Types

Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris
year 2013–2014

Course 3
5 March 2014
Introduction

Purposes of typing:

- *avoid errors* during the execution of programs by *restricting* them
- help compile programs efficiently
- 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:
typing in language design and implementation
type theory as an alternative to set theory
relations between type theory and proof 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 (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...
**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: \( \textit{expr} ::= \textit{X} \quad \text{(variable)} \)
\[ \quad | \quad \textit{c} \quad \text{(constant)} \]
\[ \quad | \quad \Diamond \textit{expr} \quad \text{(unary operation)} \]
\[ \quad | \quad \textit{expr} \Diamond \textit{expr} \quad \text{(binary operation)} \]

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

- constants: \( \textit{c} \in \mathbb{I} \overset{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \) \text{(integers and booleans)}
- operators: \( \Diamond \in \{+,-,\times,/,<,\leq,\neg,\land,\lor,=,\neq\} \)
- variables: \( \textit{X} \in \mathbb{V} \) \text{(\( \mathbb{V} \): set of all program variables)}

variables are now local, with limited scope
and must be declared \( \text{(no type information...yet!)} \)

e.g.: \texttt{local Y in (local X in (X \leftarrow 0; \textbf{while} X < Y \textbf{ do} X \leftarrow X + 1); Y \leftarrow 2)}
Reminders: 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 Prop”

**Rules:**
rule: \( \frac{J_1 \cdot \cdot \cdot J_n \text{ (hypotheses)}}{J \text{ (conclusion)}} \)
axiom: \( J \) (fact)

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

example in propositional calculus:

\[
\begin{align*}
\vdots \\
\Gamma \vdash B \\
\Gamma, A \vdash B \\
\Gamma, A \vdash C \\
\Gamma, A \vdash B \wedge C \\
\Gamma \vdash A \rightarrow (B \wedge C)
\end{align*}
\]
Typing judgments

Types

\[ \text{type} ::= \text{int} \ (\text{integers}) \]
\[ \text{bool} \ (\text{booleans}) \]

Hypotheses \( \Gamma \):

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

\[ \Gamma \vdash c : \text{int} \quad (c \in \mathbb{Z}) \quad \Gamma \vdash c : \text{bool} \quad (c \in \mathbb{B}) \quad \Gamma \vdash X : t \quad ((X:t) \in \Gamma) \]

\[ \Gamma \vdash e : \text{int} \quad \Gamma \vdash e : \text{bool} \]

\[ \Gamma \vdash -e : \text{int} \quad \Gamma \vdash \neg e : \text{bool} \]

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

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

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

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

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \]

\[ \Gamma \vdash e_1 \diamond e_2 : \text{bool} \quad (\diamond \in \{=, \neq, \wedge, \vee\}) \]

Note: the syntax of an expressions uniquely identifies a rule to apply, up to the choice of types for \( e_1 \) and \( e_2 \) in the rules for =, \( \neq \).
Statement typing

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

**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 \texttt{local } X \texttt{ in } s
Soundness of typing
### 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 analyses. 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

Reminder: denotational semantics of expressions

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

\begin{align*}
E[c] \rho & \overset{\text{def}}{=} \{c\} \\
E[c_1, c_2] \rho & \overset{=} {=} \{ c \in \mathbb{Z} \mid c_1 \leq c \leq c_2 \} \\
E[X] \rho & \overset{\text{def}}{=} \{ \rho(X) \mid \text{if } \rho(X) \in \mathbb{I} \} \cup \{ \Omega_v \mid \text{if } \rho(X) = \omega \} \\
E[-e] \rho & \overset{\text{def}}{=} \{-v \mid v \in (E[e] \rho) \cap \mathbb{Z} \} \cup \\
& \quad \{ \Omega \mid \Omega \in (E[e] \rho) \cap \{\Omega_t, \Omega_v\} \} \cup \\
& \quad \{ \Omega_t \mid \text{if } (E[e] \rho) \cap \mathbb{B} \neq \emptyset \} \\
E[e_1/e_2] \rho & \overset{\text{def}}{=} \{ v_1/v_2 \mid v_1 \in (E[e_1] \rho) \cap \mathbb{Z}, v_2 \in (E[e_2] \rho) \cap \mathbb{Z} \} \cup \\
& \quad \{ \Omega \mid \Omega \in ((E[e_1] \rho) \cup (E[e_2] \rho)) \cap \{\Omega_t, \Omega_v\} \} \cup \\
& \quad \{ \Omega_t \mid \text{if } ((E[e_1] \rho) \cup (E[e_2] \rho)) \cap \mathbb{B} \neq \emptyset \} \cup \\
& \quad \{ \Omega_v \mid \text{if } 0 \in E[e_2] \rho \}
\end{align*}

