Types

Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris
year 2015–2016

Course 5
11 March 2016
Purpose of typing:

- avoid errors during the execution of programs, by restricting the programs accepted at compilation time;
- help the compiler produce efficient code;
- document properties of programs.

In this course, we look at typing from a formal and semantic view:
what semantics can we give to types and typing?
what semantic information is guaranteed by types?

We don’t discuss:
- type systems in language design and implementation
- relations between type theory and proof theory
- type theory as an alternative to set theory
**Type:** set of values with a specific machine representation.

often, distinct types denote non-overlapping value sets, but this is not always the case:
e.g., short/int/long in C, or subtyping Java and C++

Variables are assigned a type that defines its possible values.

**Static vs. dynamic typing:**

- **static:** the type of each variable is known at compile time
  (C, Java, OCaml)

- **dynamic:** the type of each variable is discovered during the execution and may change
  (Python, Javascript)
Strongly vs. loosely typed languages:

- **loose**: typing does not prevent invalid value construction and use (e.g., view an integer as a pointer in C, C++, assembly)

- **strong**: all type errors are detected (at compile time or at run time) (Java, OCaml, Python, Javascript)

**static strong typing**: well-typed programs cannot go wrong [Milner78]

Type checking vs. type inference:

- **checking**: checks the consistency of variable use according to user declarations (C, Java)

- **inference**: discover (almost) automatically a (most general) type consistent with the use (OCaml, except modules... )
Drawbacks of dynamic type checking:

- **errors are detected late, during execution**
  - tendency to reduce the amount of type errors by adding implicit conversions
  - $\rightarrow$ complex and unpredictable semantics

- **costly in time**
  - many type checks
  - an optimizing compiler or JIT generates type-specialized versions of the code

- **costly in memory**
  - we must tag each value with type information at run-time
**Goal:** strong static typing for imperative programs

Classic workflow to introduce types:

- **design a type system**
  set of logical rules stating whether a program is “well typed”

- **prove the soundness with respect to the (operational) semantics**
  well-typed programs cannot go wrong

- **design algorithms to check** typing from user-given type annotations
  or to **infer** type annotations that make the program well typed

Less classic view:

- **design typing by abstraction of the semantics**
  sound by construction
  (static analysis)
Type systems
Simple imperative language

Expressions:  
\[ expr ::= X \quad (\text{variable}) \]
\[ \mid c \quad (\text{constant}) \]
\[ \mid \diamond expr \quad (\text{unary operation}) \]
\[ \mid expr \diamond expr \quad (\text{binary operation}) \]

Statements:  
\[ stat ::= \text{skip} \quad (\text{do nothing}) \]
\[ \mid X \leftarrow expr \quad (\text{assignment}) \]
\[ \mid stat; stat \quad (\text{sequence}) \]
\[ \mid \text{if } expr \text{ then } stat \text{ else } stat \quad (\text{conditional}) \]
\[ \mid \text{while } expr \text{ do } stat \quad (\text{loop}) \]
\[ \mid \text{local } X \text{ in } stat \quad (\text{local variable}) \]

- constants:  
  \[ c \in I \overset{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \]
  (integers and booleans)

- operators:  
  \[ \diamond \in \{ +, -, \times, /, <, \leq, \neg, \land, \lor, =, \neq \} \]

- variables:  
  \[ X \in \mathbb{V} \]
  (\(\mathbb{V}\): set of all program variables)
  variables are now local, with limited scope
  and must be declared
  (no type information... yet!)
Deductive systems

**Deductive system:**
set of axioms and logical rules to derive theorems
defines what is provable in a formal way

**Judgments:**  \( \Gamma \vdash \text{Prop} \)
a fact, meaning: “under hypotheses \( \Gamma \), we can prove \( \text{Prop} \)”

**Rules:**
rule: \( J_1 \cdots J_n \) (hypotheses) \( J \) (conclusion)
axiom: \( J \) (fact)

**Proof tree:** complete application of rules from axioms to conclusion

example in propositional calculus:

\[
\begin{align*}
\Gamma \vdash B & \quad \cdots \\
\Gamma, A \vdash B & \quad \Gamma, A \vdash C \\
\Gamma, A \vdash B \land C \\
\Gamma \vdash A \rightarrow (B \land C)
\end{align*}
\]
Typing judgments

Hypotheses $\Gamma$:
set of type assignments $X : t$, with $X \in \mathcal{V}$, $t \in \text{type}$
(meaning: variable $V$ has type $t$)

Judgments:

- $\Gamma \vdash \text{stat}$
given the type assignments $\Gamma$
$\text{stat}$ is well-typed

- $\Gamma \vdash \text{expr} : \text{type}$
given the type of variables $\Gamma$
$\text{expr}$ is well-typed and has type $\text{type}$
Expression typing

\(\Gamma \vdash c : \text{int} \quad (c \in \mathbb{Z})\)

\(\Gamma \vdash c : \text{bool} \quad (c \in \mathbb{B})\)

\(\Gamma \vdash X : t \quad ((X : t) \in \Gamma)\)

\(\Gamma \vdash e : \text{int}\)

\(\Gamma \vdash -e : \text{int}\)

\(\Gamma \vdash e : \text{bool}\)

\(\Gamma \vdash \neg e : \text{bool}\)

\(\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}\)

\(\Gamma \vdash e_1 \diamond e_2 : \text{int}\) \((\diamond \in \{+, -, \times, /\})\)

\(\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}\)

\(\Gamma \vdash e_1 \diamond e_2 : \text{bool}\) \((\diamond \in \{=, \neq, <, \leq\})\)

\(\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}\)

\(\Gamma \vdash e_1 \diamond e_2 : \text{bool}\) \((\diamond \in \{=, \neq, \land, \lor\})\)

