if (e) {
do {
s
} while (e);

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

$$\text{code}^i\ \texttt{while}(e)\ s\ \rho \quad = A:$$

$\text{code}^i_R\ e\ \rho$
jumpz R_i B
$\text{code}^i\ s\ \rho$
jump A
B:



code^i while(e) s ρ = jump A
C: code^i s ρ
A: code^i_R e ρ
jump R$_i$ C
B:

Example: Translation of Loops

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

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

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

(i)	(ii)	(iii)
A: move R_i R_7	move R_i R_9	move R_i R_7
loadc R_{i+1} 0	loadc R_{i+1} 1	move R_{i+1} R_8
gr R_i R_i R_{i+1}	add R_i R_i R_{i+1}	sub R_i R_i R_{i+1}
jumpz R_i B	move R_9 R_i	move R_7 R_i
		$\boxed{\text{jump } A}$
		B:

for-Loops

The **for**-loop $s \equiv$ **for** $(e_1; e_2; e_3)\ s'$ 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:

$$
\begin{array}{rrl}
\text{code}^i\ \texttt{for}(e_1; e_2; e_3)\ s\ \rho & = & \text{code}^i_R\ e_1\ \rho \\
 & A: & \text{code}^i_R\ e_2\ \rho \\
 & & \text{jumpz}\ R_i\ B \\
 & & \text{code}^i\ s\ \rho \\
 & & \text{code}^i_R\ e_3\ \rho \\
 & & \text{jump}\ A \\
 & B: &
\end{array}
$$

The switch-Statement

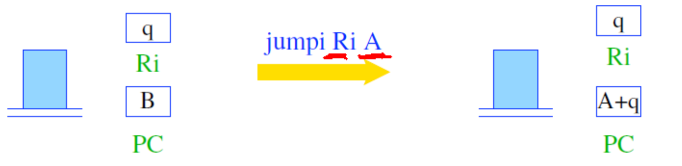
Idea:

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

The switch-Statement

Idea:

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



PC = A + R_i;

Consecutive Alternatives

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

```
switch (e) {
  case c0: s0; break;
  ⋮
  case ck−1: sk−1; break;
  default: s; break;
}
```

that is, $c_i + 1 = c_{i+1}$ for $i = [0, k-1]$.

Consecutive Alternatives

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

$$
\begin{array}{l}
\texttt{switch}\ (e)\ \{\\
\quad \texttt{case}\ c_0\texttt{:}\ s_0\texttt{;}\ \texttt{break;}\\
\quad \vdots\\
\quad \texttt{case}\ c_{k-1}\texttt{:}\ s_{k-1}\texttt{;}\ \texttt{break;}\\
\quad \texttt{default:}\ s\texttt{;}\ \texttt{break;}\\
\}
\end{array}
$$

that is, $c_i + 1 = c_{i+1}$ for $i = [0, k-1]$.
Define $\text{code}^i\ s\ \rho$ as follows:

$B + R_i$

$$
\begin{array}{rcll}
\text{code}^i\ s\ \rho & = & \text{code}^i_R\ e\ \rho & \\
 & & \mathit{check}^i\ c_0\ c_{k-1}\ B\ \ C & B:\ \ \text{jump}\ A_0\\
A_0: & & \text{code}^i\ s_0\ \rho & \quad\vdots \qquad \vdots\\
 & & \text{jump}\ D & \qquad\ \ \text{jump}\ A_{k-1}\\
\vdots & & \vdots & C:\\
A_{k-1}: & & \text{code}^i\ s_{k-1}\ \rho & D:\ \ \text{code}\ s\ \rho\ \}\ \text{default}\\
 & & \text{jump}\ D &
\end{array}
$$

Consecutive Alternatives

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

$$
\begin{array}{l}
\texttt{switch}\ (e)\ \{\\
\quad \texttt{case}\ c_0\texttt{:}\ s_0\texttt{;}\ \texttt{break;}\\
\quad \vdots\\
\quad \texttt{case}\ c_{k-1}\texttt{:}\ s_{k-1}\texttt{;}\ \texttt{break;}\\
\quad \texttt{default:}\ s\texttt{;}\ \texttt{break;}\\
\}
\end{array}
$$

that is, $c_i + 1 = c_{i+1}$ for $i = [0, k-1]$.
Define $\text{code}^i\ s\ \rho$ as follows:

$$
\begin{array}{rcll}
\text{code}^i\ s\ \rho & = & \text{code}^i_R\ e\ \rho & \\
 & & \mathit{check}^i\ c_0\ c_{k-1}\ B & B:\ \ \text{jump}\ A_0\\
A_0: & & \text{code}^i\ s_0\ \rho & \quad\vdots \qquad \vdots\\
 & & \text{jump}\ D & \qquad\ \ \text{jump}\ A_{k-1}\\
\vdots & & \vdots & C:\\
A_{k-1}: & & \text{code}^i\ s_{k-1}\ \rho & \\
 & & \text{jump}\ D &
\end{array}
$$

$\mathit{check}^i\ l\ u\ B$ checks if $l \le R_i < u$ holds and jumps accordingly.