\[
\omega \text{ denotes the special "non-initialized" value}
\]

\[
\text{special values } \Omega_t \text{ and } \Omega_v \text{ denote type and value errors}
\]

\[
\text{we show here how to mix non-determinism and errors:}
\]

\[
\text{errors } \Omega \in \{\Omega_t, \Omega_v\} \text{ from sub-expressions are propagated}
\]

\[
\text{new type errors } \Omega_t \text{ and value errors } \Omega_v \text{ may be generated}
\]

\[
\text{we return a set of values and errors}
\]
Soundness of typing

Reminder: operational semantics of statements

\[ \tau[^{l_1}\text{stat}^{l_2}] \subseteq \Sigma^2 \] where \( \Sigma \overset{\text{def}}{=} (\mathcal{L} \times \mathcal{E}) \cup \{\Omega_t, \Omega_v, \omega\} \)

\[ \tau[^{l_1}\text{skip}^{l_2}] \overset{\text{def}}{=} \{(l_1, \rho) \rightarrow (l_2, \rho) \mid \rho \in \mathcal{E}\} \]

\[ \tau[^{l_1}X \leftarrow e^{l_2}] \overset{\text{def}}{=} \}
\{ (l_1, \rho) \rightarrow (l_2, \rho[X \mapsto v]) \mid v \in (E[e] \rho) \cap \mathbb{I} \} \cup
\{ (l_1, \rho) \rightarrow \Omega \mid \Omega \in (E[e] \rho) \cap \{\Omega_t, \Omega_v\} \}

\[ \tau[^{l_1}\text{while}^{l_2}e \text{ do }^{l_3}s^{l_4}] \overset{\text{def}}{=} \}
\{ (l_1, \rho) \rightarrow (l_2, \rho) \mid \rho \in \mathcal{E} \} \cup
\{ (l_2, \rho) \rightarrow (l_3, \rho) \mid \text{true} \in E[e] \rho \} \cup \{ (l_2, \rho) \rightarrow (l_4, \rho) \mid \text{false} \in E[e] \rho \} \cup
\{ (l_2, \rho) \rightarrow \Omega_t \mid (E[e] \rho) \cap \mathbb{Z} \neq \emptyset \} \cup \{ (l_2, \rho) \rightarrow \Omega \mid \Omega \in (E[e] \rho) \cap \{\Omega_t, \Omega_v\} \} \cup \tau[^{l_3}s^{l_2}]

(and similarly for if e then s_1 else s_2)

\[ \tau[^{l_1}s_1;^{l_2}s_2^{l_3}] \overset{\text{def}}{=} \tau[^{l_1}s_1^{l_2}] \cup \tau[^{l_2}s_2^{l_3}] \]

\[ \tau[^{l_1}\text{local} X \text{ in }^{l_2}s^{l_2}] \overset{\text{def}}{=} \}
\{ (l_1, \rho) \rightarrow (l_2, \rho'[X \mapsto \rho(X)]) \mid (l_1, \rho[X \mapsto \omega]) \rightarrow (l_2, \rho') \in \tau[^{l_1}s^{l_2}] \} \cup
\{ (l_1, \rho) \rightarrow \Omega \mid (l_1, \rho[X \mapsto \omega]) \rightarrow \Omega \in \tau[^{l_1}s^{l_2}], \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 error (blocking states)
- errors \( \Omega \) from expressions are propagated; new type errors \( \Omega_t \) are generated
Type soundness

Operational semantics: maximal execution traces

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

Type soundness

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

(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:

$$\Gamma \cup \{(X : t)\} \vdash s$$

$$\Gamma \vdash \text{local } X\text{ in } s$$

$t$ 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:

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

The typing rule for local variable declarations becomes **deterministic**:

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

**Algorithm:** propagation by induction on the syntax

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

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

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

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

\[
\tau_e(\Gamma, -e) \overset{\text{def}}{=} \text{int} \quad \text{if } \tau_e(\Gamma, e) = \text{int}
\]

\[
\tau_e(\Gamma, -e) \overset{\text{def}}{=} \text{bool} \quad \text{if } \tau_e(\Gamma, e) = \text{bool}
\]