Note: the syntax of an expression uniquely identifies a rule to apply; except for = and \(\neq\), where it depends upon to the choice of a type for \(e_1\) and \(e_2\).
Statement typing

\[
\begin{align*}
\Gamma \vdash \text{skip} & \quad \Gamma \vdash e : t & \quad \Gamma \vdash \text{X} \leftarrow e \quad ((X:t) \in \Gamma) \\
\Gamma \vdash s_1 \quad \Gamma \vdash s_2 & \quad \Gamma \vdash s_1; s_2 \\
\Gamma \vdash s_1 \quad \Gamma \vdash s_2 & \quad \Gamma \vdash e : \text{bool} \\
\Gamma \vdash \text{if} \; e \; \text{then} \; s_1 \; \text{else} \; s_2 & \\
\Gamma \vdash s & \quad \Gamma \vdash e : \text{bool} \\
\Gamma \vdash \text{while} \; e \; \text{do} \; s & \\
\Gamma \vdash \text{local} \; X \; \text{in} \; s & \quad \Gamma \cup \{(X : t)\} \vdash s
\end{align*}
\]

**Definition:** \( s \) is well-typed if we can prove \( \emptyset \vdash s \)

**Note:** the syntax of a statement uniquely identifies a rule to apply, up to the choice of \( t \) in the rule for \( \text{local} \; X \; \text{in} \; s \)
Soundness of typing
Types and errors

**Goal:** well-typed programs “cannot go wrong”

The operational semantics has several kinds of errors:

1. **type mismatch in operators** \((1 \lor 2, \text{true} + 2)\)
2. **value errors** (divide or modulo by 0, use uninitialized variables)

Typing seeks only to prevent statically the first kind of errors.

Value errors can be prevented with static analysis. This is much more complex and costly; we will discuss it later in the course. Typing aims at a “sweet spot”: detect at compile-time all errors of a certain kind.

**Soundness:** well-typed programs have no type mismatch error.

It is proved based on an *operational semantics* of the program.
Soundness of typing

Expression (denotational) semantics with explicit errors

\[
E[\text{expr}] : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{I} \cup \{\Omega_t, \Omega_v\}) \quad \mathcal{E} \overset{\text{def}}{=} \forall \rightarrow (\mathbb{I} \cup \{\omega\}), \quad \mathbb{I} \overset{\text{def}}{=} \mathcal{B} \cup \mathbb{Z}
\]

\[
E[c] \rho \overset{\text{def}}{=} \{c\}
\]

\[
E[[c_1, c_2]] \rho \overset{=} {=} \{c \in \mathbb{Z} | c_1 \leq c \leq c_2\}
\]

\[
E[X] \rho \overset{\text{def}}{=} \{\rho(X) | \text{if } \rho(X) \in \mathbb{I}\} \cup \{\Omega_v | \text{if } \rho(X) = \omega\}
\]

\[
E[-e] \rho \overset{\text{def}}{=} \{-v | v \in (E[e] \rho) \cap \mathbb{Z}\} \cup \{\Omega | \Omega \in (E[e] \rho) \cap \{\Omega_t, \Omega_v\}\} \cup \{\Omega_t | \text{if } (E[e] \rho) \cap \mathcal{B} \neq \emptyset\}
\]

\[
E[e_1/e_2] \rho \overset{\text{def}}{=} \{v_1/v_2 | v_1 \in (E[e_1] \rho) \cap \mathbb{Z}, v_2 \in (E[e_2] \rho) \cap \mathbb{Z}\} \cup \{\Omega | \Omega \in ((E[e_1] \rho) \cup (E[e_2] \rho)) \cap \{\Omega_t, \Omega_v\}\} \cup \{\Omega_t | \text{if } ((E[e_1] \rho) \cup (E[e_2] \rho)) \cap \mathcal{B} \neq \emptyset\} \cup \{\Omega_v | \text{if } 0 \in E[e_2] \rho\}
\]

... 

- \(\omega\) denotes the special “non-initialized” value
- special values \(\Omega_t\) and \(\Omega_v\) denote type errors and value errors
- we show here how to mix non-determinism and errors:
  - errors \(\Omega \in \{\Omega_t, \Omega_f\}\) from sub-expressions are propagated
  - new type errors \(\Omega_t\) and value errors \(\Omega_v\) may be generated
  - we return a set of values and errors
Soundness of typing

Statement (operational) semantics with explicit errors

\( \tau[\ell_1 \text{stat} \ell_2] \subseteq \Sigma^2 \) where \( \Sigma \overset{\text{def}}{=} (\mathcal{L} \times \mathcal{E}) \cup \{\Omega_t, \Omega_v, \omega\} \)

\( \tau[\ell_1 \text{skip} \ell_2] \overset{\text{def}}{=} \{(\ell_1, \rho) \rightarrow (\ell_2, \rho) \mid \rho \in \mathcal{E}\} \)

\( \tau[\ell_1 X \leftarrow e \ell_2] \overset{\text{def}}{=} \)

\[ \{(\ell_1, \rho) \rightarrow (\ell_2, \rho[X \mapsto v]) \mid v \in (\mathbb{E}[e] \rho) \cap \bot\} \cup \{(\ell_1, \rho) \rightarrow \Omega \mid \Omega \in (\mathbb{E}[e] \rho) \cap \{\Omega_t, \Omega_v\}\} \]

\( \tau[\ell_1 \text{while} \ ell_2 e \ do \ ell_3 \ s_4] \overset{\text{def}}{=} \)

\[ \{(\ell_1, \rho) \rightarrow (\ell_2, \rho) \mid \rho \in \mathcal{E}\} \cup \{(\ell_2, \rho) \rightarrow (\ell_3, \rho) \mid \text{true} \in \mathbb{E}[e] \rho\} \cup \{(\ell_2, \rho) \rightarrow (\ell_4, \rho) \mid \text{false} \in \mathbb{E}[e] \rho\} \cup \{(\ell_2, \rho) \rightarrow \Omega_t \mid (\mathbb{E}[e] \rho) \cap \mathbb{Z} \neq \emptyset\} \cup \{(\ell_2, \rho) \rightarrow \Omega \mid \Omega \in (\mathbb{E}[e] \rho) \cap \{\Omega_t, \Omega_v\}\} \cup \tau[\ell_3 \ s_2 \ell_4] \]

(and similarly for if e then s_1 else s_2)

\( \tau[\ell_1 s_1; \ell_2 s_2 \ell_3] \overset{\text{def}}{=} \tau[\ell_1 s_1] \cup \tau[\ell_2 s_2 \ell_3] \)

\( \tau[\ell_1 \text{local } X \text{ in } s \ell_3] \overset{\text{def}}{=} \)

\[ \{(\ell_1, \rho) \rightarrow (\ell_3, \rho'[X \mapsto \rho(X)]) \mid (\ell_2, \rho[X \mapsto \omega]) \rightarrow (\ell_3, \rho') \in \tau[\ell_2 s \ell_3]\} \cup \{(\ell_1, \rho) \rightarrow \Omega \mid (\ell_2, \rho[X \mapsto \omega]) \rightarrow \Omega \in \tau[\ell_2 s \ell_3], \Omega \in \{\Omega_t, \Omega_v\}\} \]

- when entering its scope, a local variable is assigned the “non-initialized” value \( \omega \)
- at the end of its scope, its former value is restored
- special \( \Omega_t, \Omega_v \) states denote errors (blocking states)
- errors \( \Omega \) from expressions are propagated; new type errors \( \Omega_t \) are generated
Type soundness

**Operational semantics:** maximal execution traces

\[
\mathcal{t}[s] \overset{\text{def}}{=} \left\{ (\sigma_0, \ldots, \sigma_n) \mid n \geq 0, \sigma_0 \in I, \sigma_n \in B, \forall i < n: \sigma_i \rightarrow \sigma_{i+1} \right\} \cup \\
\left\{ (\sigma_0, \ldots) \mid \sigma_0 \in I, \forall i \in \mathbb{N}: \sigma_i \rightarrow \sigma_{i+1} \right\}
\]

**Type soundness**

\[ s \text{ is well-typed} \implies \forall (\sigma_0, \ldots, \sigma_n) \in \mathcal{t}[s]: \sigma_n \neq \Omega_t \]

well-typed programs never stop on a type error at run-time
Typing checking
Problem: how do we prove that a program is well typed?

Bottom-up reasoning:
construct a proof tree ending in \( \emptyset \vdash s \) by applying rules “in reverse”

- given a conclusion, there is generally only one rule to apply
- the only rule that requires imagination is:
  \[
  \frac{\Gamma \cup \{(X : t)\} \vdash s}{\Gamma \vdash \text{local } X \text{ in } s}
  \]
  
  \( t \) is a free variable in the hypothesis
  \( \implies \) we need to guess a good \( t \) that makes the proof work

- to type \( \Gamma \vdash e_1 = e_2 : \text{bool} \), we also have to choose between
  \( \Gamma \vdash e_1 : \text{bool} \) and \( \Gamma \vdash e_1 : \text{int} \)
Solution:

ask the programmer to add type information to all variable declarations

we change the syntax of declaration statements into:

\[
\text{stat ::= local } X : type \text{ in stat} \\
| \cdots
\]

The typing rule for local variable declarations becomes deterministic:

\[
\Gamma \cup \{(X : t)\} \vdash s \\
\Gamma \vdash \text{local } X : t \text{ in } s
\]
Given variable types, we assign a single type to each expression. this also solves the indeterminacy in the typing of $e_1 = e_2$.

Algorithm: propagation by induction on the syntax

$$\tau_e : ((\forall \rightarrow \text{type}) \times \text{expr}) \rightarrow (\text{type} \cup \{\Omega_t\})$$

\[
\begin{align*}
\tau_e(\Gamma, c) & \overset{\text{def}}{=} \text{int} \quad \text{if } c \in \mathbb{Z} \\
\tau_e(\Gamma, c) & \overset{\text{def}}{=} \text{bool} \quad \text{if } c \in \mathbb{B} \\
\tau_e(\Gamma, X) & \overset{\text{def}}{=} \Gamma(X) \\
\tau_e(\Gamma, -e) & \overset{\text{def}}{=} \text{int} \quad \text{if } \tau_e(\Gamma, e) = \text{int} \\
\tau_e(\Gamma, -e) & \overset{\text{def}}{=} \text{bool} \quad \text{if } \tau_e(\Gamma, e) = \text{bool} \\
\tau_e(\Gamma, e_1 \odot e_2) & \overset{\text{def}}{=} \text{int} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \text{int}, \odot \in \{+, -, \times, \}/ \\
\tau_e(\Gamma, e_1 \odot e_2) & \overset{\text{def}}{=} \text{bool} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \text{int}, \odot \in \{=, \neq, <, \leq\} \\
\tau_e(\Gamma, e_1 \odot e_2) & \overset{\text{def}}{=} \text{bool} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \text{bool}, \odot \in \{=, \neq, \&, \vee\} \\
\tau_e(\Gamma, e) & \overset{\text{def}}{=} \Omega_t \quad \text{otherwise}
\end{align*}
\]

