**Equality of Types**

Summary type checking:
- Choosing which rule to apply at an AST node is determined by the type of the child nodes
- \(\sim\) determining the rule requires a check for equality of types

_type equality_ in C:
- **struct **A** and struct **B** are considered to be different**
  - \(\sim\) 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:
    ```c
    typedef struct { 
    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:**

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

Consider declarations **struct list** \(l\) and **struct list1** \(l\)**. Both allow

\[ l->info \quad l->next->info \]

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 introduced using type equalities of the form:

\[ A = t \]

(we omit the \( \Gamma \)). Then define the following rules:

Example:

\[ \frac{A = \text{struct } \{ \text{int info}; \ A \ast \text{next}; \} \quad B = \text{struct } \{ \text{int info}; \ B \ast \text{next}; \} \ast \text{next}; \} }{A = B} \]

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

\[ \text{struct } \{ \text{int info}; \ A \ast \text{next}; \} = B \]

We construct the following derivation tree:

Proof for the Example:
Implementation

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

- if the 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 automatically satisfied

$\sim$ 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

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`)
- coercion 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

\[
\begin{align*}
\text{unsigned char} & \leq \text{unsigned short} \\
\text{signed char} & \leq \text{signed short} \\
\text{int} & \leq \text{unsigned int}
\end{align*}
\]

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

subtle errors possible! Compute the character distribution

```c
char str;
char str = "...";
int dist[256];
memset(dist, 0, sizeof(dist));
while (*str) {
    dist[*(unsigned int)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 \( t_1 \)
  - form a subset of the values of type \( t_2 \);
  - can be converted into a value of type \( t_2 \);
  - fulfill the requirements of type \( t_2 \).

Example: Subtyping

Observe:

```c
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

\[ t \leq t \]
\[ s \times \leq t \]
\[ s + t \leq \]
\[ A \leq \]
\[ A = s \]

\( \{ a_1, \ldots, a_m \} \vartriangleleft \{ a_{i_1}, \ldots, a_{i_k} \} \)

\[ \text{struct } \{ s_1, a_{1i}; \ldots, s_m, a_{mi}; \} \vartriangleleft \text{struct } \{ t_{j_1}, a_{ji}; \ldots, t_{j_k}, a_{ki}; \} \]

\[ s_1 \leq \]
\[ s_1 \leq \]
\[ s_m \leq \]

Examples:

\[ \text{struct } \{ \text{int } a; \text{ int } b; \} \]
\[ \text{int } (\text{int}) \leq \text{float } (\text{float}) \]

Rules and Examples for Subtyping

\[ S_0 \{ s_1, \ldots, s_m \} \leq \]
\[ T_0 \{ t_1, \ldots, t_m \} \leq \]

\[ S_0 \leq \]
\[ T_0 \leq \]

Examples:

\[ \text{struct } \{ \text{int } a; \text{ int } b; \} \leq \text{struct } \{ \text{float } a; \} \]
\[ \text{int } (\text{int}) \leq \text{float } (\text{float}) \]
\[ \text{int } (\text{float}) \leq \text{float } (\text{int}) \]

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-variace** of the return type \( S_0 \leq T_0 \) and
- **contra-variance** of the arguments \( s_i \geq t_i \) for \( 1 < i \leq n \)

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-\textit{variance} of the return type \( s_0 \leq t_0 \) and
- contra-\textit{variance} of the arguments \( s_i \geq t_i \) für \( 1 \leq i \leq n \)

Example from function languages:
\[
\text{int} \rightarrow \text{(float } \rightarrow \text{int)} \leq \text{int} \rightarrow \text{(int } \rightarrow \text{float)}
\]

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

Subtypes: Application of Rules (I)
Check if \( S_1 \leq R_1 \):
\[
\begin{align*}
R_1 &= \text{struct } \{ \text{int a; } R_1(R_1)f; \} \\
S_1 &= \text{struct } \{ \text{int a; int b; } S_1(S_1)f; \} \\
R_2 &= \text{struct } \{ \text{int a; } R_2(S_2)f; \} \\
S_2 &= \text{struct } \{ \text{int a; int b; } S_2(R_2)f; \}
\end{align*}
\]

Subtypes: Application of Rules (II)
Check if \( S_2 \leq S_1 \):
\[
\begin{align*}
R_1 &= \text{struct } \{ \text{int a; } R_1(R_1)f; \} \\
S_1 &= \text{struct } \{ \text{int a; int b; } S_1(S_1)f; \} \\
R_2 &= \text{struct } \{ \text{int a; } R_2(S_2)f; \} \\
S_2 &= \text{struct } \{ \text{int a; int b; } S_2(R_2)f; \}
\end{align*}
\]

Subtypes: Application of Rules (III)
Check if \( S_2 \leq R_1 \):
\[
\begin{align*}
R_1 &= \text{struct } \{ \text{int a; } R_1(R_1)f; \} \\
S_1 &= \text{struct } \{ \text{int a; int b; } S_1(S_1)f; \} \\
R_2 &= \text{struct } \{ \text{int a; } R_2(S_2)f; \} \\
S_2 &= \text{struct } \{ \text{int a; int b; } S_2(R_2)f; \}
\end{align*}
\]
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

