Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. 
for example: `int, void*, struct { int x; int y; }`

Types are useful to
- manage memory
- to avoid certain run-time errors

In imperative and object-oriented programming languages a declaration has to specify a type. The compiler then checks for a type correct use of the declared entity.

Type Expressions

Types are given using type expressions.
The set of type expressions $T$ contains:
- base types: `int, char, float, void, ...`
- type constructors that can be applied to other types

example for type constructors in C:
- records: `struct [ $t_1, ..., t_n ]`
- pointer: `$t *$
- arrays: `$t [ ]`
  - the size of an array can be specified
  - the variable to be declared is written between $t$ and $[n]$
- functions: `$t (t_1, ..., t_n)`
  - the variable to be declared is written between $t$ and $(t_1, ..., t_n)$
  - in ML function types are written as: `$t_1 \ldots \rightarrow t$`
Type Definitions in C

A type definition is a synonym for a type expression. In C they are introduced using the `typedef` keyword.

Type definitions are useful:
- as abbreviation:
  ```
  typedef struct { int x; int y; } point_t;
  ```
- to construct recursive types:
  ```
  struct list { int info; struct list* next; }
  ```

Possible declaration in C:
```
struct list {
   typedef struct list list_t;
   int info;
   struct list* next;
}
```  
more readable:
```
struct list { int info; struct list* next; }
```  
```
struct list* head;  list_t* head;
```

Type Checking

Problem:
Given: a set of type declarations $\Gamma = \{ t_1, x_1; \ldots ; t_m, x_m; \}$
Check: Can an expression $e$ be given the type $t$?

Example:
```
struct list { int info; struct list* next; };
int f(struct list* l) { return l; };
struct ( struct list* c )* b;
int* a[11];
```

Consider the expression:
```
+a[f(b->c)]+2;
```

Type Checking using the Syntax Tree

Check the expression $*a[f(b->c)]+2$:

```
+  2
|   |
| a |\(f\) |
| \(a\) | b |
```

Idea:
- traverse the syntax tree bottom-up
- for each identifier, we 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

Formal consider judgement 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: $\Gamma \vdash e : t \quad \Gamma \vdash k e : t*$
- Der: $\Gamma \vdash e : t$
Type Systems for C-like Languages

More rules for typing an expression:

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

Array:

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

Array:

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

Struct:

\[ \frac{\Gamma \vdash e : \text{struct \{ \text{a}_1; \ldots; \text{a}_m; \}} \quad \Gamma \vdash e[a_1] : t_1 \quad \ldots \quad \Gamma \vdash e[a_m] : t_m }{ \Gamma \vdash e : \text{int} } \]

App:

\[ \frac{\Gamma \vdash e : \text{int} \quad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m }{ \Gamma \vdash (e_1 \ldots e_m) : t } \]

Op:

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

Cast:

\[ \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t_2 }{ \Gamma \vdash (e_2) e : t_2 } \quad \text{can be converted to } t_2 \]

Example: Type Checking

Given expression \( *a[f(b->c)] + 2 \) and \( \Gamma = \{ \)
- struct list \{ int info; struct list* next; \};
- int f(struct list* l);
- struct / struct list* c;* b;
- int* a[11];

\( \Gamma \vdash \begin{array}{l}
* f(b) \\
* a[f(b->c)] + 2
\end{array} \)

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:

- \( \sim \) struct \( A \{ \} \) and \( \sim \) 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:

\[ \text{typedef struct } B \{ \]
- \( \text{struct A d; } \)
- \( \text{int field of } B; \)
- \} extension of A; \]
- After issuing \( \text{typedef int C; } \) the types \( C \) and \( \text{int} \) 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 they accept the same set of access paths.