$\Omega_t$ indicates a type error
Type propagation in statements

Type checking is performed by induction on the syntax of statements:

\[ \tau_s : ((\forall \rightarrow \text{type}) \times \text{stat}) \rightarrow \mathbb{B} \]

\[ \tau_s(\Gamma, \text{skip}) \overset{\text{def}}{=} \text{true} \]

\[ \tau_s(\Gamma, (s_1; s_2)) \overset{\text{def}}{=} \tau_s(\Gamma, s_1) \land \tau_s(\Gamma, s_2) \]

\[ \tau_s(\Gamma, X \leftarrow e) \overset{\text{def}}{=} \tau_e(\Gamma, e) = \Gamma(X) \]

\[ \tau_s(\Gamma, \text{if } e \text{ then } s_1 \text{ else } s_2) \overset{\text{def}}{=} \tau_s(\Gamma, s_1) \land \tau_s(\Gamma, s_2) \land \tau_e(\Gamma, e) = \text{bool} \]

\[ \tau_s(\Gamma, \text{while } e \text{ do } s) \overset{\text{def}}{=} \tau_s(\Gamma, s) \land \tau_e(\Gamma, e) = \text{bool} \]

\[ \tau_s(\Gamma, \text{local } X : t \text{ in } s) \overset{\text{def}}{=} \tau_s(\Gamma[X \mapsto t], s) \]

In particular, \( \tau_s(\Gamma, s) = \text{false} \) if \( \tau_e(\Gamma, e) = \Omega_t \) for some expression \( e \) inside \( s \)

**Theorem**

\[ \tau_s(\emptyset, s) = \text{true} \iff \emptyset \vdash s \text{ is provable} \]

- we have an algorithm to check if a program is well-typed
- the algorithm also assigns statically a type to every sub-expression
  (useful to compile expressions efficiently, without dynamic type information)
Type inference
Type inference

**Problem:** can we avoid specifying types in the program?

**Solution:** automatic type inference

- each variable $X \in \mathbb{V}$ is assigned a type variable $t_X$
- we generate a set of type constraints ensuring that the program is well typed
- we solve the constraint system to infer a type value for each type variable

**Type constraints:** we need equalities on types and type variables

\[
\begin{align*}
\text{type constr} & \ ::= \ type \ expr \ = \ type \ expr \quad \quad \quad \quad \text{(type equality)} \\
\text{type expr} & \ ::= \ int \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{(integers)} \\
& \quad | \quad bool \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{(booleans)} \\
& \quad | \quad t_X \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{(type variable for } X \in \mathbb{V})
\end{align*}
\]
Generating type constraints for expressions

Principle: similar to type propagation

\[ \tau_e : expr \rightarrow (type \ expr \times \mathcal{P}(type\ constr)) \]

\[ \tau_e(c) \overset{def}{=} (int, \emptyset) \quad \text{if } c \in \mathbb{Z} \]

\[ \tau_e(c) \overset{def}{=} (bool, \emptyset) \quad \text{if } c \in \mathbb{B} \]

\[ \tau_e(X) \overset{def}{=} (t_X, \emptyset) \]

\[ \tau_e(\neg e_1) \overset{def}{=} (int, C_1 \cup \{t_1 = int\}) \]

\[ \tau_e(\neg e_1) \overset{def}{=} (bool, C_1 \cup \{t_1 = bool\}) \]

\[ \tau_e(e_1 \diamond e_2) \overset{def}{=} (int, C_1 \cup C_2 \cup \{t_1 = int, t_2 = int\}) \quad \text{if } \diamond \in \{+, -, \times, /\} \]

\[ \tau_e(e_1 \diamond e_2) \overset{def}{=} (bool, C_1 \cup C_2 \cup \{t_1 = int, t_2 = int\}) \quad \text{if } \diamond \in \{<, \leq\} \]

\[ \tau_e(e_1 \diamond e_2) \overset{def}{=} (bool, C_1 \cup C_2 \cup \{t_1 = bool, t_2 = bool\}) \quad \text{if } \diamond \in \{\land, \lor\} \]

\[ \tau_e(e_1 \diamond e_2) \overset{def}{=} (bool, C_1 \cup C_2 \cup \{t_1 = t_2\}) \quad \text{if } \diamond \in \{=, \neq\} \]

where \((t_1, C_1) \overset{def}{=} \tau_e(e_1)\) and \((t_2, C_2) \overset{def}{=} \tau_e(e_2)\)

- we return the type of the expression (possibly a type variable) and a set of constraints to satisfy to ensure it is well typed
- no type environment is needed: variable \(X\) has symbolic type \(t_X\)
- typing \(e_1 = e_2\) and \(e_1 \neq e_2\) reduces to ensuring an equal type for \(e_1\) and \(e_2\)
Generating type constraints for statements

\[ \tau_s : \text{stat} \rightarrow \mathcal{P}(\text{type constr}) \]

\[ \tau_s(\text{skip}) \quad \text{def} \quad \emptyset \]

\[ \tau_s(s_1; s_2) \quad \text{def} \quad \tau_s(s_1) \cup \tau_s(s_2) \]

\[ \tau_s(X \leftarrow e) \quad \text{def} \quad C \cup \{ t_X : t \} \]

\[ \tau_s(\text{if } e \text{ then } s_1 \text{ else } s_2) \quad \text{def} \quad \tau_s(s_1) \cup \tau_s(s_2) \cup C \cup \{ t = \text{bool} \} \]

\[ \tau_s(\text{while } e \text{ do } s) \quad \text{def} \quad \tau_s(s) \cup C \cup \{ t = \text{bool} \} \]

\[ \tau_s(\text{local } X \text{ in } s) \quad \text{def} \quad \tau_s(s) \]

where \((t, C) \equiv \tau_e(e)\)

- we return a set of constraints to satisfy to ensure it is well typed
- for simplicity, scoping in \texttt{local } X \in s \text{ is not handled}
  \[ \implies \text{ we assign a single type for all the local variables with the same name} \]
\[ \tau_s(s) \] is a set of equalities between type variables and constants \texttt{int}, \texttt{bool}

**Solving algorithm:** compute equivalence classes by \textit{unification}

consider \( T = \{ \texttt{int}, \texttt{bool} \} \cup \{ t_X \mid X \in V \} \)

- start with disjoint equivalence classes \( \{ \{ t \} \mid t \in T \} \)
- for each equality \( (t_1 = t_2) \in \tau_s(s) \), merge the classes of \( t_1 \) and \( t_2 \)
  (with union-find data-structure: \( O(|\tau_s(s)| \times \alpha(|T|)) \) time cost)
- if \texttt{int} and \texttt{bool} end up in the same equivalence class, the program is not typable
  otherwise, there exists type assignments \( \Gamma \in V \rightarrow \text{type} \)
  such that the program is typable
If the program is typable, we end up with several equivalence classes:

- the class containing `int` gives the set of integer variables
- the class containing `bool` gives the set of boolean variables
- other classes correspond to “polymorphic” variables
  - e.g. `local X in if X = X then ...`
  - each class can be assigned independently either type `bool` or `int`
  - however, we can prove that these variables are in fact never initialized
    \[\Rightarrow\] polymorphism is not useful in this language
Object-oriented languages
Object-oriented programs

In general, objects are records of fields (values) and methods (functions). Object-oriented languages focus more on code reuse.

Subtyping: type-based formalization of code reuse

“$t_1$ is a subtype of $t_2$” (noted $t_1 <: t_2$) $\iff$ objects of type $t_1$ can be used in all contexts where objects of type $t_2$ can

Examples: different languages implement subtyping differently

- **nominal type systems** (C++, Java, C#)
  
  objects belong to classes
  subtyping is achieved through explicit inheritance
  e.g., `class Circle extends Figure \Rightarrow Circle <: Figure`

- **structural type systems** (OCaml)
  
  objects have types, which list their typed fields and methods
  $t_1 <: t_2$ if $t_1$ has more members than $t_2$
  this is a more semantic definition
Including subtypes in type systems

Types and terms:

\[
\begin{align*}
\text{type} & ::= \text{int} \mid \text{bool} \quad \text{(base types)} \\
& \mid \text{type} \rightarrow \text{type} \quad \text{(functions)} \\
& \mid \{a_1 : \text{type}_1, \ldots, a_n : \text{type}_n\} \quad \text{(record)} \\
\text{term} & ::= \{a_1 = \text{term}_1, \ldots, a_n = \text{term}_n\} \quad \text{(record creation)} \\
& \mid \text{term}.a \quad \text{(record member)} \\
& \mid \ldots
\end{align*}
\]

(a, a_i \in A, where A is a set of record labels)

Structural subtyping rules: defining <:

\[
\begin{align*}
\vdash t <: t && \text{(reflexivity)} \\
\vdash t_2 <: t_1 & \quad \vdash t_1 \rightarrow t_2' <: t_2 \rightarrow t_2' && \text{(functions)}
\end{align*}
\]

\[
\begin{align*}
t_1 <: t'_1 & \quad \cdots & \quad t_i <: t'_i \\
\{a_1 : t_1, \ldots, a_i : t_i, \ldots, a_i+j : t_{i+j}\} <: \{a_1 : t_1, \ldots, a_i : t'_i\} && \text{(record)}
\end{align*}
\]

functions are covariant in their result, contravariant in their argument
records can be extended and/or their members sub-typed
Including subtypes in type systems

Term typing rules:

\[
\begin{align*}
\Gamma \vdash X : t & \quad ((X:t) \in \Gamma) \\
\Gamma \vdash c : \text{type}(c) & \quad (\text{constants}) \\
\Gamma \cup \{X : t\} \vdash m : t' & \quad \Gamma \vdash m : t \rightarrow t' & \quad \Gamma \vdash n : t & \quad (\text{functions}) \\
\Gamma \vdash \text{fun} X \rightarrow m : t \rightarrow t' & \\
\Gamma \vdash m : \{a_1 : t_1, \ldots, a_n : t_n\} & \quad \Gamma \vdash m. a_i : t_i & \quad (\text{record member}) \\
\Gamma \vdash m_1 : t_1 & \quad \ldots & \quad \Gamma \vdash m_n : t_n & \quad (\text{record creation}) \\
\Gamma \vdash \{a_1 = m_1, \ldots, a_n = m_n\} : \{a_1 : t_1, \ldots, a_n : t_n\} & \\
\Gamma \vdash m : t & \quad t <: t' & \quad (\text{subtyping}) \\
\Gamma \vdash m : t' & \\
\end{align*}
\]

(see [Cardelli88])
Type inference

**Type checking:**
easy if all variables are annotated with a type (or class)

**Type inference:** more difficult

- we can still use a constraint-based algorithm
- constraints now have the form $t_1 <: t_2$
  - we cannot use a unification algorithm anymore
  - a closure algorithm is required, which is much more costly
    - ($\simeq$ transitive closure of $=$ vs. transitive closure of $\leq$)
- there is not always a principal solution
  - (closed, constraint-free representation of all the types satisfying the constraints)

More efficient but less powerful object type systems exist

- e.g. OCaml uses row variables with explicit coercion and unification
Types as semantic abstraction
Type semantics

We return to our simple imperative language:

\[
\begin{align*}
expr & ::= X | c | [c_1, c_2] | \Diamond expr | expr \Diamond expr \\
stat & ::= \text{skip} | X \leftarrow expr | \text{stat}; \text{stat} | \text{if } expr \text{ then } \text{stat} \text{ else } \text{stat} | \text{while } expr \text{ do } \text{stat} | \text{local } X \text{ in } \text{stat}
\end{align*}
\]

