### Rules and Examples for Subtyping

\[
\begin{align*}
\text{int} \times i : \text{float} \ y : & \quad \text{type}(x) \leq \text{type}(y) \\
\text{y} & = \times i \text{ convert}
\end{align*}
\]

#### Examples:

\[
\begin{align*}
\text{int} \times \langle \text{int} \rangle \times \langle \text{int} \rangle & \quad \text{float} \ (\times y) (\langle \text{float} \rangle) \\
\langle \text{int} \rangle & \quad \text{struct} \{ \text{int} \ a; \text{int} b; \} \\
\langle \text{int} \rangle & \quad \text{struct} \{ \text{float} \ a; \} \\
\langle \text{float} \rangle & \quad \text{float} (\langle \text{float} \rangle) \\
\langle \text{float} \rangle & \quad \text{float} (\langle \text{int} \rangle)
\end{align*}
\]

#### 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 C-Machine 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:
- 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
- the global registers \( R_j \)
  - save the parameters of a function
  - store the result of a function
Translation of Simple Expressions

Using variables stored in registers; loading constants:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $R_i$, $c$</td>
<td>$R_i = c$</td>
<td>load constant</td>
</tr>
<tr>
<td>move $R_i$, $R_j$</td>
<td>$R_i = R_j$</td>
<td>copy $R_j$ to $R_i$</td>
</tr>
</tbody>
</table>

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

- $\text{code}_{R} c \rho = \text{load} \ R_i$, $c$
- $\text{code}_{R} x \rho = \text{move} \ R_i$, $R_j$
- $\text{code}_{R} x = e \rho = \text{code}_{R} e \rho$
- $\text{move} \ R_i$, $R_j$

Note: all instructions use the Intel convention (in contrast to the AT&T convention): $\text{op} \ \text{dst} \ \text{src}_1 \ldots \text{src}_n$.

Translation of Expressions

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

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

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

Correspondingly, we generate code as follows:

$\text{code}_{R} e_1 \text{op} e_2 \rho = \text{code}_{R} e_1 \rho \text{op} R_i$, $R_j$, $R_{i+1}$

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

$\text{code}_{R} 3 \times 4 \rho = \text{code}_{R} 3 \rho \text{op} R_4$, $R_4$, $R_5$

Translation of Expressions

Using variables stored in registers; loading constants:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $R_i$, $c$</td>
<td>$R_i = c$</td>
<td>load constant</td>
</tr>
<tr>
<td>move $R_i$, $R_j$</td>
<td>$R_i = R_j$</td>
<td>copy $R_j$ to $R_i$</td>
</tr>
</tbody>
</table>

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

- $\text{code}_{R} c \rho = \text{load} \ R_i$, $c$
- $\text{code}_{R} x \rho = \text{move} \ R_i$, $R_j$
- $\text{code}_{R} x = e \rho = \text{code}_{R} e \rho$
- $\text{move} \ R_i$, $R_j$

Note: all instructions use the Intel convention (in contrast to the AT&T convention): $\text{op} \ \text{dst} \ \text{src}_1 \ldots \text{src}_n$.
Applying Translation Schema for Expressions