Example:

```c
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

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

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:

---

Rules for Well-Typedness

\[ \begin{align*}
    & t, t \\
    & s, t * \\
    & \downarrow \quad \text{fun} \quad \text{let} \\
    & t, s \\
    & s, t \\
\end{align*} \]

\[ A = t \]

\[ A = s \]

---

Example:

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

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

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

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

We construct the following derivation tree:
Proof for the Example:

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

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
- 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

\[ \sim \text{ termination is ensured} \]

Overloading and Coercion

Some operators such as \( + \) are **overloaded**:

- \( + \) has **several possible** types
  - for example: \( \text{int } + (\text{int, int}), \text{float } + (\text{float, float}) \)
  - but also \( \text{float} + (\text{float, float}), \text{int} + (\text{int, int}) \)
- 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 \text{int} and \text{float}.

- instead of defining \( + \) for all possible combinations of types, the arguments are automatically **coerced**
- this coercion may generate code (z.B. conversion from \text{int} to \text{float})
- coercion is usually done towards more **general** types i.e. \( 5 + 0.5 \) has type \text{float} (since \text{float} \geq \text{int})

Coercion of Integer-Types in C: Promotion

C defines special conversion rules for integers: **promotion**

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

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

\[ \sim \text{ where a conversion has to happen via all intermediate types:} \]

subtle errors possible! Compute the character distribution of \text{str}:

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

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

- on the arithmetic basic types \texttt{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 \( t_2 \)

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

\[
\begin{align*}
& \text{int} \leq \text{double} \\
& x : t_1 \\
& y : t_2 \\
& y = x ;
\end{align*}
\]

Example: Subtyping

Observe:

\[
\text{string extractInfo} \left( \text{struct} \{ \text{string info} ; \} x \right) \{ \\
\quad \text{return} \ x . \text{info} ; \ W \}
\]

- we would like \texttt{extractInfo} to be applicable to all argument records that contain a field \texttt{string info}
- use deduction rules to describe when \( n_1 \leq t_2 \) should hold
- the idea of subtyping on values is related to subtyping as implemented in object-oriented languages

\[
\begin{align*}
& t_1 \leq t_2 \\
& \text{int} \leq \text{double} \\
& y = x ;
\end{align*}
\]

Rules for Well-Typedness of Subtyping

\[
\begin{array}{c}
\text{struct} \{ s_j, a_{j1} \cdots s_j, a_{j_k} ; \} \leq \text{struct} \{ t_j, a_{j1} \cdots t_j, a_{j_k} ; \} \\
\end{array}
\]

Rules and Examples for Subtyping

\[
\begin{array}{c}
\text{struct} \{ \text{int} a ; \text{int} b ; \} \leq \text{struct} \{ \text{float} a ; \} \\
\text{int} (\text{int}) \leq \text{float} (\text{float}) \\
\text{int} (\text{float}) \leq \text{float} (\text{int})
\end{array}
\]

Examples:

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 \( 1 < i \leq n \)

Example from function languages:
\[
\begin{align*}
\text{int} \rightarrow (\text{float} \rightarrow \text{int}) & \leq \text{int} \rightarrow (\text{int} \rightarrow \text{float}) \\
\Leftrightarrow \text{int} & \leq \text{int} \\
(\text{float} \rightarrow \text{int}) & \leq (\text{int} \rightarrow \text{float}) \\
\Leftrightarrow \text{int} & \leq \text{float} \land \text{int} \leq \text{float}
\end{align*}
\]

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

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

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

Subtypes: Application of Rules (III)
Check if \( S_2 \leq R_1 \):
\[
R_1 = \text{struct } \{ \text{int } a; R_1(R_1) f; \}
S_1 = \text{struct } \{ \text{int } a; \text{int } b; S_1(S_1) f; \}
R_2 = \text{struct } \{ \text{int } a; R_2(S_2) f; \}
S_2 = \text{struct } \{ \text{int } a; \text{int } b; S_2(R_2) f; \}
\]
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 from 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 component in $O$
- Java does not allow argument subtyping for methods since it uses different signatures for overloading