Translation of the check^i Macro

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

- if $l \le R_i < u$ it jumps to $B + R_i - l$
- if $R_i < l$ or $R_i \ge u$ it jumps to C

$$
\begin{array}{ll}
B: & \text{jump}\ A_0\\
\vdots & \quad\vdots\\
 & \text{jump}\ A_{k-1}\\
C: &
\end{array}
$$

Translation of the check^i Macro

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

- if $l \le R_i < u$ it jumps to $B + R_i - l$
- if $R_i < l$ or $R_i \ge u$ it jumps to C

we define: $R_i - l \ge k$

$$
\begin{array}{rcll}
\mathit{check}^i\ l\ u\ B & = & \text{loadc}\ R_{i+1}\ l & \\
 & & \text{geq}\ R_{i+2}\ R_i\ R_{i+1} & \\
 & & \text{jumpz}\ R_{i+2}\ E & B:\ \ \text{jump}\ A_0 \quad \big]\ +R_i - l\\
 & & \text{sub}\ R_i\ R_i\ R_{i+1} & \quad\vdots \qquad \vdots\\
 & & \text{loadc}\ R_{i+1}\ k & \qquad\ \ \text{jump}\ A_{k-1}\\
 & & \text{geq}\ R_{i+2}\ R_i\ R_{i+1} & C:\\
 & & \text{jumpz}\ R_{i+2}\ D & \\
E: & & \text{loadc}\ R_i\ k & \\
D: & & \text{jumpi}\ R_i\ B &
\end{array}
$$

Translation of the $check^i$ Macro

The macro $check^i\ l\ u\ B$ 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:

$$
\begin{array}{rrll}
check^i\ l\ u\ B & = & & \text{loadc } R_{i+1}\ l \\
 & & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & & \text{jumpz } R_{i+2}\ E \\
 & & & \text{sub } R_i\ R_i\ R_{i+1} \\
 & & & \text{loadc } R_{i+1}\ k \\
 & & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & & \text{jumpz } R_{i+2}\ D \\
 & & E: & \text{loadc } R_i\ k \\
 & & D: & \text{jumpi } R_i\ B
\end{array}
\qquad
\begin{array}{rl}
B: & \text{jump } A_0 \\
\vdots & \quad \vdots \\
 & \text{jump } A_{k-1} \\
C: &
\end{array}
$$

Note: a jump $\text{jumpi } R_i\ B$ with $R_i = k$ winds up at C.

Improvements for Jump Tables

This translation is only suitable for *certain* **switch**-statement.

- In case the table starts with 0 instead of u we don't need to subtract it from e before we use it as index
- if the value of e is guaranteed to be in the interval $[l, u]$, we can omit check
- can we implement the **switch**-statement using an L-attributed system without symbolic labels?

Improvements for Jump Tables

This translation is only suitable for *certain* **switch**-statement.

- In case the table starts with 0 instead of u we don't need to subtract it from e before we use it as index
- if the value of e is guaranteed to be in the interval $[l, u]$, we can omit check
- can we implement the **switch**-statement using an L-attributed system without symbolic labels?
  - difficult since B is unknown when $check^i$ is translated
  - $\leadsto$ use symbolic labels or basic blocks

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

switch (e) {
case 5:
9
12
13
14
}

if (e ≤ 12) {
if (e ≤ 9) {
if (e = 5) ...
else
if (e = 9) ...
} else
if (e = 12) {...}
else ...
} else
if

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 $\leadsto O(\log n)$ tests

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 $\leadsto O(\log n)$ tests
- if the sequence of numbers has small gaps (≤ 3), a jump table may be smaller and faster

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 $\leadsto O(\log n)$ tests
- if the sequence of numbers has small gaps (≤ 3), a jump table may be smaller and faster
- one could generate several jump tables, one for each sets of consecutive cases

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 $\leadsto O(\log n)$ tests
- if the sequence of numbers has small gaps (≤ 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

Translation into Basic Blocks

Problem: How do we connect the different basic blocks?

Idea:

- translation of a function: create an empty block and store a pointer to it in the node of the function declaration

Translation into Basic Blocks

Problem: How do we connect the different basic blocks?

Idea:

- translation of a function: create an empty block and store a pointer to it in the node of the function declaration
- pass this block down to the translation of statements

Translation into Basic Blocks

Problem: How do we connect the different basic blocks?

Idea:

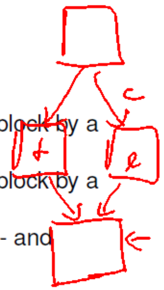
- translation of a function: create an empty block and store a pointer to it in the node of the function declaration
- pass this block down to the translation of statements
- each new statement is appended to this basic block
- a two-way **if**-statement creates three new blocks:
  1. one for the `then`-branch, connected with the current block by a jumpz-edge
  2. one for the **else**-branch, connected with the current block by a jump-edge
  3. one for the following statements, connect to the `then`- and **else**-branch by a jump edge

Translation into Basic Blocks

Problem: How do we connect the different basic blocks?

Idea:

- translation of a function: create an empty block and store a pointer to it in the node of the function declaration
- pass this block down to the translation of statements
- each new statement is appended to this basic block
- a two-way **if**-statement creates three new blocks:
  1. one for the `then`-branch, connected with the current block by a jumpz-edge
  2. one for the **else**-branch, connected with the current block by a jump-edge
  3. one for the following statements, connect to the `then`- and **else**-branch by a jump edge
- similar for other constructs



Translation into Basic Blocks

Problem: How do we connect the different basic blocks?
Idea:

- translation of a function: create an empty block and store a pointer to it in the node of the function declaration
- pass this block down to the translation of statements
- each new statement is appended to this basic block
- a two-way **if**-statement creates three new blocks:
  1. one for the `then`-branch, connected with the current block by a jumpz-edge
  2. one for the **else**-branch, connected with the current block by a jump-edge
  3. one for the following statements, connect to the `then`- and **else**-branch by a jump edge
- similar for other constructs

For better navigation in later stages, it can be necessary to also add *backward* edges.

(e ? e1 : e2)

Chapter 5:
Functions

Ingredients of a Function

int f() { ... }

The definition of a function consists of

int g(int (*f)()) {
f()
}

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

g(&f)

In C we have:

$$\mathrm{code}^i_R\ f\ \rho \quad = \quad \mathrm{loadc}^{K_i}\ \_f \quad \text{with } \_f \text{ starting address of } f$$

Observe: loadc K_i F F: code f ρ

- 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

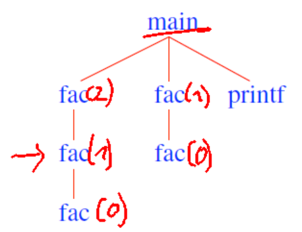
Memory Management in Functions

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

```
int main(void) {
  int n;
  n = fac(2) + fac(1);
  printf("%d", n);
}
```

main
↓
fac

At run-time several instance 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:

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 allocate on the stack

Memory Management in Function Variables

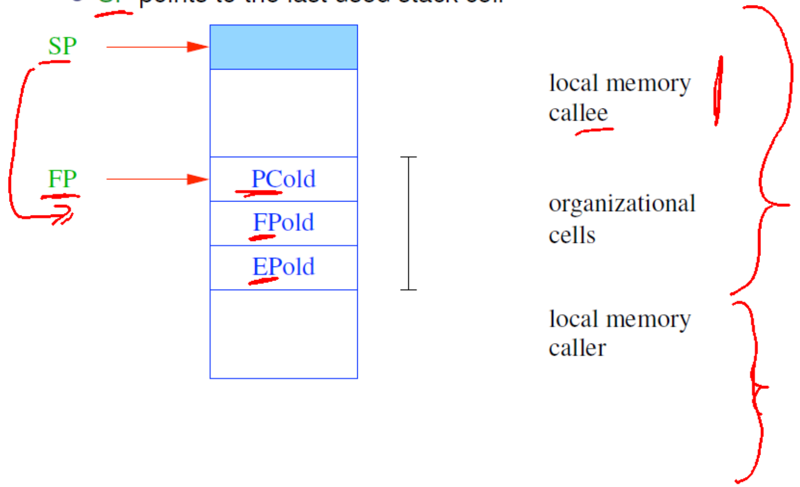
The formal parameters and the local variables of the various (instances) of a function must be kept separate

Idea for implementing functions:

- set up a region of memory each time it is called
- in sequential programs this memory region can be allocate on the stack
- thus, each instance of a function has its own region on the stack

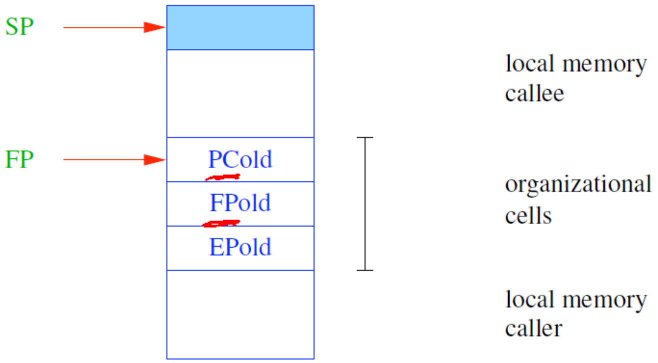
Organization of a Stack Frame

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



Organization of a Stack Frame

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



- FP ≙ frame pointer: points to the last organizational cell
- use to recover the previously active stack frame

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.

local memory – callee

f(int s) {
char *p = alloca(s);
}

Observation:

The space requirement for ~~local~~ memory parameters is only know by the ~~callee~~ caller
Example: `printf` (char *s, ...)

Principle of Function Call and Return

actions taken on entering g: caller f callee g

1. compute the start address of g
2. compute actual parameters
3. backup of caller-save registers } saveloc
4. backup of FP, EP } mark
5. set the new FP
6. back up of PC und jump to the beginning of g } call
7. setup new EP } enter
8. allocate space for local variables } alloc

(1.–6. are in f; 7.–8. are in g)

actions taken on leaving g:

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

(1.–3. are in g; 4.–5. are in f)