Suppose the following function is given:

```c
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, \( O = 4 \).

- \( \text{code}^4_x = y + z + 3 \rho \) = move \( R_4, R_3 \)

- \( \text{code}^4_y = y + z + 3 \rho \) = move \( R_4, R_2 \)

- \( \text{code}^4_z = y + z + 3 \rho \) = add \( R_4, R_1, R_5 \)

About Statements and Expressions

General idea for translation:

- \( \text{code}^{s} s \rho \) : generate code for statement \( s \)
- \( \text{code}^{e} e \rho \) : 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:

\[
\text{code}^{e}_x x = e \rho = \text{code}^{e}_x e \rho
\]

However, \( x = e \) is also a statement:

- Define:

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

The temporary register \( R_i \) is ignored here. More general:

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

Translation of Statement Sequences

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

\[
\text{code}^{d} (s; s) \rho = \text{code}^{d} s \rho
\]

\[
\text{code}^{d} _\emptyset \rho = \text{code}^{d} ss \rho
\]

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:

\[ \text{PC} = A; \]

Conditional Jumps

A conditional jump branches depending on the value in \( R_i \):

\[ \text{if } (R_i == 0) \text{ PC} = A; \]

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 \textit{far jumps} through \textit{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

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:
- 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.

Outgoing edges must have the following form:
- a single edge (unconditional jump), translated with jump
- two edges, one with $c = 0$ as condition and one without condition, translated with jumpz and jump, respectively

 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
- sometimes 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
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
- sometimes 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 $s = \text{if } (c) \{ ss \}$

... 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

\[
\text{code}^f_s \  \rho \quad = \quad \begin{cases} \text{code}^c_c \  \rho \\
\quad \quad \text{jumpz} \  R_i \  A \\
\quad \quad \text{code}^f_{ss} \  \rho \\
\end{cases}
\]

Example for if-statement

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

\[
\text{if } (x > y) \{ \begin{array}{l}
\text{/* (i) */}
\quad x = x - y; \quad /\!\!\!\text{/* (ii) */}
\text{else} \{ \begin{array}{l}
\text{/* (iii) */}
\quad y = y - x;
\end{array} \}
\end{array}
\]

Then $\text{code}^f_s \  \rho$ yields:

\[
\text{code}^f_{x>y} \  \rho \quad = \quad \begin{cases} \text{mov} \  R_y \  R_x \\
\quad \quad \text{mov} \  R_y \  R_4 \\
\quad \quad \text{mov} \  R_g \  R_8 \\
\quad \quad \text{mov} \  R_8 \  R_8 \\
\quad \quad \text{mov} \  R_8 \  R_4 \\
\quad \quad \text{jump} \  R_i \  A \\
\\end{cases}
\]

\[
\begin{array}{l}
\text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^c_c \  \rho \\
\quad \quad \text{jumpz} \  R_i \  A \\
\quad \quad \text{jump} \\
\quad \quad \text{code}^f_{tt} \  \rho \\
\quad \quad \text{code}^f_{tt} \  \rho \\
\quad \quad \text{code}^f_{tt} \  \rho \\
\\end{array}
\]

\[
\begin{array}{l}
\text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\\end{array}
\]

\[
\begin{array}{l}
\text{A:} \quad \text{jumpz} \  R_i \  A \\
\quad \quad \text{jump} \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\\end{array}
\]

\[
\begin{array}{l}
\text{B:} \quad y = y - x; \\
\quad \quad \text{jump} \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\quad \quad \text{code}^f_{x>y} \  \rho \\
\\end{array}
\]
Example for if-statement

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

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

Then $\text{code}'$ $s$ $\rho$ yields:

(i) move $R_4$
move $R_{i+1}$ $R_7$
move $R_{i+1}$ $R_7$
gr $R_i$ $R_i$ $R_{i+1}$
jumpz $R_i$ $A$

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

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

General Conditional

Translation of $\text{if} (c) \text{ or else} ee$.

```c
\text{code}' \text{if}(c) \text{ or else} ee \ \rho =
\text{code}_R \text{ for } c
\text{jumpz}
\text{code}_R \text{ for } ee \ \rho
\text{jump} \text{for } ee$
```

Iterating Statements

We only consider the loop $s = \text{while} (c)$.

For this statement we define:

```c
\text{code}' \text{while}(c) \ s \ \rho = A:
\text{code}_R \text{ for } c
\text{jumpz } R_i \ B
\text{code}' s \ \rho
\text{jump } A$
```

Example: Translation of Loops

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

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

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

(i) move $R_4$
load $R_{i+1}$ $0$
gr $R_i$ $R_i$ $R_{i+1}$
jumpz $R_i$ $B$

(ii) move $R_9$
load $R_{i+1}$ $1$
add $R_i$ $R_i$ $R_{i+1}$

(iii) move $R_6$
sub $R_i$ $R_i$ $R_{i+1}$
for-Loops

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

Thus, we translate:

$$
\text{code}' \text{ for}(e_1; e_2; e_3) \ s' = \text{code}' e_1 \ s' \ \text{ A} \\
\text{code}' e_2 \ s' \ \text{ B} \\
\text{code}' e_3 \ s' \ \text{ A} \\
\text{jump} z_{\text{B}} \ \text{ A} \\
\text{jump} z \ \text{ A}
$$

The switch-Statement

Idea:

- Suppose choosing from multiple options in \textit{constant time} if possible
- use a \textit{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 \textit{indirect jump} instruction

Consecutive Alternatives

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

```
switch (e) {
    case (c_0) \ s_0; break;
    ...
    case (c_{k-1}) \ s_{k-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:

```c
switch (s) {
    case $c_0$: $s_0$; break;
    ...
    case $c_{k-1}$: $s_{k-1}$; break;
    default: $s$; break;
}
```

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

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

```c
\text{check}^i c_0 \rho \quad \text{jump } A_0
\vdots
\text{check}^i c_{k-1} \rho \quad \text{jump } A_{k-1}
\text{jump } D
```

Translation of the check$^i$ Macro

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

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

```c
B : \text{jump } A_0
    \vdots
    \text{jump } A_{k-1}
C :
```

Translation of the check$^i$ Macro

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

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

we define: $R_l - l < l$

```c
\text{check}^2 l u B = \text{load } R_{i+2}, l
    \text{geq } R_{i+2}, R_{i+1}
    \text{jumpz } R_{i+2}, E
\vdots
\text{load } R_{i+2}, l
    \text{geq } R_{i+2}, R_{i+1}
    \text{jumpz } R_{i+2}, D
E : \text{load } R_{i} k
    \text{jump } R_{i} B
```

```c
B : \text{jump } A_0
    \vdots
    \text{jump } A_{k-1}
C :
```
Translation of the `checkl` Macro

The macro `checkl 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:

\[
\begin{align*}
\text{checkl l u B} &= \text{loadc } R_{i+1} l \\
& \quad \text{geq } R_{i+2} R_i R_{i+1} \\
& \quad \text{jumpz } R_{i+2} E \\
& \quad \text{sub } R_i R_i R_{i+1} \\
& \quad \text{loadc } R_{i+1} k \\
& \quad \text{geq } R_{i+2} R_i R_{i+1} \\
& \quad \text{jumpz } R_{i+2} D \\
E : & \text{ loadc } R_i k \\
C : & \text{ jump } A_{k-1}
\end{align*}
\]

Note: a jump `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 \( l \) 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?

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
In general, the values of the various cases may be far apart:

- generate an \texttt{if}\text{-}ladder, that is, a sequence of \texttt{if}\text{-}statements
- for \( n \) cases, an \texttt{if}\text{-}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
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:
  - one for the then-branch, connected with the current block by a jump edge
  - one for the else-branch, connected with the current block by a jump edge
  - 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:
  - one for the then-branch, connected with the current block by a jump-edge
  - one for the else-branch, connected with the current block by a jump-edge
  - 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.

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 return type;
- a sequence of statements.

In C we have:

\[
\begin{align*}
\text{code}_f &= \text{load}_{\mathbb{F}_i} f \\
\text{with } f &\text{ starting address of } f
\end{align*}
\]

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) {
    int g(int); 
    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:
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

**Organization of a Stack Frame**

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

```
+-------------------+
| PCold             |
| FPold             |
| EPold             |
+-------------------+
```

local memory
caller

organizational
cells

local memory
caller
Organization of a Stack Frame

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

**FP**
- **PCold**
- **FPold**
- **EPold**

**SP**
- **Local memory**
  - **Caller**
  - **Callee**

**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.

**Observation**

The space requirement for local parameters is only known by the callee.

**Example**:

```c
printf(char *s, ...)
```