**Principle:** derive typing from the semantics

- view each type as a set of values
- modify the non-deterministic denotational semantics to reason on types instead of sets of values
  \(\Rightarrow\) the semantics expresses the absence of dynamic type error
  \(\Omega_t\) never occurs in any computation
- the semantics on types is computable, always terminates
  \(\Rightarrow\) we have a static analysis
Types $\#$: representative subsets of $\mathbb{I} \cup \{\omega\} \overset{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \{\Omega_t, \Omega_v, \omega\}$:

- we should distinguish integers, booleans, and type errors $\Omega_t$
- but not value errors $\Omega_v$ nor non-initialization $\omega$ from valid values
- a type in $\#$ over-approximates a set of values in $\mathcal{P}(\mathbb{I} \cup \{\omega\})$
  $\implies$ every subset of $\mathbb{I} \cup \{\omega\}$ must have an over-approximation in $\#$
- $\#$ should be closed under $\cap$
  $\implies$ every $I \subseteq \mathbb{I} \cup \{\Omega\}$ has a best approximation: $\alpha(I) \overset{\text{def}}{=} \cap \{t \in \# \mid I \subseteq t\}$

We define a finite lattice $\# \overset{\text{def}}{=} \{\text{int}^\#, \text{bool}^\#, \text{all}^\#, \bot, \top\}$ where

- $\text{int}^\# \overset{\text{def}}{=} \mathbb{Z} \cup \{\Omega_v, \omega\}$
- $\text{bool}^\# \overset{\text{def}}{=} \mathbb{B} \cup \{\Omega_v, \omega\}$
- $\text{all}^\# \overset{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \{\Omega_v, \omega\}$ (no information, no type error)
- $\bot \overset{\text{def}}{=} \{\Omega_v, \omega\}$ (value error, non-initialization)
- $\top \overset{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \{\Omega_t, \Omega_v, \omega\}$ (no information, possible type error)

$\implies (\#, \subseteq, \cup, \cap, \bot, \top)$ forms a complete lattice
Abstract denotational semantics of expressions

\[ E\![\text{expr}] : \mathcal{E}\# \rightarrow \mathbb{I}\# \] where \( \mathcal{E}\# \overset{\text{def}}{=} \forall \rightarrow \mathbb{I}\#

- \( E\![\ c\ ] \rho \overset{\text{def}}{=} \text{int}\# \) if \( c \in \mathbb{Z} \)
- \( E\![\ c\ ] \rho \overset{\text{def}}{=} \text{bool}\# \) if \( c \in \mathbb{B} \)
- \( E\![\ [c_1, c_2]\ ] \rho \overset{\text{def}}{=} \text{int}\# \) if \( c_1 \leq c_2 \)
- \( E\![\ [c_1, c_2]\ ] \rho \overset{\text{def}}{=} \bot \) if \( c_1 > c_2 \)
- \( E\![\ X\ ] \rho \overset{\text{def}}{=} \rho(X) \)
- \( E\![\circ e\ ] \rho \overset{\text{def}}{=} \circ\#(E\![e\ ] \rho) \)
- \( E\![\ e_1 \circ e_2\ ] \rho \overset{\text{def}}{=} (E\![e_1\ ] \rho) \circ\#(E\![e_2\ ] \rho) \)

- an abstract environment \( \rho \in \mathcal{E}\# \) assigns a type to each variable
- we return \( \bot \) when using a non-initialized variable \( (\rho(X) = \bot) \) or the expression has no value \( ([c_1, c_2]\ where \ c_1 > c_2) \)
- we use abstract unary operators \( \circ\# : \mathbb{I}\# \rightarrow \mathbb{I}\# \)
  and abstract binary operators \( \circ\# : (\mathbb{I}\# \times \mathbb{I}\#) \rightarrow \mathbb{I}\# \)
  (defined in the next slide)
Abstract operators

The abstract operators $\diamond^\#$, $\heartsuit^\#$ are defined as:

\[
\begin{align*}
-^\# x & \overset{\text{def}}{=} \begin{cases} 
\bot & \text{if } x = \bot \\
\text{int}^\# & \text{if } x = \text{int}^\# \\
\top & \text{if } x \in \{\text{bool}^\#, \text{all}^\#, \top\}
\end{cases} \\
\neg^\# x & \overset{\text{def}}{=} \begin{cases} 
\bot & \text{if } x = \bot \\
\text{bool}^\# & \text{if } x = \text{bool}^\# \\
\top & \text{if } x \in \{\text{int}^\#, \text{all}^\#, \top\}
\end{cases} \\
x +^\# y & \overset{\text{def}}{=} \begin{cases} 
\bot & \text{if } x = \bot \lor y = \bot \\
\text{int}^\# & \text{if } x = y = \text{int}^\# \\
\top & \text{otherwise}
\end{cases} \\
x \land^\# y & \overset{\text{def}}{=} \begin{cases} 
\bot & \text{if } x = \bot \lor y = \bot \\
\text{bool}^\# & \text{if } x = y = \text{bool}^\# \\
\top & \text{otherwise}
\end{cases} \\
x <^\# y & \overset{\text{def}}{=} \begin{cases} 
\bot & \text{if } x = \bot \lor y = \bot \\
\text{bool}^\# & \text{if } x = y = \text{int}^\# \\
\top & \text{otherwise}
\end{cases} \\
x =^\# y & \overset{\text{def}}{=} \begin{cases} 
\bot & \text{if } x = \bot \lor y = \bot \\
\text{bool}^\# & \text{if } x = y \in \{\text{int}^\#, \text{bool}^\#\} \\
\top & \text{otherwise}
\end{cases}
\end{align*}
\]

and other operators are similar:

\[
\begin{align*}
-^\# & \overset{\text{def}}{=} x^\# \\
/x^\# & \overset{\text{def}}{=} \lor^\#, \land^\# \overset{\text{def}}{=} \land^\#, \leq^\# \overset{\text{def}}{=} <^\#, \text{and } \neq^\# \overset{\text{def}}{=} =^#
\end{align*}
\]

the operators are strict \(\text{return } \bot \text{ if one argument is } \bot\)
the operators propagate type errors \(\text{return } \top \text{ if one argument is } \top\)
the operators create new type errors \(\text{return } \top\)
Abstract denotational semantics of statements