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

\[
\tau_e(\Gamma, e_1 \diamond e_2) \overset{\text{def}}{=} \text{bool} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \text{int}, \diamond \in \{=, \neq, <, \leq\}
\]

\[
\tau_e(\Gamma, e_1 \diamond e_2) \overset{\text{def}}{=} \text{bool} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \text{bool}, \diamond \in \{=, \neq, \land, \lor\}
\]

\[
\tau_e(e) \overset{\text{def}}{=} \Omega_t \quad \text{otherwise}
\]

\(\Omega_t\) indicates a type error
Typing checking

Type propagation in statements

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

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

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

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

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

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

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

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

(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 checks)
Type inference
Type inference

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

**Solution:** automatic type inference

- each variable $X \in V$ 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

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

**Principle:** similar to type propagation

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

\[
\begin{align*}
\tau_e(c) & \overset{\text{def}}{=} (\text{int}, \emptyset) \quad \text{if } c \in \mathbb{Z} \\
\tau_e(c) & \overset{\text{def}}{=} (\text{bool}, \emptyset) \quad \text{if } c \in \mathbb{B} \\
\tau_e(X) & \overset{\text{def}}{=} (t_X, \emptyset) \\
\tau_e(-e_1) & \overset{\text{def}}{=} (\text{int}, C_1 \cup \{t_1 = \text{int}\}) \\
\tau_e(-e_1) & \overset{\text{def}}{=} (\text{bool}, C_1 \cup \{t_1 = \text{bool}\}) \\
\tau_e(e_1 \diamond e_2) & \overset{\text{def}}{=} (\text{int}, C_1 \cup C_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\}) \quad \text{if } \diamond \in \{+, -, \times, /\} \\
\tau_e(e_1 \diamond e_2) & \overset{\text{def}}{=} (\text{bool}, C_1 \cup C_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\}) \quad \text{if } \diamond \in \{<, \le\} \\
\tau_e(e_1 \diamond e_2) & \overset{\text{def}}{=} (\text{bool}, C_1 \cup C_2 \cup \{t_1 = \text{bool}, t_2 = \text{bool}\}) \quad \text{if } \diamond \in \{\land, \lor\} \\
\tau_e(e_1 \diamond e_2) & \overset{\text{def}}{=} (\text{bool}, C_1 \cup C_2 \cup \{t_1 = t_2\}) \quad \text{if } \diamond \in \{=, \ne\}
\end{align*}
\]

where \((t_1, C_1) = \tau_e(e_1)\) and \((t_2, C_2) = \tau_e(e_2)\)

- 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 \(t_X\)
- \(e_1 = e_2\) and \(e_1 \neq e_2\) reduce to type equality
Generating type constraints for statements

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

\[
\begin{align*}
\tau_s(\text{skip}) & \overset{\text{def}}{=} \emptyset \\
\tau_s(s_1; s_2) & \overset{\text{def}}{=} \tau_s(s_1) \cup \tau_s(s_2) \\
\tau_s(X \leftarrow e) & \overset{\text{def}}{=} C \cup \{t_X : t\} \\
\tau_s(\text{if } e \text{ then } s_1 \text{ else } s_2) & \overset{\text{def}}{=} \tau_s(s_1) \cup \tau_s(s_2) \cup C \cup \{t = \text{bool}\} \\
\tau_s(\text{while } e \text{ do } s) & \overset{\text{def}}{=} \tau_s(s) \cup C \cup \{t = \text{bool}\} \\
\tau_s(\text{local } X \text{ in } s) & \overset{\text{def}}{=} \tau_s(s) \\
\end{align*}
\]

where \((t, C) \overset{\text{def}}{=} \tau_e(e)\)

- we return a set of constraints to satisfy to ensure it is well typed
- for simplicity, scoping in \textbf{local } X \in s is not handled
  \[\implies\] we assign a single type for all local variables of the same name
Solving type constraints

\( \tau_s(s) \) is a set of equalities between type variables and constants \texttt{int}, \texttt{bool}

**Solving algorithm:** compute equivalence classes by unification

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

- 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 exist type assignments \( \Gamma \in \mathbb{V} \rightarrow \text{type} \)
  such that the program is typable
Solving type constraints

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 ···`

  such classes can be assigned either type `bool` or `int`

  however, we can prove that these variables are in fact never initialized
  
  $\implies$ polymorphism is not useful in this language
Types as semantic abstraction
Type semantics

We return to our simple imperative language:

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

Principle: derive typing from the semantics

- view types as sets of values
- modify the non-deterministic denotational semantics to reason on types instead of sets of values (abstraction)
  \[\Rightarrow\] 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 $\mathbb{I} \overset{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \{\Omega_t, \Omega_v\}$:

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

We define a **finite lattice** $\mathbb{I}^\# \overset{\text{def}}{=} \{\text{int}^\#, \text{bool}^\#, \text{all}^\#, \bot, \top\}$ 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, type error)

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

\[ E\[\text{expr}\] : E \rightarrow I \]

where \( E \) \( \overset{\text{def}}{=} \) \( \forall \rightarrow I \)

\[
\begin{align*}
E\[c\] \rho & \overset{\text{def}}{=} \text{int} & \text{if } c \in \mathbb{Z} \\
E\[c\] \rho & \overset{\text{def}}{=} \text{bool} & \text{if } c \in \mathbb{B} \\
E\[\{c_1, c_2\}\] \rho & \overset{\text{def}}{=} \text{int} & \text{if } c_1 \leq c_2 \\
E\[\{c_1, c_2\}\] \rho & \overset{\text{def}}{=} \bot & \text{if } c_1 > c_2 \\
E\[X\] \rho & \overset{\text{def}}{=} \rho(X) \\
E\[e\] \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 environments \( \rho \in E \) 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 \( \circ : I \rightarrow I \)
  and abstract binary operators \( \circ : (I \times I) \rightarrow I \)
  (defined in the next slide)
The abstract operators $\circ\#$, $\diamond\#$ are defined as:

$$
\begin{align*}
-\# \ x & \overset{\text{def}}{=} \\
\quad & \begin{cases}
\bot & \text{if } x = \bot \\
\text{int}\# & \text{if } x = \text{int}\# \\
\top & \text{if } x \in \{\text{bool}\#, \text{all}\#, \top\}
\end{cases} & \\
\rightarrow\# \ x & \overset{\text{def}}{=} \\
\quad & \begin{cases}
\bot & \text{if } x = \bot \\
\text{bool}\# & \text{if } x = \text{bool}\# \\
\top & \text{if } x \in \{\text{int}\#, \text{all}\#, \top\}
\end{cases}
\end{align*}
$$

$$
\begin{align*}
x \ +\# \ y & \overset{\text{def}}{=} \\
\quad & \begin{cases}
\bot & \text{if } x = \bot \lor y = \bot \\
\text{int}\# & \text{if } x = y = \text{int}\# \\
\top & \text{otherwise}
\end{cases} & \\
x \ \lor\# \ y & \overset{\text{def}}{=} \\
\quad & \begin{cases}
\bot & \text{if } x = \bot \lor y = \bot \\
\text{bool}\# & \text{if } x = y = \text{bool}\# \\
\top & \text{otherwise}
\end{cases}
\end{align*}
$$

$$
\begin{align*}
x <\# \ y & \overset{\text{def}}{=} \\
\quad & \begin{cases}
\bot & \text{if } x = \bot \lor y = \bot \\
\text{bool}\# & \text{if } x = y = \text{int}\# \\
\top & \text{otherwise}
\end{cases} & \\
x \ =\# \ y & \overset{\text{def}}{=} \\
\quad & \begin{cases}
\bot & \text{if } x = \bot \lor y = \bot \\
\text{bool}\# & \text{if } x = y \in \{\text{int}\#, \text{bool}\#\} \\
\top & \text{otherwise}
\end{cases}
\end{align*}
$$

and other operators are similar:

$$
\begin{align*}
-\# & \overset{\text{def}}{=} \times\# & \overset{\text{def}}{=} /\#,
\\land\# & \overset{\text{def}}{=} \lor\#, \ \le\# & \overset{\text{def}}{=} <\#,
\text{and } \ne\# & \overset{\text{def}}{=} =\#
\end{align*}
$$

the operators are strict
the operators propagate type errors
the operators create new type errors

(\text{return } \bot \text{ if one argument is } \bot)
(\text{return } \top \text{ if one argument is } \top)
(\text{return } \top)
Abstract denotational semantics of statements

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

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

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

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

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

- 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

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

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

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

if \( (S^\#[][] s_1 \rho)(X) = \text{int}^\# \) and \( (S^\#[][] s_2 \rho)(X) = \text{bool}^\# \), 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

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

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

and \( S^\# \left[ e \right] \rho \overset{\text{def}}{=} \begin{cases} \top & \text{if } \rho = \top \lor E^\# \left[ e \right] \rho \notin \{\text{bool}^\#, \bot\} \\ \bot & \text{if } E^\# \left[ e \right] \rho = \bot \\ \rho & \text{otherwise} \end{cases} \)

- similar to tests \( S^\# \left[ \textbf{if } e \textbf{ then } s \right] \), 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: \( X_i \subseteq X_{i+1} \)
  - converges in finite time
  - its limit \( X_\delta \) satisfies \( X_\delta = X_\delta \cup F(X_\delta) \)
  - and so \( F(X_\delta) \subseteq X_\delta \)
  - \( X_\delta \) is a post-fixpoint of \( F \)

\( \Rightarrow S^\# \left[ s \right] \) can be computed in finite time
Consider a standard (non abstract) denotational semantics:
\[ S[s] : \mathcal{P}(E) \to \mathcal{P}(E) \text{ where } E \overset{\text{def}}{=} \{\Omega_t, \Omega_v\} \cup (\forall \to (\mathbb{Z} \cup B \cup \{\omega\})) \]

**Soundness theorem**
\[ \Omega_t \in S[s](\lambda X.\omega) \implies S^\#[s]\bot = \top \]

Proof sketch:
every set of environments \( R \) can be over-approximated by a function \( \alpha_E(R) \in \forall \to \bot^\# \)
\[ \alpha_E(R) \overset{\text{def}}{=} \begin{cases} \top & \text{if } \Omega_t \in R \\ \lambda X.\alpha_i(\{\rho(X) \mid \rho \in R \setminus \{\Omega_t, \Omega_v\}\}) & \text{otherwise} \end{cases} \]

where we abstract sets of values \( V \) as \( \alpha_i(V) \in \bot^\# \)
\[ \alpha_i(V) \overset{\text{def}}{=} \begin{cases} \bot & \text{if } V \subseteq \{\omega\} \\ \text{int}^\# & \text{else if } V \subseteq \mathbb{Z} \cup \{\omega\} \\ \text{bool}^\# & \text{else if } V \subseteq B \cup \{\omega\} \\ \text{any}^\# & \text{otherwise} \end{cases} \]

we can then prove by induction on \( s \) that \( \forall R : (\alpha \circ S[s])(R) \subseteq (S^\#[s] \circ \alpha)(R) \)
we conclude by noting that \( \alpha(\lambda X.\omega) = \bot \) and \( \Omega_t \in \alpha(x) \iff x = \top \)
\[ \implies S^\#[s] \text{ can find statically all dynamic typing errors!} \]
The typing analysis is not complete in general: $\mathcal{S}^\# s \downarrow = \top \not\Rightarrow \Omega_t \in \mathcal{S} s (\lambda X.\omega)$

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: $\mathcal{S} P R = \emptyset$
  
  but $\mathcal{S}^\# P \downarrow = \top$ as $\mathcal{S}^\# P$ 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 $\mathcal{S}^\# P$ cannot express (and so cannot infer) non-termination

$\implies \mathcal{S}^\# s$ can report spurious typing errors

(checking exactly $\Omega_t \in \mathcal{S} s R$ 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: “$X ← 10; ⋮; X ← true$” is not well typed
but its execution has no type error and $S^♯[s] ⊥ \neq ⊤$

The analysis takes “dead variables” into account
not-typable variables do not necessarily result in a typing error
example: “(if $[0, 1] = 0$ then $X ← 10$; else $X ← true$); ⋮”
is not well typed as $X$ cannot store values of type either `int` or `bool` at ⋮
but its execution has not type error and $S^♯[s] ⊥ \neq ⊤$

⇒

**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

\[
P = \begin{cases} \text{if } [0, 1] = 0 \text{ then } X \leftarrow 10; \text{ else } X \leftarrow \text{true} \end{cases}; \ Y \leftarrow X; \ Z \leftarrow X = Y
\]

- \(S[ P]\) has no type error as \(X\) and \(Y\) always hold values of the same type
- \(S^[ P]\) \(\vdash = \top\): 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: \(\mathcal{E}^{\#} = \mathcal{P}(\forall \rightarrow \forall^{\#})\) (instead of \(\mathcal{E}^{\#} = \forall \rightarrow \forall^{\#}\))

e.g. \{ \([X \mapsto \text{int}^{\#}, Y \mapsto \text{int}^{\#}], [X \mapsto \text{bool}^{\#}, Y \mapsto \text{bool}^{\#}]\) \} on which \(X \equiv^{\#} Y\) gives \(\text{bool}^{\#}\) and no error

- 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** (in a way similar to the denotational semantics).

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


Research articles and surveys:

