Type Expressions

Types are given using type-expressions. The set of type expressions $\Gamma$ contains:

- base types: int, char, float, void, ...
- type constructors that can be applied to other types

Type Checking

Problem:

Given: A set of type declarations $\Gamma = \{ t_1, x_1; \ldots; t_n, x_n \}$

Check: Can an expression $e$ be given the type $t$?
**Type Checking using the Syntax Tree**

Check the expression `a \{ f(b>c) \} + 2`:

```
+  
|   
|   a  
|     +  
|     |   
|     |   a 
|     |   +  
|     |   |   
|     |   2 
```

**Idea:**
- Traverse the syntax tree bottom-up.
- For each identifier, lookup its type in `\Gamma`.
- Constants such as 2 or 0.5 have a fixed type.
- The types of the inner nodes of the tree are deduced using typing rules.

---

**Type Systems**

Formally, consider judgements of the form:

```
\Gamma \vdash e : t
```

// In the type environment `\Gamma` the expression `e` has type `t`.

**Axioms:**

- **Const:** \( \Gamma \vdash c : t_c \) (\( t_c \) type of constant `c`)
- **Var:** \( \Gamma \vdash x : \Gamma(x) \) (\( x \) Variable)

**Rules:**

**Ref:**

```
\Gamma \vdash e : t \\
\Gamma \vdash \& e : t* \\
\Gamma \vdash e : \Gamma \\
```

**Def:**

```
\Gamma \vdash e : t+ \\
\Gamma \vdash e : \Gamma \\
```

---

**Type Systems for C-like Languages**

More rules for typing an expression:

**Array:**

```
\Gamma \vdash e_1 : t* \\
\Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e[e_2] : t \\
```

**Array:**

```
\Gamma \vdash e_1 : t[] \\
\Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e[e_2] : t \\
```

**Struct:**

```
\Gamma \vdash e : \text{struct} \{ l_1, a_1; \ldots, l_m, a_m \} \\
```

**App:**

```
\Gamma \vdash e : t \{ l_1, \ldots, l_m \} \\
\Gamma \vdash e_1 : t_1 \\
\ldots \\
\Gamma \vdash e_m : t_m \\
```

**Op:**

```
\Gamma \vdash e_1 : t_1 \\
\Gamma \vdash e_2 : \text{int} \\
```

**Explicit Cast:**

```
\Gamma \vdash e : t_1 \\
\Gamma \vdash (\text{int}) e : t_2 \\
```

---

**Example: Type Checking**

Given expression `a \{ f(b>c) \} + 2`:

```
+  
|   
|   a  
|     +  
|     |   
|     |   a 
|     |   +  
|     |   |   
|     |   2 
```

```
int * [ ] 
```

```
int (struct list *)  
```

```
struct {struct list * c;}  
```

---
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 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:
    `struct B {
    struct A;
    int field_of_B;
    | extension_of_A;
    }` after issuing `typedef int C;` the types `C` and `int` are the same

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 set of simpler type expressions

Suppose that recursive types were introduced using type definitions:

```
typedef A f
```

(we omit the `f`). Then define the following rules:

Structural Type Equality

Alternative interpretation of type equality (does not hold in C):
- Semantically, two types `T1, T2` can be considered as equal if they accept the same set of access paths.

Example:
```
struct list {
    int info;
    struct list* next;
}
```
```
struct list1 {
    int info;
    struct list1* 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.

Rules for Well-Typedness
Example:

typedef struct {int info; A * next;} A
typedef struct {int info; struct {int info; B * next;} * next;} B

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

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

We construct the following deduction tree:

Overloading and Coercion

Some operators such as \( + \) are overloaded:
- \( + \) has several possible types
  - for example: \( \text{int} + (\text{int}, \text{int}), \text{float} + (\text{float}, \text{float}) \)
  - but also \( \text{float} + (\text{float}, \text{int}), \text{int} + (\text{int}, \text{int}) \)
- depending on the type, the operator \( + \) has a different implementation
- determining which implementation should be used is based on the type of the \textit{arguments} only
Subtypes

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

Subtypes

$t_1 \leq t_2$, 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 $t_2$.

Rules for Well-Typedness of Subtyping

- $t \leq t'$
- $s \times t \leq A \times A$
- $s \leq s$
- $s \leq t$

Example: Subtyping

Extending the subtype relationship to more complex types, observe:

```c
string extractInfo(string info; ) {
    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...

Rules and Examples for Subtyping

- $s_0 \times (s_1, \ldots, s_m)$
- $t_0 \times (t_1, \ldots, t_m)$
- $s_j \leq t_j$
- $s_0 \leq t_0$
- $\cdots$
- $s_m \leq t_m$

Examples:

- `struct {int a; int b;}`
- `struct {float a;}`
- `int (int)`
- `float (float)`
Rules and Examples for Subtyping

\[ s_0 (s_1, \ldots, s_m), t_0 (t_1, \ldots, t_n) \]

Examples:

- `struct { int a; int b; }`
- `struct { float a; }`
- `int (int)`
- `float (float)`
- `float (int)`

Definition

Given two function types in subtype relation `s_0 (s_1, \ldots, s_m) \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`, for `1 < i \leq n`

Subtypes: Application of Rules (I)

Check if `S_2 \leq S_1`:

\[
\begin{align*}
R_1 &= \text{struct } \{ \text{int a; b; } S_1 (R_1) \ f_1 \} \\
S_1 &= \text{struct } \{ \text{int a; int b; } S_1 (S_1) \ f_1 \} \\
R_2 &= \text{struct } \{ \text{int a; b; } S_2 (R_2) \ f_1 \} \\
S_2 &= \text{struct } \{ \text{int a; int b; } S_2 (S_2) \ f_1 \}
\end{align*}
\]

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 records to objects/classes where a sub-class `A` inheriting form base class `O` is a subtype `A \leq O`.
- Subtype relations between classes must be explicitly declared.
- Inheritance ensures that all sub-classes contain all (visible) components of the super class.
- A shadowed (overwritten) component in `A` must have a subtype of the the component in `O`.
- Java does not allow argument subtyping for methods since it uses different signatures for overloaded.
Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

$$R_1 = \text{struct} \{ \text{int } a, R_1(R_1) f_1 \}$$
$$S_1 = \text{struct} \{ \text{int } a, \text{int } b, R_1(S_1) f_1 \}$$
$$S_2 = \text{struct} \{ \text{int } a, \text{int } b, S_2(R_2) f_1 \}$$

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 records to objects/classes where a sub-class $A$ inheriting form base class $O$ is a subtype $A \leq O$
- subtype relations between classes must be explicitly declared
- inheritance ensures that all sub-classes contain all (visible) components of the super class
- a shadowed (overwritten) component in $A$ must have a subtype of the the component in $O$
- Java does not allow argument subtyping for methods since it uses different signatures for overloading

Topic:

Code Synthesis