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 machine 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 \( \sim \) stack.
- SP, (\( \equiv \) stack pointer) pointer to the last used cell in S
- beyond S, the memory containing the heap follows

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 \( \sim \) stack.
- SP, (\( \equiv \) 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, (\( \equiv \) 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(\text{PC})} into an instruction register IR in order to execute it.
- before evaluating the instruction, the PC is incremented by one.

```c
while (true) {
    IR = C(\text{PC}); \text{PC}++;
    \text{execute (IR)};
}
```

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

Simple Expressions and Assignments

Task: evaluate the expression \((1 + 7) \times 3\)
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

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 \textit{iconst q} puts the int-constant \(q\) onto the stack.
Binary Operators

Operators with two arguments run as follows:

\[
\begin{align*}
&3 & 8 & \quad \text{imul} \\
&\to & \quad \text{SP--;} \\
& & \quad \text{S[SP]} = \text{S[SP]} + \text{S[SP+1]}; \\
&24 & \\
\end{align*}
\]

- `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 \(1 + 7\):

\[
\begin{align*}
&\text{iconst 1} & \quad \text{iconst 7} & \quad \text{iadd} \\
\end{align*}
\]

Execution of this instruction sequence:

\[
\begin{align*}
&\text{iconst 1} & \quad \text{iconst 7} & \quad \text{iadd} \\
&1 & \quad 7 & \quad 8 \\
\end{align*}
\]
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 $\rho$ that contains mappings from a variable $v$ to the (relative) address of $v$; the map $\rho$ is called address environment (or simply environment).

Expressions with Variables

- 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.

Reading from a Variable

The instruction $iload$ 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 \( \text{iload} \ k \) 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 \( \rho = \{ x \mapsto 1 \} \):

\[ \text{iload} 1 \]
\[ \text{iconst} 2 \]
\[ \text{tadd} \]

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

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 $R_1, R_2, \ldots, R_l, \ldots$
- global register are $R_0, R_{-1}, \ldots, R_2, \ldots$  $\dot{l} < 0$

The Register Sets of the R-CMA

The two register sets have the following purpose:
- the local registers $R_l$
  - 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. **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. **Global registers** $R_j$
   - 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. **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. **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:

\[ p = \{ \times \leftarrow R_i \} \]

**Instruction**

- load $R_i\ c$
- move $R_i\ R_j$
- $R_i = c$
- $R_i = R_j$
- copy $R_j$ to $R_i$

**Intuition**

- registers $R_i$ 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load e R_i</code></td>
<td>$R_i = e$</td>
<td>load constant</td>
</tr>
<tr>
<td><code>move R_i R_j</code></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 $\rho x = \{ c \}$):

$$
\text{code}_{R} c \rho = \begin{cases} 
\text{load}_{R} R_i c & \text{if } c \in \mathbb{Z} \\
\text{move}_{R} R_i R_j & \text{otherwise}
\end{cases} \\
\text{code}_{R} x \rho = \text{code}_{R} e \rho \\
\text{code}_{R} x = e \rho = \text{code}_{R} e \rho \\
\text{move}_{R} R_i R_j \\

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

Translation of Expressions

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

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

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

Correspondingly, we generate code as follows:

$$
\text{code}_{R} e_1 \mathsf{op} e_2 \rho = \begin{cases} 
\text{code}_{R} e_1 \rho & \text{if } e_1 \in \mathbb{Z} \\
\text{code}_{R} e_2 \rho & \text{if } e_2 \in \mathbb{Z} \\
\text{code}_{R} \mathsf{op} R_i R_j R_k & \text{otherwise}
\end{cases}
$$

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

$$
\text{code}_{R} 3 \times 4 \rho = \begin{cases} 
\text{code}_{R} 3 \rho & \text{if } 3 \in \mathbb{Z} \\
\text{code}_{R} 4 \rho & \text{if } 4 \in \mathbb{Z} \\
\text{mul}_{R} R_i R_j R_k & \text{otherwise}
\end{cases}
$$
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, $i = 4$.

```
code_4 x=y+z+3 \rho = \ CODE_4 y+z+3 \rho \\
move R_1 R_4
```

```
code_4 y+z+3 \rho = \ CODE_4 R_3 R_4
```

```
code_4 z+3 \rho = \ CODE_4 R_5 R_6
```

```
code_4 3 \rho = \ CODE_4 R_0 R_3
```

```
\sim the assignment x = y+z+3 is translated as
```
move R_4 R_5; move R_5 R_6; load R_6; mul R_5 R_6 R_7; add R_4 R_6
```


Managing Temporary Registers

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

```
code_5 3*4 + 3*4 \rho = \ CODE_5 3*4 \rho \\
CODE_5 3*4 \rho \\
add R_4 R_5
```

where

```
code_5 3*4 \rho = \ CODE_5 R_6 R_7
```

we obtain

```
code_4 3*4 + 3*4 \rho = \ CODE_4 R_0 R_3
```