Type Systems for C-like Languages

More rules for typing an expression:

Array:

\[
\begin{align*}
\Gamma & \vdash e_1 : \text{int} \\
\Gamma & \vdash e_2 : \text{int}
\end{align*}
\]

\[
\frac{\Gamma \vdash e_1 : [\text{int}]}{\Gamma \vdash e_1[e_2] : \text{int}}
\]

Struct:

\[
\Gamma \vdash e : \text{struct} \{ t_1, a_1, \ldots, t_m, a_m; \}
\]

\[
\frac{\Gamma \vdash e.a_i : t_i}{\Gamma \vdash e.a_i : t_i}
\]

App:

\[
\begin{align*}
\frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash e : \text{int}}
\end{align*}
\]

\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}}
\]

Explicit Cast:

\[
\frac{\Gamma \vdash e : t_1 \quad t_1 \text{ can be converted to } t_2}{\Gamma \vdash (t_1) e : t_2}
\]

Equality of Types

Summary of 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 \(\sim\) 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:

  \[
  \text{struct}\ B\ \{\ \\
  \quad \text{struct}\ A;\ \\
  \quad \text{int}\ \text{field}\_of\_B;\ \\
  \quad \text{extension}\_of\_A;\ \\
  \}
  \]

- 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 types \(t_1, t_2\) can be considered as equal if they accept the same set of access paths.

Example:

\[
\begin{align*}
\text{struct}\ list\ \{\ \\
\quad \text{int}\ info;\ \\
\quad \text{struct}\ list\\ast\ next;\ \\
\}
\end{align*}
\]

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

\[
l->\text{info}\ l->\text{next}->\text{info}
\]

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 definitions:

\[
\text{typedef } A t
\]

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

Example:

\[
\begin{align*}
\text{typedef } & \quad \text{struct } \{ \text{int info; } A \ast \text{next; } \} \\
\text{typedef } & \quad \text{struct } \{ \text{int info; } B \ast \text{next; } \} \ast \text{next } \} \quad \text{\(A\)} \\
\text{\(B\)}
\end{align*}
\]

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

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

We construct the following deduction tree:

\[
\text{struct } \{ \text{int info; } A \ast \text{next; } \} \quad \text{\(A\)} \\
\text{struct } \{ \text{int info; } B \ast \text{next; } \} \ast \text{next } \} \quad \text{\(B\)}
\]

Proof for the Example:
Implementation

We implement a function that implements 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
- in case an equivalence query appears a second time, the types are equal by definition

Example: Subtyping

Extending the subtype relationship to more complex types, observe:

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

- we want `extractInfo` to be applicable to all argument structures that return a string typed field for accessor `info`
- the idea of subtyping on values is related to subclasses
- we use deduction rules to describe when $t_1 \leq t_2$ should hold...

Subtypes

On the arithmetic basic types `char`, `int`, `long`, etc. there exists a rich subtype hierarchy

- $t_1 \leq t_2$, 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 $t_2$;
- are assignable to variables of type $t_2$.

\[
\frac{t_i \leq t_j \quad t_j \leq t_k}{t_i \leq t_k}
\]

Rules for Well-Typedness of Subtyping

```c
struct { s_1 a_1 ... s_j a_j; } 
struct { t_1 a_1 ... t_k a_k; } 
struct { int u, int v; } 
struct { int u; } 
struct { int u; }
```

```c
\frac{t \leq t'} {s * t *} 
\frac{s t} {s t} 
\frac{A t} {\text{typedef } s A}
```
Rules and Examples for Subtyping

Examples:

\[
\begin{align*}
\text{struct } \{ \text{int } a; \text{ int } b; \} \\
\text{int} (\text{int}) \\
\text{int} (\text{float}) \\
\Downarrow f \\
\text{float} (\text{float}) \\
\text{float} (\text{int})
\end{align*}
\]

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 \( 1 < i < n \)

Subtypes: Application of Rules (I)

Check if \( S_1 \leq R_1 \):

\[
\begin{align*}
R_1 &= \text{struct } \{ \text{int } a; R_1(R_1) f; \} \\
S_1 &= \text{struct } \{ \text{int } a; \text{int } b; S_1(S_1) f; \} \\
R_2 &= \text{struct } \{ \text{int } a; R_2(S_2) f; \} \\
S_2 &= \text{struct } \{ \text{int } a; \text{int } b; S_2(R_2) f; \}
\end{align*}
\]

Subtypes: Application of Rules (II)

Check if \( S_2 \leq S_1 \):

\[
\begin{align*}
R_1 &= \text{struct } \{ \text{int } a; R_1(R_1) f; \} \\
S_1 &= \text{struct } \{ \text{int } a; \text{int } b; S_1(S_1) f; \} \\
R_2 &= \text{struct } \{ \text{int } a; R_2(S_2) f; \} \\
S_2 &= \text{struct } \{ \text{int } a; \text{int } b; S_2(R_2) f; \}
\end{align*}
\]
Discussion

- for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree
- structural sub-types are very powerful and can be quite intricate to understand
- **Java** generalizes structs to objects/classes where a sub-class $A$ inheriting from base class $O$ is a subtype $A \subseteq O$
- subtype relations between classes must be explicitly declared

Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

- $R_1 = \text{struct\{int\ a; \ R_1(R_1)\ f\;\}}$
- $S_1 = \text{struct\{int\ a; \ int\ b; \ S_1(S_1)\ f\;\}}$
- $R_2 = \text{struct\{int\ a; \ R_2(S_2)\ f\;\}}$
- $S_2 = \text{struct\{int\ a; \ int\ b; \ S_2(R_2)\ f\;\}}$

Code Synthesis

Chapter 1:
The Register C-Machine

for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree
- structural sub-types are very powerful and can be quite intricate to understand
- **Java** generalizes structs to objects/classes where a sub-class $A$ inheriting from base class $O$ is a subtype $A \subseteq O$
- subtype relations between classes must be explicitly declared
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

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 Dalvik VM 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 follows the memory containing the heap

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

```java
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
Chapter 2:
Generating Code for the Register C-Machine