We consider the complete lattice \((\forall \rightarrow \#), \subseteq, \cup, \cap, \bot, \top)\)
(point-wise lifting)

\[
S\#[\text{stat}] : \mathcal{E}\# \rightarrow \mathcal{E}\# \quad \text{where } \mathcal{E}\# \overset{\text{def}}{=} \forall \rightarrow \#
\]

\[
S\#[\text{skip}] \rho \overset{\text{def}}{=} \rho
\]

\[
S\#[s_1; s_2] \overset{\text{def}}{=} S\#[s_2] \circ S\#[s_1]
\]

\[
S\#[X \leftarrow e] \rho \overset{\text{def}}{=} \begin{cases} 
\top & \text{if } \rho = \top \lor E\#[e] \rho = \top \\
\bot & \text{if } E\#[e] \rho = \bot \\
\rho[X \mapsto E\#[e] \rho] & \text{otherwise}
\end{cases}
\]

- the possibility of a type error is denoted by \(\top\) and is propagated
  we never construct \(\rho\) where \(\rho(X) = \top\) and \(\rho(Y) \neq \top\)

- using a non-initialized variable results in \(\bot\)
  we can have \(\rho(X) = \bot\) and \(\rho(Y) \neq \bot\), if \(X\) is not initialized but \(Y\) is,
  however, \(X \leftarrow X + 1\) will output \(\bot\) where \(Y\) maps to \(\bot\)
Abstract denotational semantics of statements

\[ S^\# \llbracket \text{local } X \ \text{in } s \rrbracket \rho \overset{\text{def}}{=} \begin{cases} \top & \text{if } \rho = \top \\ S\llbracket s \rrbracket (\rho[X \mapsto \bot]) & \text{otherwise} \end{cases} \]

\[ S^\# \llbracket \text{if } e \ \text{then } s_1 \ \text{else } s_2 \rrbracket \rho \overset{\text{def}}{=} \begin{cases} \top & \text{if } \rho = \top \lor E^\# \llbracket e \rrbracket \rho \notin \{\text{bool}^\#, \bot\} \\ \bot & \text{if } E^\# \llbracket e \rrbracket \rho = \bot \\ (S^\# \llbracket s_1 \rrbracket \rho) \cup (S^\# \llbracket s_2 \rrbracket \rho) & \text{otherwise} \end{cases} \]

- returns an error \( \top \) if \( e \) is not boolean
- merges the types inferred from \( s_1 \) and \( s_2 \)

if \( (S^\# \llbracket s_1 \rrbracket \rho)(X) = \text{int}^\# \) and \( (S^\# \llbracket s_2 \rrbracket \rho)(X) = \text{bool}^\# \), we get \( X \mapsto \text{all}^\# \)

i.e., depending on the branch taken, \( X \) may be an integer or a boolean

**Notes:**

constructing \( \rho \) such that \( \rho(X) = \text{all}^\# \) is not a type error
but a type error is generated if \( X \) is used when \( \rho(X) = \text{all}^\# \)
Abstract denotational semantics of statements

\[ S^{\#}[\text{while } e \text{ do } s] \rho \overset{\text{def}}{=} S^{\#}[e] \text{ (lfp } F) \]

where \( F(x) \overset{\text{def}}{=} \rho \cup S^{\#}[s] (S^{\#}[e] x) \)

and \( S^{\#}[e] \rho \overset{\text{def}}{=} \)

\[
\begin{cases} 
\top & \text{if } \rho = \top \lor E^{\#}[e] \rho \notin \{\text{bool}^{\#}, \bot\} \\
\bot & \text{if } E^{\#}[e] \rho = \bot \\
\rho & \text{otherwise}
\end{cases}
\]

- similar to tests \( S^{\#}[\text{if } e \text{ then } s] \), but with a fixpoint
- the sequence \( X_0 \overset{\text{def}}{=} \bot, X_{i+1} \overset{\text{def}}{=} X_i \cup F(X_i) \) is:
  - increasing: \( X_i \subseteq X_{i+1} \) (due to \( \cup \))
  - converges in finite time (because \( \lor \to \emptyset^{\#} \) has bounded height)
  - its limit \( X_\delta \) satisfies \( X_\delta = X_\delta \cup F(X_\delta) \)
    and so \( F(X_\delta) \subseteq X_\delta \)
    \( \implies X_\delta \) is a post-fixpoint of \( F \)

\( \implies S^{\#}[s] \) can be computed in finite time
Soundness

Consider a standard (non abstract) denotational semantics:
\[ S[\text{s}] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E}) \text{ where } \mathcal{E} \overset{\text{def}}{=} \{\Omega_t, \Omega_v\} \cup (\forall \rightarrow (\mathbb{Z} \cup \mathbb{B} \cup \{\omega\})) \]

**Soundness theorem**

\[ \Omega_t \in S[\text{s}] (\lambda X.\omega) \implies S^\#[\text{s}] \bot = \top \]

Proof sketch:
Every set of environments \( R \) can be over-approximated by a function \( \alpha_{\mathcal{E}}(R) \in \forall \rightarrow \mathbb{I}^\# \)

\[ \alpha_{\mathcal{E}}(R) \overset{\text{def}}{=} \begin{cases} \top & \text{if } \Omega_t \in R \\ \lambda X.\alpha_{\mathbb{I}}(\{\rho(X) | \rho \in R \setminus \{\Omega_t, \Omega_v\}\}) & \text{otherwise} \end{cases} \]

where we abstract sets of values \( \mathcal{V} \) as \( \alpha_{\mathbb{I}}(\mathcal{V}) \in \mathbb{I}^\# \)

\[ \alpha_{\mathbb{I}}(\mathcal{V}) \overset{\text{def}}{=} \begin{cases} \bot & \text{if } \mathcal{V} \subseteq \{\omega\} \\ \text{int}^\# & \text{else if } \mathcal{V} \subseteq \mathbb{Z} \cup \{\omega\} \\ \text{bool}^\# & \text{else if } \mathcal{V} \subseteq \mathbb{B} \cup \{\omega\} \\ \text{any}^\# & \text{otherwise} \end{cases} \]

We can then prove by induction on \( s \) that \( \forall R: (\alpha \circ S[\text{s}]) (R) \subseteq (S^\#[\text{s}] \circ \alpha)(R) \)
We conclude by noting that \( \alpha(\lambda X.\omega) = \bot \) and \( \Omega_t \in \alpha(x) \iff x = \top \)

\[ \implies S^\#[\text{s}] \text{ can find statically all dynamic type errors!} \]
Incompleteness

The typing analysis is not complete in general: $S^\#[s] \downarrow = \top \Rightarrow \Omega_t \in S[s](\lambda X.\omega)$

Examples: correct programs that are reported as incorrect

- $P \overset{\text{def}}{=} X \leftarrow 10; \text{if } X < 0 \text{ then } X \leftarrow X + \text{true}$
  
  the erroneous assignment $X \leftarrow X + \text{true}$ is never executed: $S[P] R = \emptyset$
  
  but $S^\#[P] \downarrow = \top$ as $S^\#[P]$ cannot prove that the branch is never executed

- $P \overset{\text{def}}{=} X \leftarrow 10; (\text{while } X > 0 \text{ do } X \leftarrow X + 1); X \leftarrow X + \text{true}$
  
  similarly, $X \leftarrow X + \text{true}$ is never executed
  
  but $S^\#[P]$ cannot express (and so cannot infer) non-termination

$\implies S^\#[s]$ can report spurious type errors

checking exactly $\Omega_t \in S[s] R$ is undecidable, by reduction to the halting problem
Comparison with classic type inference

The analysis is flow-sensitive, classic type inference is flow-insensitive:

- type inference assigns a single static type to each variable
- \( S^\#[s] \) can assign different types to \( X \) at different program points

**example:** “\( X \leftarrow 10; \cdots; X \leftarrow true \)” is not well typed
but its execution has no type error and \( S^\#[s] \downarrow \neq \top \)

The analysis takes “dead variables” into account

not-typable variables do not necessarily result in a type error

**example:** “(if \([0, 1] = 0 \) then \( X \leftarrow 10 \); else \( X \leftarrow true \)); •”
is not well typed as \( X \) cannot store values of type either \text{int} \ or \text{bool} \ at •
but its execution has not type error and \( S^\#[s] \downarrow \neq \top \)

\[ \Rightarrow \]

**static type analysis is more precise than type inference**
but it does not always give a unique, program-wide type assignment for each variable

It is also possible to design a flow-insensitive version of the analysis
(e.g., replace \( S^\#[s] X \) with \( X \cup S^\#[s] X \))
**Problem:** imprecision of the type analysis

\[ P \overset{\text{def}}{=} (\text{if } [0, 1] = 0 \text{ then } X \leftarrow 10; \text{ else } X \leftarrow \text{true}) ; \ Y \leftarrow X ; \ Z \leftarrow X = Y \]

- \( S[P] \) has no type error as \( X \) and \( Y \) always hold values of the same type
- \( S\#[P] \| = \top \): incorrect type error
  
  \( S\#[P] \) gives the environment \([X \mapsto \text{all\#}, \ Y \mapsto \text{all\#}]\)
  
  which contains environments such as \([X \mapsto 12, \ Y \mapsto \text{true}]\)
  
  on which \( X = Y \) causes a type error

**Solution:** polymorphism

represent a set of type assignments: \( \mathcal{E}\# \overset{\text{def}}{=} \mathcal{P}(\forall \rightarrow \#) \) (instead of \( \mathcal{E}\# \overset{\text{def}}{=} \forall \rightarrow \# \))

\begin{align*}
\{ [X \mapsto \text{int\#}, \ Y \mapsto \text{int\#}], & \quad [X \mapsto \text{bool\#}, \ Y \mapsto \text{bool\#}] \} \\
\text{on which } X \# Y \text{ gives bool\# and no error}
\end{align*}

- we can represent relations between types
  
  (e.g., \( X \) and \( Y \) have the same type)

- this typing analysis is more precise but still incomplete

- the analysis is more costly (\(|\mathcal{E}\#| \) is larger)
  
  but still decidable and sound
Conclusion
Type systems are added to programming languages to help ensuring statically the correctness of programs.

Traditional type checking is performed by propagation of declarations. Traditional type inference is performed by constraint solving.

We can also view typing as an abstraction of the dynamic semantic; which can be computed statically, trading precision for decidability. (and the abstract semantics has the same structure as our denotational semantics)

Typing always results in conservative approximations but the amount of approximation can be controlled. (flow-sensitivity, relationality, etc.)
Courses and references on typing:


Research articles and surveys:

