Script generated by TTT

Title: Simon: Compilerbau (24.06.2013)

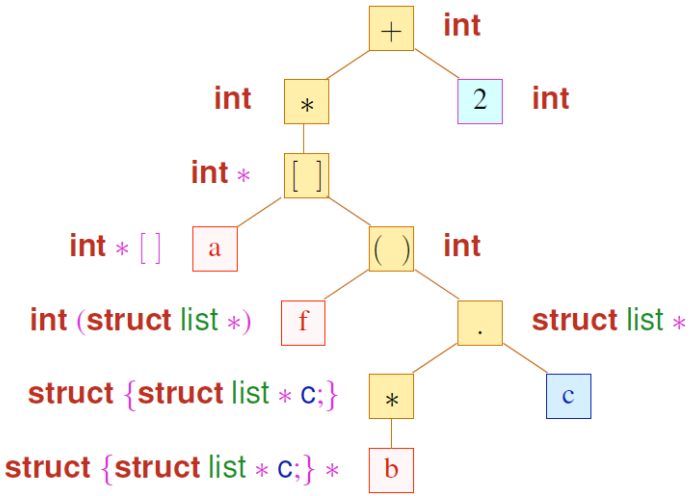
Date: Mon Jun 24 14:17:57 CEST 2013

Duration: 87:29 min

Pages: 62

Example: Type Checking

Expression `*a[f(b->c)]+2`:



Equality of Types

Summary type checking:

- Choosing which rule to apply at an AST node is determined by the type of the child nodes
- ⇝ determining the rule requires a check for *equality* of types

type equality in C:

- **struct** `A {}` and **struct** `B {}` are considered to be different
  - ⇝ the compiler could re-order the fields of `A` and `B` independently (*not* allowed in C)
  - to extend an record `A` with more fields, it has to be embedded into another record:

```
typedef struct B {
          struct A a;
          int field_of_B;
        } extension_of_A;
```

- after issuing **typedef int** `C;` the types `C` and **int** are the same

Structural Type Equality

Alternative interpretation of type equality (*does not hold in C*):

semantically, two type t_1, t_2 can be considered as *equal* if the accept the same set of access paths.

Example:

```
struct list {
  int info;
  struct list* next;
}
```

```
struct list1 {
  int info;
  struct {
    int info;
    struct list1* next;
  }* next;
}
```

Consider declarations **struct** `list* l` and **struct** `list1* l`. Both allow

`l->info` `l->next->info`

but the two declarations of `l` have unequal types in C.

Algorithm for Testing Structural Equality

Idea:

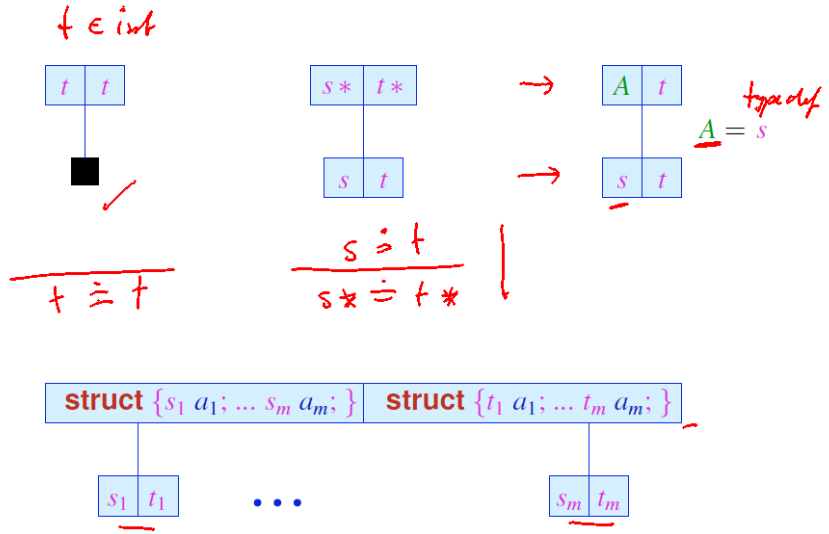
- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduces using type equalities of the form:

$$A = t$$

(we omit the Γ). Then define the following rules:

Rules for Well-Typedness



Example:

$$\begin{array}{rcl} A & = & \textbf{struct}\ \{\textbf{int}\ \text{info};\ A * \text{next};\} \\ B & = & \textbf{struct}\ \{\textbf{int}\ \text{info}; \\ & & \textbf{struct}\ \{\textbf{int}\ \text{info};\ B * \text{next};\} * \text{next};\} \end{array}$$

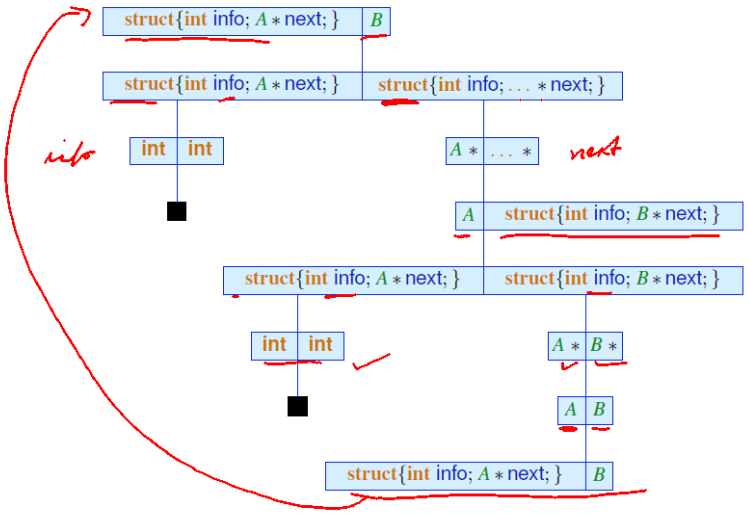
We ask, for instance, if the following equality holds:

$$\textbf{struct}\ \{\textbf{int}\ \text{info};\ A * \text{next};\} = B$$

We construct the following derivation tree:

Proof for the Example:

$$\begin{array}{rcl} A & = & \textbf{struct}\ \{\textbf{int}\ \text{info};\ A * \text{next};\} \\ B & = & \textbf{struct}\ \{\textbf{int}\ \text{info}; \\ & & \textbf{struct}\ \{\textbf{int}\ \text{info};\ B * \text{next};\} * \text{next};\} \end{array}$$



Implementation

We implement a function that implement the equivalence query for two types by applying the deduction rules:

- if no deduction rule applies, then the two types are *not equal*
- if the deduction rule for expanding a type definition applies, the function is called recursively with a *potentially larger* type
- during the construction of the proof tree, an equivalence query might occur several times
- in case an equivalence query appears a second time, the types are by definition equal

Implementation

We implement a function that implement the equivalence query for two types by applying the deduction rules:

- if no deduction rule applies, then the two types are *not equal*
- if the deduction rule for expanding a type definition applies, the function is called recursively with a *potentially larger* type
- during the construction of the proof tree, an equivalence query might occur several times
- in case an equivalence query appears a second time, the types are by definition equal

Termination?

- the set D of all declared types is finite
- there are no more than $|D|^2$ different equivalence queries
- repeated queries for the same inputs are are automatically satisfied

$\rightsquigarrow$ termination is ensured

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 (z.B. conversion from `int` to `float`)
- coersion is usually done towards more general types i.e. `5+0.5` has type `float` (since `float` $\geq$ `int`)

Coercion of Integer-Types in C: Promotion

C defines special conversion rules for integers: *promotion*

$$\texttt{unsigned char} \atop \texttt{signed char} \leq \texttt{unsigned short} \atop \texttt{signed short} \leq \texttt{int} \leq \texttt{unsigned int}$$

. . . 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*

$$\texttt{unsigned char} \atop \texttt{signed char} \leq \texttt{unsigned short} \atop \texttt{signed short} \leq \texttt{int} \leq \texttt{unsigned int}$$

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

subtle errors possible! Compute the character distribution

```
char* 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.

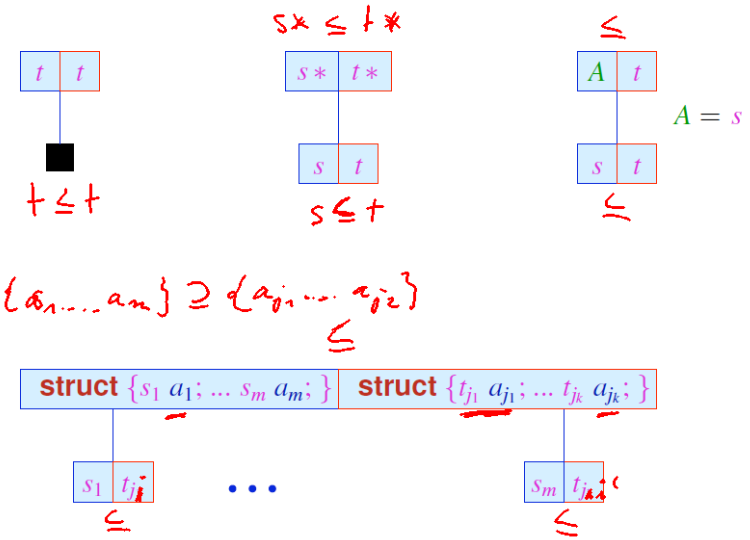
Example: Subtyping

Observe:

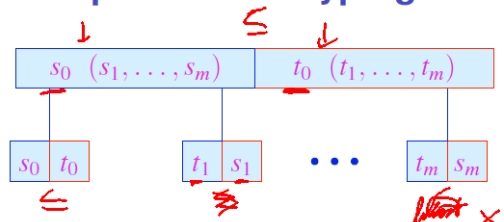
```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

- 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 is comparable to the question of when a sub-class can be passed-in (but more general)

Rules for Well-Typedness of Subtyping



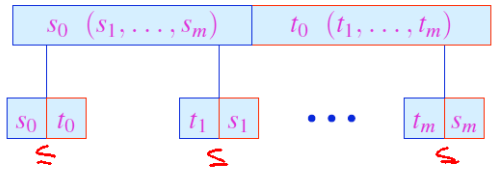
Rules and Examples for Subtyping



Examples:

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

Rules and Examples for Subtyping



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

Co- and Contra Variance

Definition

Given two function types in subtype relation $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ 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

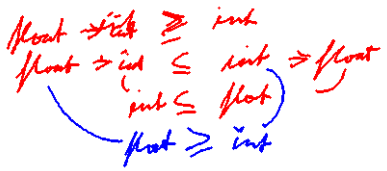
Definition

Given two function types in subtype relation $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ 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 function languages:

$$\textbf{int} \to (\textbf{float} \to \textbf{int}) \quad \leq \quad \textbf{int} \to (\textbf{int} \to \textbf{float})$$

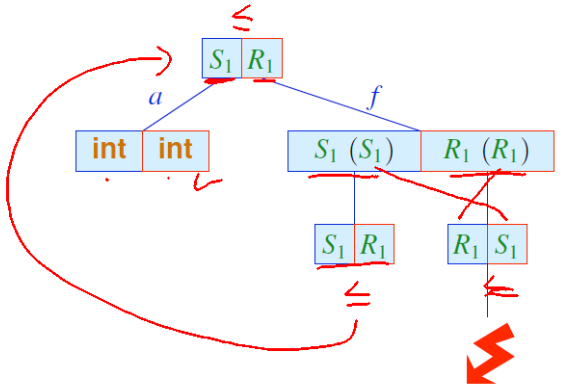


These rules can be applied directly to test for sub-type relationship of recursive types

Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

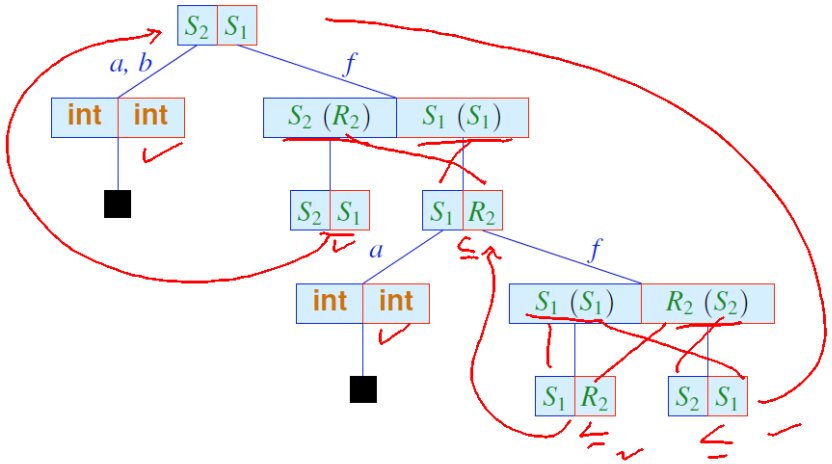
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\,(R_1)\,f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\,(S_1)\,f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\,(S_2)\,f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\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



Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

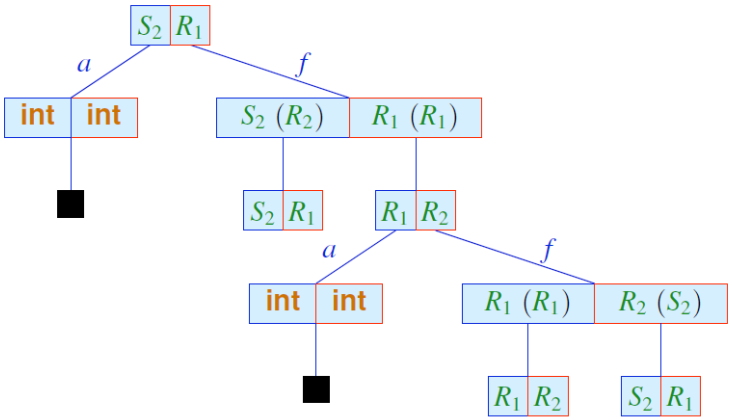
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\,(R_1)\,f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\,(S_1)\,f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\,(S_2)\,f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\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



Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

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



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)
- the semantic of the machine instructions

Chapter 1:
The Register C-Machine

The Register C-Machine (RCMa)

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 (RCMa)

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 Java virtual machine (JVM) is similar to the R-CMa but has no registers
- an interpreter of R-CMA can run on any platform

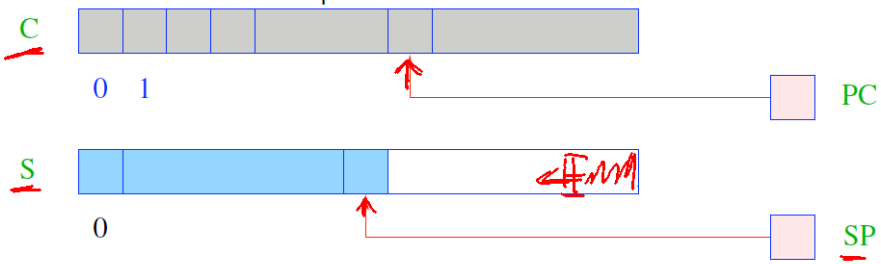
Virtual Machines

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:

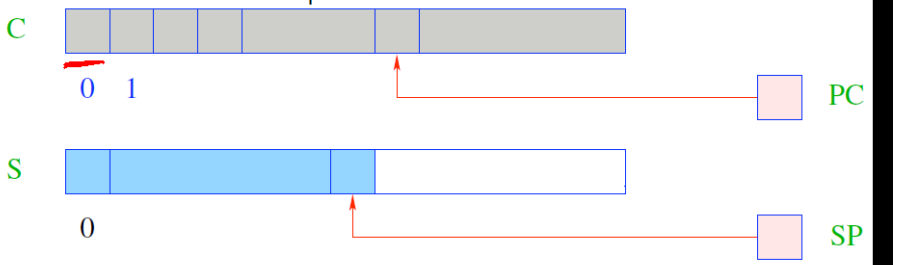


A virtual machine such as the JVM has the following structure:

- S: the data store – a memory region in which cells can be stored in LIFO order $\rightsquigarrow$ stack.
- SP: ($\widehat{=}$ stack pointer) pointer to the last used cell in S
- beyond S, the memory containing the heap follows

Components of a Virtual Machine

Consider Java as an example:



A virtual machine such as the JVM has the following structure:

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

Executing a Program

- the machine loads an instruction form C[PC] into an 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

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

- computes the value of the expression and
- stores it on top of the stack

Idea:

- first compute the value of the sub-expressions
- store the intermediate result on top of the stack
- apply the operator

Simple Expressions and Assignments

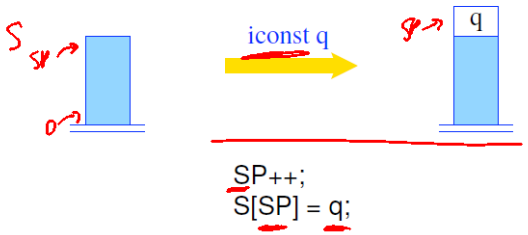
Task: evaluate the expression $(1 + 7) * 3$
that is, generate an instruction sequence that

- computes the value of the expression and
- stores it on top of the stack

General Principle

Evaluating an operation op($a_1, \ldots a_n$)

- the arguments $a_1, \ldots a_n$ must be on top of the stack
- the execution of the operation op consumes its arguments
- any resulting values are stored on top of the stack



the instruction iconst q puts the `int`-constant q onto the stack

Binary Operators

Operators with two arguments run as follows:

SP--;
S[SP] = S[SP] ∗ S[SP+1];

Binary Operators

Operators with two arguments run as follows:

SP--;
S[SP] = S[SP] ∗ S[SP+1];

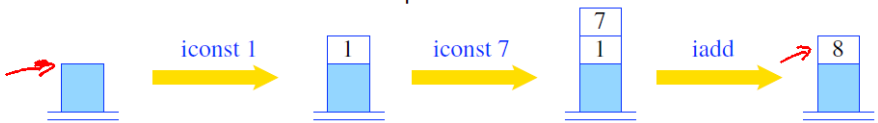
- imul expects two arguments on top of the stack, consumes them and puts the result on top of the stack

Composition of Instructions

Example: generate code for $1 + 7$:

iconst 1 iconst 7 iadd

Execution of this instruction sequence:



Composition of Instructions

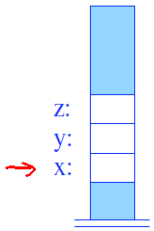
Example: generate code for $1 + 7$:

iconst 1 iconst 7 iadd

Execution of this instruction sequence:

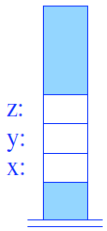
Expressions with Variables

Variables occupy a memory cell in S:



Expressions with Variables

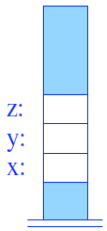
Variables occupy a memory cell in S:



- Associating addresses with variables can be done while creating the symbol table. The address is stored in any case at the *node of the declaration* of a variable.

Expressions with Variables

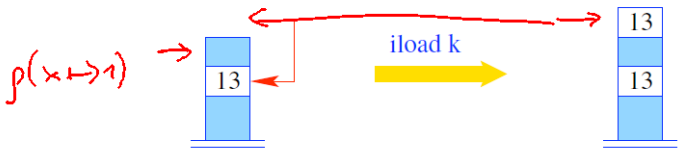
Variables occupy a memory cell in S:



- Associating addresses with variables can be done while creating the symbol table. The address is stored in any case at the *node of the declaration* of a variable.
- For each *use* of a variable, the address has to be looked up by inspecting its declaration node.
- in the sequel, we use a mathematical map ρ, that contains mappings form a variable x to the (relative) address of x; the map ρ is called *address environment* (or simply *environment*).

Reading from a Variable

The instruction iload k loads the value at address k, where k is *relative* to the top of the stack

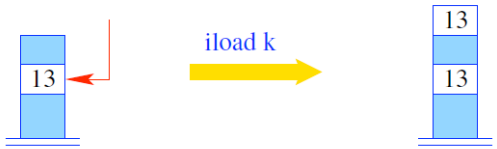


S[SP+1] = S[SP-k]; SP = SP+1;

Example: Compute $x + 2$ where $\rho = \{x \mapsto 1\}$:

Reading from a Variable

The instruction iload k loads the value at address k, where k is *relative* to the top of the stack



S[SP+1] = S[SP-k]; SP = SP+1;

Example: Compute $x + 2$ where $\rho = \{x \mapsto 1\}$:

```
iload 1
iconst 2
iadd
```


Chapter 3:
Generating Code for the Register C-Machine

Motivation for the Register C-Machine

A modern RISC processor features a fixed number of universal registers.

Motivation for the Register C-Machine

A modern RISC processor features a fixed number of universal registers.

- arithmetic operations can only use these registers as arguments
- access to memory are done via instructions to load and store to and from registers
- unlike the stack, registers have to be explicitly saved before a function is called

Motivation for the Register C-Machine

A modern RISC processor features a fixed number of universal registers.

- arithmetic operations can only use these registers as arguments
- access to memory are done via instructions to load and store to and from registers
- unlike the stack, registers have to be explicitly saved before a function is called

A translation for a RISC processor must therefore:

1. store variables and function arguments in registers
2. save the content of registers *onto the stack* before calling a function
3. express any arbitrary computation using *finitely* many registers

Motivation for the Register C-Machine

A modern RISC processor features a fixed number of universal registers.

- arithmetic operations can only use these registers as arguments
- access to memory are done via instructions to load and store to and from registers
- unlike the stack, registers have to be explicitly saved before a function is called

A translation for a RISC processor must therefore:

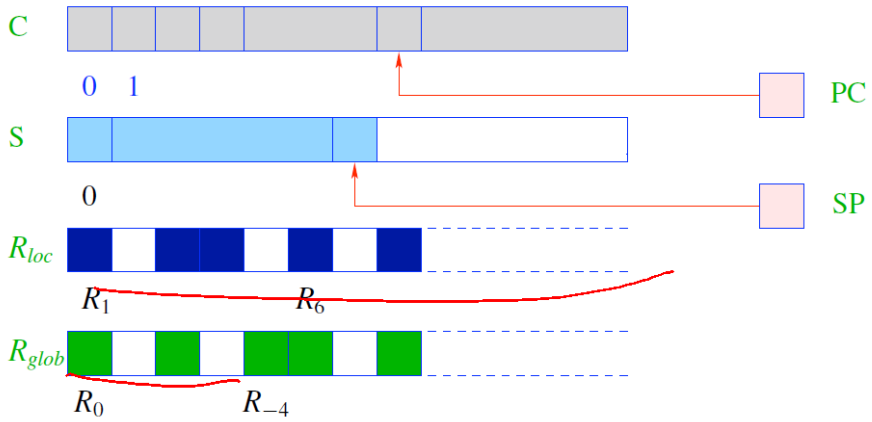
1. store variables and function arguments in registers
2. save the content of registers *onto the stack* before calling a function
3. express any arbitrary computation using *finitely* many registers

$\rightsquigarrow$ only consider the first two problems (and deal with the other two later)

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

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

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

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
loadc R_i c	$R_i = c$	load constant
move R_i 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
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 $\rho\ x = a$):

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

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 $\rho\ x = a$):

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

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:

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

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:

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

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

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

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, $i = 4$.

$$
\begin{aligned}
\text{code}^4\ \texttt{x=y+z*3}\ \rho &= \text{code}_R^4\ \texttt{y+z*3}\ \rho \\
&\quad \text{move}\ R_1\ R_4 \\
\text{code}_R^4\ \texttt{y+z*3}\ \rho &= \text{move}\ R_4\ R_2 \\
&\quad \text{code}_R^5\ \texttt{z*3}\ \rho \\
&\quad \text{add}\ R_4\ R_4\ R_5 \\
\text{code}_R^5\ \texttt{z*3}\ \rho &= \text{move}\ R_5\ R_3 \\
&\quad \text{code}_R^6\ \texttt{3}\ \rho \\
&\quad \text{mul}\ R_5\ R_5\ R_6
\end{aligned}
$$

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, $i = 4$.

$$
\begin{aligned}
\text{code}^4\ \texttt{x=y+z*3}\ \rho &= \text{code}_R^4\ \texttt{y+z*3}\ \rho \\
&\quad \text{move}\ R_1\ R_4 \\
\text{code}_R^4\ \texttt{y+z*3}\ \rho &= \text{move}\ R_4\ R_2 \\
&\quad \text{code}_R^5\ \texttt{z*3}\ \rho \\
&\quad \text{add}\ R_4\ R_4\ R_5 \\
\text{code}_R^5\ \texttt{z*3}\ \rho &= \text{move}\ R_5\ R_3 \\
&\quad \text{code}_R^6\ \texttt{3}\ \rho \\
&\quad \text{mul}\ R_5\ R_5\ R_6 \\
\text{code}_R^6\ \texttt{3}\ \rho &= \text{loadc}\ R_6\ 3
\end{aligned}
$$

$\rightsquigarrow$ the assignment `x=y+z*3` is translated as

$\text{move}\ R_4\ R_2; \text{move}\ R_5\ R_3; \text{loadc}\ R_6\ 3; \text{mul}\ R_5\ R_5\ R_6; \text{add}\ R_4\ R_4\ R_5; \text{move}\ R_1\ R_4$

Managing Temporary Registers

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

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

where

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

we obtain

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