Script **generated by TTT**

Title: Petter: Compilerbau (06.07.2015)

Date: Mon Jul 06 14:18:07 CEST 2015

Duration: 98:11 min

Pages: 63

Overloading and Coercion

Some operators such as + are *overloaded*:

- + has *several possible* types
  for example: `int +(int,int)`, `float +(float, float)`
  but also `float* +(float*, int)`, `int* +(int, int*)`
- depending on the type, the operator + has a different implementation
- determining which implementation should be used is based on the *arguments* only

---

Overloading and Coercion

Some operators such as + are *overloaded*:

- + has *several possible* types
  for example: `int +(int,int)`, `float +(float, float)`
  but also `float* +(float*, int)`, `int* +(int, int*)`
- depending on the type, the operator + has a different implementation
- determining which implementation should be used is based on the *arguments* only

Coercion: allow the application of + to `int` and `float`.

- instead of defining + for all possible combinations of types, the arguments are automatically coerced
- this coercion may generate code (e.g. conversion from `int` to `float`)
- conversion is usually done towards more general types i.e. `5+0.5` has type `float` (since `float` ≥ `int`)

Coercion of Integer-Types in C: Promotion

C defines special conversion rules for integers: *promotion*

`unsigned char` ≤ `unsigned short` ≤ `int` ≤ `unsigned int`
`signed char` `signed short`

. . . where a conversion has to happen via all intermediate types.

Coercion of Integer-Types in C: Promotion

C defines special conversion rules for integers: *promotion*

$$\begin{array}{c}\texttt{unsigned char}\\ \texttt{signed char}\end{array} \leq \begin{array}{c}\texttt{unsigned short}\\ \texttt{signed short}\end{array} \leq \texttt{int} \leq \boxed{\texttt{unsigned int}}$$

. . . where a conversion has to happen via all intermediate types.

subtle errors possible! Compute the character distribution of `str`:

```
char* str = "...";
int dist[256];
memset(dist, 0, sizeof(dist));
while (*str) {
  dist[(unsigned) *str]++;
  str++;
};
```

Note: **`unsigned`** is shorthand for **`unsigned int`**.

Subtypes

- on the arithmetic basic types **`char`**, **`int`**, **`long`**, etc. there exists a rich *subtype* hierarchy
- here $t_1 \leq t_2$, means that the values of type t_1
  1. form a subset of the values of type t_2;
  2. can be converted into a value of type t_2;
  3. fulfill the requirements of type t_2.

Subtypes

- on the arithmetic basic types **`char`**, **`int`**, **`long`**, etc. there exists a rich *subtype* hierarchy
- here $t_1 \leq t_2$, means that the values of type t_1
  1. form a subset of the values of type t_2;
  2. can be converted into a value of type t_2;
  3. fulfill the requirements of type t_2.

Example: assign smaller type (fewer values) to larger type

$$\begin{array}{l} t_1 \quad x; \\ t_2 \quad y; \\ y = x; \end{array}$$

Subtypes

- on the arithmetic basic types **`char`**, **`int`**, **`long`**, etc. there exists a rich *subtype* hierarchy
- here $t_1 \leq t_2$, means that the values of type t_1
  1. form a subset of the values of type t_2;
  2. can be converted into a value of type t_2;
  3. fulfill the requirements of type t_2.

Example: assign smaller type (fewer values) to larger type

$$\begin{array}{l} t_1 \quad x; \\ t_2 \quad y; \\ y = x; \end{array}$$

extend the subtype relationship to more complex types

Example: Subtyping

Observe:

```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

- we would like `extractInfo` to be applicable to all argument records that contain a field `string info`
- use deduction rules to describe when $t_1 \leq t_2$ should hold
- the idea of subtyping on values is related to subtyping as implemented in object-oriented languages

---

Example: Subtyping

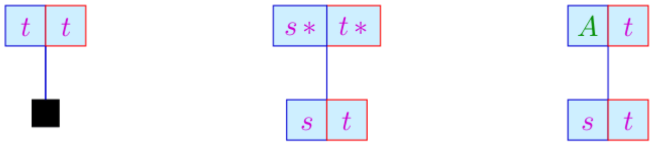
Observe:

```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

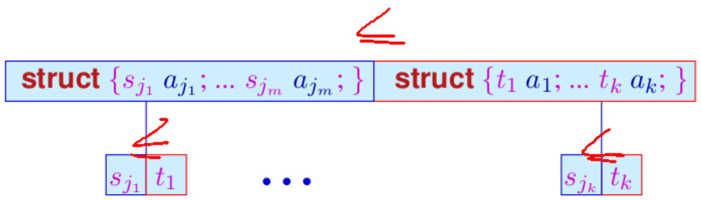
- we would like `extractInfo` to be applicable to all argument records that contain a field `string info`
- use deduction rules to describe when $t_1 \leq t_2$ should hold
- the idea of subtyping on values is related to subtyping as implemented in object-oriented languages

---

Rules for Well-Typedness of Subtyping

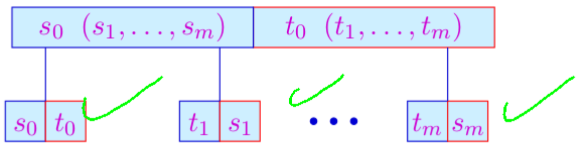


$A = s$



$\text{struct }\{\text{int } u, \text{int } v\}\quad x;$
$\text{struct }\{\text{int } u\}\quad y;$
$y = x;$

Rules and Examples for Subtyping



Examples:

struct {int a; int b;}   struct {float a;}
int (int)   float (float)
int (float) ≤ float (int)

Co- and Contra Variance

Definition

Given two function types in subtype relation $s_0(s_1, \dots s_n) \leq t_0(t_1, \dots t_n)$ then we have

- co-variance of the return type $s_0 \leq t_0$ and
- contra-variance of the arguments $s_i \geq t_i$ für $1 < i \leq n$

Co- and Contra Variance

Definition

Given two function types in subtype relation $s_0(s_1, \dots s_n) \leq t_0(t_1, \dots t_n)$ then we have

- co-variance of the return type $s_0 \leq t_0$ and
- contra-variance of the arguments $s_i \geq t_i$ für $1 < i \leq n$

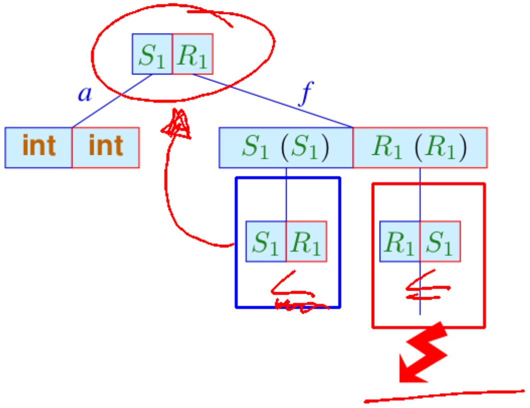
Example from functional languages:

$$\boxed{\textbf{int} \rightarrow \textbf{float}} \rightarrow \boxed{\textbf{int}} \leq \boxed{\textbf{int} \rightarrow \textbf{int}} \rightarrow \boxed{\textbf{float}}$$

Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

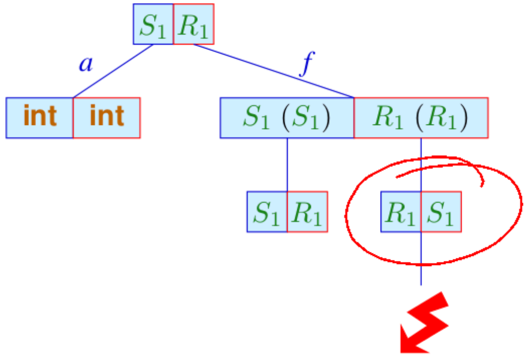
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\} \end{array}$$



Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

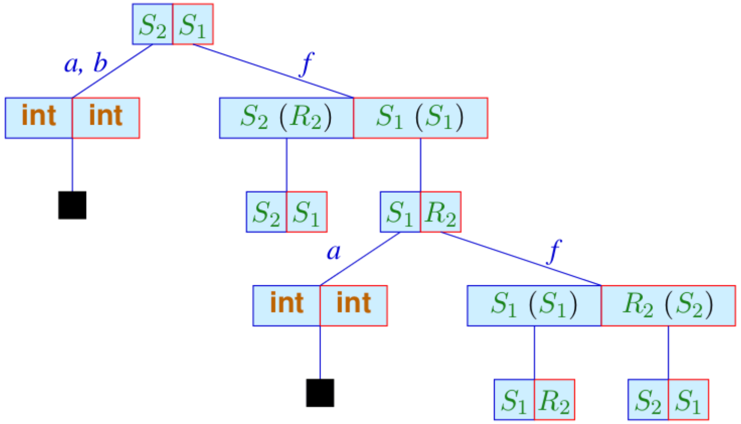
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\} \end{array}$$



Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

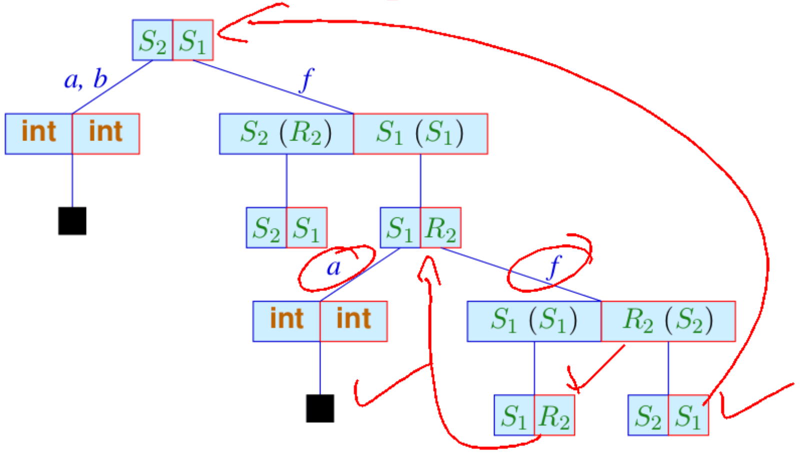
$$\begin{aligned}
R_1 &= \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\
S_1 &= \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\
R_2 &= \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\
S_2 &= \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\}
\end{aligned}$$



Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

$$\begin{aligned}
R_1 &= \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\
S_1 &= \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\
R_2 &= \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\
S_2 &= \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\}
\end{aligned}$$



Generating Code: Overview

We inductively generate instructions from the AST:

- there is a rule stating how to generate code for each non-terminal of the grammar
- the code is merely another attribute in the syntax tree
- code generation makes use of the already computed attributes

---

Generating Code: Overview

We inductively generate instructions from the AST:

- there is a rule stating how to generate code for each non-terminal of the grammar
- the code is merely another attribute in the syntax tree
- code generation makes use of the already computed attributes

In order to specify the code generation, we require

- a semantics of the language we are compiling (here: C standard)
- a semantics of the machine instructions

Generating Code: Overview

We inductively generate instructions from the AST:

- there is a rule stating how to generate code for each non-terminal of the grammar
- the code is merely another attribute in the syntax tree
- code generation makes use of the already computed attributes

In order to specify the code generation, we require

- a semantics of the language we are compiling (here: C standard)
- a semantics of the machine instructions

⇝ we commence by specifying machine instruction semantics

The Register C-Machine (R-CMa)

We generate Code for the Register C-Machine.
The Register C-Machine is a virtual machine (VM).

- there exists no processor that can execute its instructions
- . . . but we can build an interpreter for it
- we provide a visualization environment for the R-CMa
- the R-CMa has no **`double`**, **`float`**, **`char`**, **`short`** or **`long`** types
- the R-CMa has no instructions to communicate with the operating system
- the R-CMa has an unlimited supply of registers

The Register C-Machine (R-CMa)

We generate Code for the Register C-Machine.
The Register C-Machine is a virtual machine (VM).

- there exists no processor that can execute its instructions
- . . . but we can build an interpreter for it
- we provide a visualization environment for the R-CMa
- the R-CMa has no **`double`**, **`float`**, **`char`**, **`short`** or **`long`** types
- the R-CMa has no instructions to communicate with the operating system
- the R-CMa has an unlimited supply of registers

The R-CMa is more realistic than it may seem:

- the mentioned restrictions can easily be lifted
- the *Dalvik VM* or the *LLVM* are similar to the R-CMa
- an interpreter of R-CMa can run on any platform

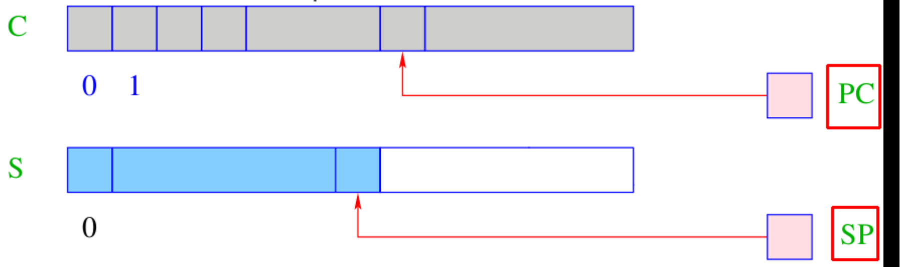
Virtual Machines

A virtual machines has the following ingredients:

- any virtual machine provides a set of instructions
- instructions are executed on virtual hardware
- the virtual hardware is a collection of data structures that is accessed and modified by the VM instructions
- ... and also by other components of the run-time system namely functions that go beyond the instruction semantics
- the interpreter is part of the run-time system

Components of a Virtual Machine

Consider Java as an example:

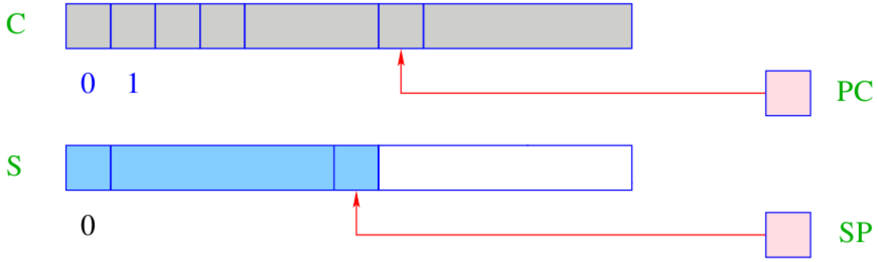


A virtual machine such as the Dalvik VM has the following structure:

- S: the data store – a memory region in which cells can be stored in LIFO order ⇝ stack.
- SP: (≙ stack pointer) pointer to the last used cell in S
- beyond S follows the memory containing the heap

Components of a Virtual Machine

Consider Java as an example:



A virtual machine such as the Dalvik VM has the following structure:

- S: the data store – a memory region in which cells can be stored in LIFO order ⇝ stack.
- SP: (≙ stack pointer) pointer to the last used cell in S
- beyond S follows the memory containing the heap
- C is the memory storing *code*
  - each cell of C holds exactly one virtual instruction
  - C can only be *read*
- PC (≙ program counter) address of the instruction that is to be executed next
- PC contains 0 initially

Executing a Program

- the machine loads an instruction from C[PC] into the instruction register IR in order to execute it
- before evaluating the instruction, the PC is incremented by one

```
while (true) {
  IR = C[PC]; PC++;
  execute (IR);
}
```

- node: the PC must be incremented before the execution, since an instruction may modify the PC
- the loop is exited by evaluating a halt instruction that returns directly to the operating system

---

Simple Expressions and Assignments in R-CMa

Task: evaluate the expression $(1+7)*3$
that is, generate an instruction sequence that

- computes the value of the expression and
- keeps its value accessible in a reproducable way

Simple Expressions and Assignments in R-CMa

Task: evaluate the expression $(1+7)*3$
that is, generate an instruction sequence that

- computes the value of the expression and
- keeps its value accessible in a reproducable way

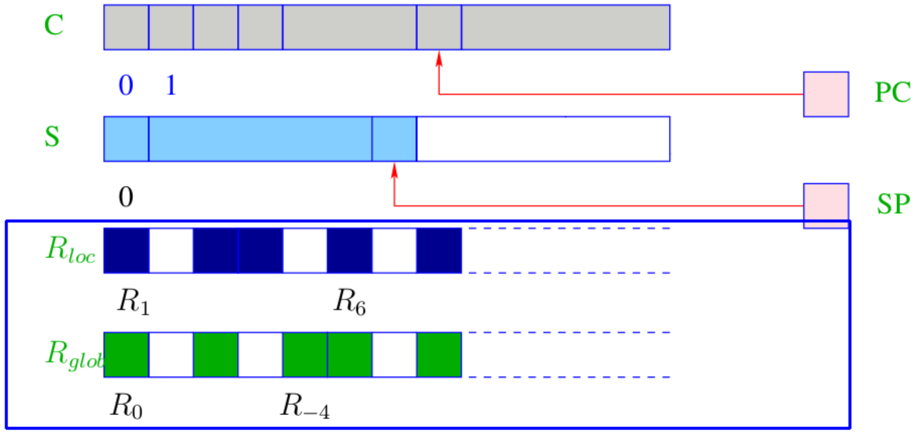
Idea:

- first compute the value of the sub-expressions
- store the intermediate result in a temporary register
- apply the operator
- loop

Principles of the R-CMa

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

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

Note:
for now, we only use registers to store temporary computations

Idea for the translation: use a register counter i:

- registers R_j with $j < i$ are *in use*
- registers R_j with $j \geq i$ are *available*

Translation of Simple Expressions

Using variables stored in registers; loading constants:

instruction	semantics	intuition
$\mathrm{loadc}\ R_i\ c$	$R_i = c$	load constant
$\mathrm{move}\ \boxed{R_i}\,\boxed{R_j}$	$R_i = R_j$	copy R_j to R_i

---

Translation of Simple Expressions

Using variables stored in registers; loading constants:

instruction	semantics	intuition
$\mathrm{loadc}\ R_i\ c$	$R_i = c$	load constant
$\mathrm{move}\ R_i\ R_j$	$R_i = R_j$	copy R_j to R_i

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

$$\begin{aligned}
\mathrm{code}_\mathrm{R}^i\ c\ \rho &= \mathrm{loadc}\ R_i\ c \\
\mathrm{code}_\mathrm{R}^i\ x\ \rho &= \mathrm{move}\ R_i\ R_a \\
\mathrm{code}_\mathrm{R}^i\ \boxed{x} = e\ \rho &= \mathrm{code}_\mathrm{R}^i\ e\ \rho \\
&\quad \boxed{\mathrm{move}\ R_a\ \boxed{R_i}\quad}
\end{aligned}$$

Translation of Simple Expressions

Using variables stored in registers; loading constants:

instruction	semantics	intuition
$\mathrm{loadc}\ R_i\ c$	$R_i = c$	load constant
$\mathrm{move}\ R_i\ R_j$	$R_i = R_j$	copy R_j to R_i

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

$$\begin{aligned}
\mathrm{code}_\mathrm{R}^i\ c\ \rho &= \mathrm{loadc}\ R_i\ c \\
\mathrm{code}_\mathrm{R}^i\ x\ \rho &= \mathrm{move}\ R_i\ R_a \\
\mathrm{code}_\mathrm{R}^i\ x = e\ \rho &= \mathrm{code}_\mathrm{R}^i\ e\ \rho \\
&\quad \mathrm{move}\ R_a\ R_i
\end{aligned}$$

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

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:

$$\begin{aligned}
\mathrm{code}_\mathrm{R}^{\boxed{i}}\ e_1\ \boxed{\text{op}}\ e_2\ \rho &= \mathrm{code}_\mathrm{R}^i\ e_1\ \rho \\
&\quad \mathrm{code}_\mathrm{R}^{i+1}\ e_2\ \rho \\
&\quad \boxed{\text{op}}\,\boxed{R_i}\,\boxed{R_i}\,\boxed{R_{i+1}}
\end{aligned}$$

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:

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

Example: Translate $3*4$ with $i = 4$:

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

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:

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

Example: Translate $3*4$ with $i = 4$:

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

Managing Temporary Registers

Observe that temporary registers are re-used: translate $3*4+3*4$ with $t = 4$:

$$\begin{aligned} \text{code}_R^4 \; 3*4+3*4 \; \rho \quad = \quad & \text{code}_R^{\boxed{4}} \; 3*4 \; \rho \\ & \text{code}_R^{\boxed{5}} \; 3*4 \; \rho \\ & \text{add} \; R_4 \; R_4 \; R_5 \end{aligned}$$

where

$$\begin{aligned} \text{code}_R^i \; 3*4 \; \rho \quad = \quad & \text{loadc} \; R_{\boxed{i}} \; 3 \\ & \text{loadc} \; R_{\boxed{i+1}} \; 4 \\ & \text{mul} \; R_i \; R_i \; R_{i+1} \end{aligned}$$

we obtain

$$\text{code}_R^4 \; 3*4+3*4 \; \rho \quad =$$

Managing Temporary Registers

Observe that temporary registers are re-used: translate $3*4+3*4$ with $t = 4$:

$$\begin{aligned} \text{code}_R^4 \; 3*4+3*4 \; \rho \quad = \quad & \text{code}_R^4 \; 3*4 \; \rho \\ & \text{code}_R^5 \; 3*4 \; \rho \\ & \text{add} \; R_4 \; R_4 \; R_5 \end{aligned}$$

where

$$\begin{aligned} \text{code}_R^i \; 3*4 \; \rho \quad = \quad & \text{loadc} \; R_i \; 3 \\ & \text{loadc} \; R_{i+1} \; 4 \\ & \text{mul} \; R_i \; R_i \; R_{i+1} \end{aligned}$$

we obtain

$$\begin{aligned} \text{code}_R^4 \; 3*4+3*4 \; \rho \quad = \quad & \text{loadc} \; R_4 \; 3 \\ & \text{loadc} \; R_5 \; 4 \\ & \text{mul} \; R_4 \; R_4 \; R_5 \\ & \text{loadc} \; R_5 \; 3 \\ & \text{loadc} \; R_6 \; 4 \\ & \text{mul} \; R_5 \; R_5 \; R_6 \\ & \text{add} \; R_4 \; R_4 \; R_5 \end{aligned}$$

Semantics of Operators

The operators have the following semantics:

add R_i R_j R_k	$R_i = R_j + R_k$
sub R_i R_j R_k	$R_i = R_j - R_k$
div R_i R_j R_k	$R_i = R_j / R_k$
mul R_i R_j R_k	$R_i = R_j * R_k$
mod R_i R_j R_k	$R_i = sgn(R_k)k$ wobei $\lvert R_j\rvert = n\lvert R_k\rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k\rvert$
le R_i R_j R_k	$R_i =$ if $R_j < R_k$ then 1 else 0
gr R_i R_j R_k	$R_i =$ if $R_j > R_k$ then 1 else 0
eq R_i R_j R_k	$R_i =$ if $R_j = R_k$ then 1 else 0
leq R_i R_j R_k	$R_i =$ if $R_j \leq R_k$ then 1 else 0
geq R_i R_j R_k	$R_i =$ if $R_j \geq R_k$ then 1 else 0
and R_i R_j R_k	$R_i = R_j \,\&\, R_k$ // bit-wise and
or R_i R_j R_k	$R_i = R_j \mid R_k$ // bit-wise or

Translation of Unary Operators

Unary operators op $= \{neg,\ not\}$ take only two registers:

$$\begin{aligned} \text{code}^i_{\text{R}}\ \text{op}\ e\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e\ \rho \\ & \quad\quad \text{op}\ R_i\ R_i \end{aligned}$$

Translation of Unary Operators

Unary operators op $= \{neg,\ not\}$ take only two registers:

$$\begin{aligned} \text{code}^i_{\text{R}}\ \text{op}\ e\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e\ \rho \\ & \quad\quad \text{op}\ R_i\ R_i \end{aligned}$$

Note: We use the same register.

Example: Translate -4 into R_5:

$$\begin{aligned} \text{code}^5_{\text{R}}\ \boxed{-4}\ \rho \quad &= \quad \text{code}^5_{\text{R}}\ 4\ \rho \\ & \quad\quad \text{neg}\ R_5\ R_5 \end{aligned}$$

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, $i = 4$.

$$\begin{aligned} \text{code}^4\ \texttt{x=}\boxed{\texttt{y+z*3}}\,\rho \quad &= \quad \text{code}^{\boxed{4}}_{\text{R}}\ \boxed{\texttt{y+z*3}}\,\rho \\ & \quad\quad \text{move}\ R_1\ R_4 \end{aligned}$$

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, $i = 4$.

$$\begin{array}{rcl}
\mathrm{code}^4\ \texttt{x=y+z*3}\ \rho & = & \mathrm{code}^4_\mathrm{R}\ \texttt{y+z*3}\ \rho \\
 & & \mathrm{move}\ R_1\ R_4 \\
\mathrm{code}^4_\mathrm{R}\ \texttt{y+z*3}\ \rho & = & \mathrm{move}\ R_4\ R_2 \\
 & & \mathrm{code}^5_\mathrm{R}\ \texttt{z*3}\ \rho \\
 & & \mathrm{add}\ R_4\ R_4\ R_5 \\
\mathrm{code}^5_\mathrm{R}\ \texttt{z*3}\ \rho & = & \mathrm{move}\ R_5\ R_3 \\
 & & \mathrm{code}^6_\mathrm{R}\ \texttt{3}\ \rho \\
 & & \mathrm{mul}\ R_5\ R_5\ R_6 \\
\mathrm{code}^6_\mathrm{R}\ \texttt{3}\ \rho & = & \mathrm{loadc}\ R_6\ 3
\end{array}$$

$\rightsquigarrow$ the assignment `x=y+z*3` is translated as
$\mathrm{move}\ R_4\ R_2; \mathrm{move}\ R_5\ R_3; \mathrm{loadc}\ R_6\ 3; \mathrm{mul}\ R_5\ R_5\ R_6; \mathrm{add}\ R_4\ R_4\ R_5; \mathrm{move}\ R_1\ R$

Chapter 3:

Statements and Control Structures

About Statements and Expressions

General idea for translation:

$\mathrm{code}^i\ s\ \rho$: generate code for statement s
$\mathrm{code}^i_\mathrm{R}\ e\ \rho$: generate code for expression e into R_i

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

About Statements and Expressions

General idea for translation:

$\mathrm{code}^i\ s\ \rho$: generate code for statement s
$\mathrm{code}^i_\mathrm{R}\ e\ \rho$: 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:

$$\mathrm{code}^i_\mathrm{R}\ x = e\ \rho \quad = \quad \boxed{\begin{array}{l} \mathrm{code}^i_\mathrm{R}\ e\ \rho \\ \mathrm{move}\ R_a\ R_i \end{array}}$$

However, $\boxed{x = e;}$ is also an *expression statement*:

About Statements and Expressions

General idea for translation:

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

$\text{code}^i_{\text{R}}\ e\ \rho$: 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:

$$\begin{aligned} \text{code}^i_{\text{R}} \boxed{x = e} \rho \quad &= \quad \text{code}^i_{\text{R}}\ e\ \rho \\ & \qquad \text{move}\ \boxed{R_a\ \underline{R_i}} \end{aligned}$$

However, $x = e;$ is also an *expression statement*:

- Define:

$x = a = \boxed{z = \underline{42};}$

$$\text{code}^i\ e_1 = e_2;\ \rho \quad = \quad \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho$$

The temporary register R_i is ignored here. More general:

$$\text{code}^i\ e;\ \rho = \text{code}^i_{\text{R}}\ e\ \rho$$

Translation of Statement Sequences

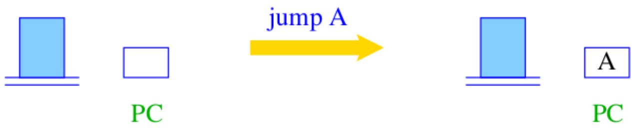
The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

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

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

Jumps

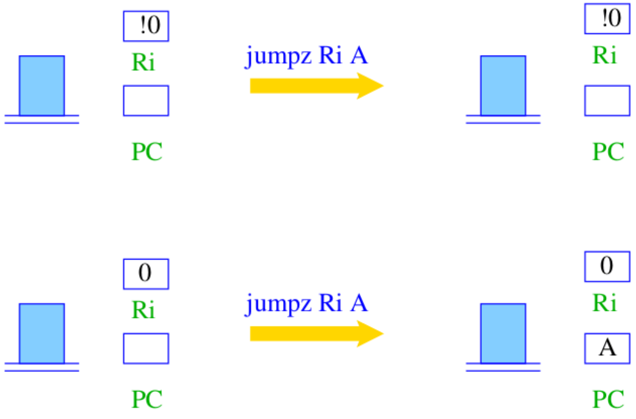
In order to diverge from the linear sequence of execution, we need *jumps*:



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

To this end, we define:

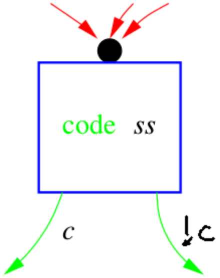
Definition

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:

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

---

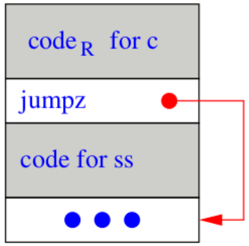
Simple Conditional

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

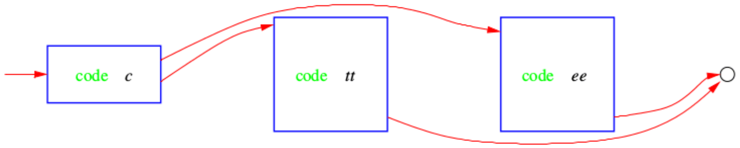
Idea:

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

$\text{code}^i\ s\ \rho \quad = \quad \text{code}^i_R\ c\ \rho$
jumpz R_i A
$\text{code}^i\ ss\ \rho$
$A:\ \ldots$

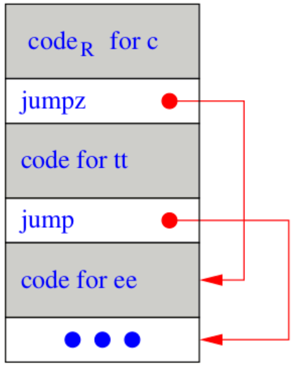


General Conditional



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
$A:\ \text{code}^i\ ee\ \rho$
$B:$



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

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

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

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)

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

(ii) then

move $R_i\ R_4$
move $R_{i+1}\ R_7$
sub $R_i\ R_i\ R_{i+1}$
move $R_4\ R_i$
jump B

(iii) else

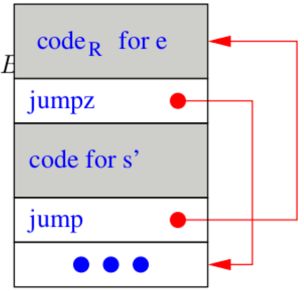
$A:$ move $R_i\ R_7$
move $R_{i+1}\ R_4$
sub $R_i\ R_i\ R_{i+1}$
move $R_7\ R_i$
$B:$

Iterating Statements

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

$$\text{code}^i\ \texttt{while}(e)\ s\ \rho \quad = A: \quad \begin{array}{l} \text{code}^i_{\text{R}}\ e\ \rho \\ \text{jumpz}\ R_i\ B \\ \text{code}^i\ s\ \rho \\ \text{jump}\ A \end{array}$$

$B:$



---

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:

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

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