Types

Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris

year 2014–2015

Course 6

18 March 2015

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

Classification

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)

Classification

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. . .)

Overview

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: $\mathit{expr} ::= X$ (*variable*)
$\quad\mid\; c$ (*constant*)
$\quad\mid\; \diamond\, \mathit{expr}$ (*unary operation*)
$\quad\mid\; \mathit{expr} \diamond \mathit{expr}$ (*binary operation*)

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

- constants: $c \in \mathbb{I} \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathbb{B}$ (integers and booleans)
- operators: $\diamond \in \{+, -, \times, /, <, \leq, \neg, \wedge, \vee, =, \neq\}$
- 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!)

Reminders: deductive systems

Deductive system:

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

Judgments: $\Gamma \vdash \mathrm{Prop}$

a fact, meaning: "under hypotheses Γ, we can prove Prop"

Rules: rule: $\dfrac{J_1 \cdots J_n \quad (\textit{hypotheses})}{J \quad (\textit{conclusion})}$ axiom: $\dfrac{}{J \quad (\textit{fact})}$

Proof tree: complete application of rules from axioms to conclusion

example in propositional calculus:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma \vdash B}}{\Gamma, A \vdash B} \qquad \dfrac{\cdots}{\Gamma, A \vdash C}}{\Gamma, A \vdash B \wedge C}}{\Gamma \vdash A \rightarrow (B \wedge C)}$$

Typing jugements

Types

$$type ::= \mathbf{int} \quad (integers)$$
$$\mid \mathbf{bool} \quad (booleans)$$

Hypotheses Γ:

set of type assignments $X : t$, with $X \in \mathbb{V}$, $t \in type$

(meaning: variable V has type t)

Judgments:

- $\Gamma \vdash stat$

  given the type assignments Γ
  stat is well-typed

- $\Gamma \vdash expr : type$

  given the type of variables Γ
  expr is well-typed and has type *type*

Expression typing

$$\frac{}{\Gamma \vdash c : \textbf{int}} \ (c \in \mathbb{Z}) \qquad \frac{}{\Gamma \vdash c : \textbf{bool}} \ (c \in \mathbb{B}) \qquad \frac{}{\Gamma \vdash X : t} \ ((X : t) \in \Gamma)$$

$$\frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash -e : \textbf{int}} \qquad \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \neg e : \textbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 \diamond e_2 : \textbf{int}} \ (\diamond \in \{+, -, \times, /\})$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 \diamond e_2 : \textbf{bool}} \ (\diamond \in \{=, \neq, <, \leq\})$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1 \diamond e_2 : \textbf{bool}} \ (\diamond \in \{=, \neq, \wedge, \vee\})$$

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

$$\frac{}{\Gamma \vdash \textbf{skip}} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash X \leftarrow e} \ ((X:t) \in \Gamma)$$

$$\frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash s_1 ; s_2} \qquad \frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2 \qquad \Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2}$$

$$\frac{\Gamma \vdash s \qquad \Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \textbf{while}\ e\ \textbf{do}\ s} \qquad \frac{\Gamma \cup \{(X : t)\} \vdash s}{\Gamma \vdash \textbf{local}\ X\ \textbf{in}\ s}$$

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 **local** X **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 ($1 \vee 2$, $\text{true} + 2$)
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

Reminder: denotational semantics of expressions

$\mathsf{E}[\![\, expr \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{I} \cup \{\Omega_t, \Omega_v\}) \qquad \mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \to (\mathbb{I} \cup \{\omega\})$

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

...

- ω denotes the special “non-initialized” value
- special values Ω_t and Ω_v denote type 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 Ω_t and value errors Ω_v may be generated
  - we return a set of values and errors

Reminder: operational semantics of statements

$\underline{\tau[^{\ell 1}stat^{\ell 2}] \subseteq \Sigma^2}$ where $\Sigma \stackrel{\text{def}}{=} (\mathcal{L} \times \mathcal{E}) \cup \{\Omega_t, \Omega_v, \omega\}$

$\tau[^{\ell 1}\mathbf{skip}^{\ell 2}] \stackrel{\text{def}}{=} \{\, (\ell 1, \rho) \to (\ell 2, \rho) \mid \rho \in \mathcal{E} \,\}$

$$\tau[^{\ell 1}X \leftarrow e^{\ell 2}] \stackrel{\text{def}}{=}$$
$$\{\, (\ell 1, \rho) \to (\ell 2, \rho[X \mapsto v]) \mid v \in (\mathsf{E}[\![\, e \,]\!]\, \rho) \cap \mathbb{I} \,\} \cup$$
$$\{\, (\ell 1, \rho) \to \Omega \mid \Omega \in (\mathsf{E}[\![\, e \,]\!]\, \rho) \cap \{\Omega_t, \Omega_v\} \,\}$$

$$\tau[^{\ell 1}\mathbf{while}\ ^{\ell 2}e\ \mathbf{do}\ ^{\ell 3}s^{\ell 4}] \stackrel{\text{def}}{=}$$
$$\{\, (\ell 1, \rho) \to (\ell 2, \rho) \mid \rho \in \mathcal{E} \,\} \cup$$
$$\{\, (\ell 2, \rho) \to (\ell 3, \rho) \mid \text{true} \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\} \cup \{\, (\ell 2, \rho) \to (\ell 4, \rho) \mid \text{false} \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\} \cup$$
$$\{\, (\ell 2, \rho) \to \Omega_t \mid (\mathsf{E}[\![\, e \,]\!]\, \rho) \cap \mathbb{Z} \neq \emptyset \,\} \cup \{\, (\ell 2, \rho) \to \Omega \mid \Omega \in (\mathsf{E}[\![\, e \,]\!]\, \rho) \cap \{\Omega_t, \Omega_v\} \,\} \cup \tau[^{\ell 3}s^{\ell 2}]$$

(and similarly for **if** e **then** s_1 **else** s_2)

$\tau[^{\ell 1}s_1;\, ^{\ell 2}s_2{}^{\ell 3}] \stackrel{\text{def}}{=} \tau[^{\ell 1}s_1{}^{\ell 2}] \cup \tau[^{\ell 2}s_2{}^{\ell 3}]$

$$\tau[^{\ell 1}\mathbf{local}\ X\ \mathbf{in}\ s^{\ell 3}] \stackrel{\text{def}}{=}$$
$$\{\, (\ell 1, \rho) \to (\ell 3, \rho'[X \mapsto \rho(X)]) \mid (\ell 2, \rho[X \mapsto \omega]) \to (\ell 3, \rho') \in \tau[^{\ell 2}s^{\ell 3}] \,\} \cup$$
$$\{\, (\ell 1, \rho) \to \Omega \mid (\ell 2, \rho[X \mapsto \omega]) \to \Omega \in \tau[^{\ell 2}s^{\ell 3}],\ \Omega \in \{\Omega_t, \Omega_v\} \,\}$$

- when entering its scope, a local variable is assigned the "non-initialized" value ω
- at the end of its scope, its former value is restored
- special Ω_t, Ω_v states denote error (blocking states)
- errors Ω from expressions are propagated; new type errors Ω_t are generated

Type soundness

Operational semantics: maximal execution traces

$$t[\![s]\!] \stackrel{\text{def}}{=} \{ (\sigma_0, \ldots, \sigma_n) \mid n \geq 0, \sigma_0 \in I, \sigma_n \in B, \forall i < n: \sigma_i \to \sigma_{i+1} \} \cup \{ (\sigma_0, \ldots) \mid \sigma_0 \in I, \forall i \in \mathbb{N}: \sigma_i \to \sigma_{i+1} \}$$

Type soundness

s 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

Type declarations

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: $\dfrac{\Gamma \cup \{(X : t)\} \vdash s}{\Gamma \vdash \textbf{local } X \textbf{ in } s}$

  t is a free variable in the hypothesis
  $\Longrightarrow$ we need to guess a good t that makes the proof work
- to type $\Gamma \vdash e_1 = e_2 : \textbf{bool}$, we also have to choose between $\Gamma \vdash e_1 : \textbf{bool}$ and $\Gamma \vdash e_1 : \textbf{int}$

Type declarations

Solution:

ask the programmer to add type information to all variable declarations

we change the syntax of declaration statements into:

$$\begin{array}{lcl} \textit{stat} & ::= & \textbf{local}\ X : \textit{type}\ \textbf{in}\ \textit{stat} \\ & | & \cdots \end{array}$$

The typing rule for local variable declarations becomes deterministic:

$$\frac{\Gamma \cup \{(X : t)\} \vdash s}{\Gamma \vdash \textbf{local}\ X : t\ \textbf{in}\ s}$$

Type propagation in expressions

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

$$\tau_e : ((\mathbb{V} \rightharpoonup \mathit{type}) \times \mathit{expr}) \to (\mathit{type} \cup \{\Omega_t\})$$

$$
\begin{array}{llll}
\tau_e(\Gamma, c) & \stackrel{\text{def}}{=} & \textbf{int} & \text{if } c \in \mathbb{Z} \\
\tau_e(\Gamma, c) & \stackrel{\text{def}}{=} & \textbf{bool} & \text{if } c \in \mathbb{B} \\
\tau_e(\Gamma, X) & \stackrel{\text{def}}{=} & \Gamma(X) & \\
\tau_e(\Gamma, -e) & \stackrel{\text{def}}{=} & \textbf{int} & \text{if } \tau_e(\Gamma, e) = \textbf{int} \\
\tau_e(\Gamma, \neg e) & \stackrel{\text{def}}{=} & \textbf{bool} & \text{if } \tau_e(\Gamma, e) = \textbf{bool} \\
\tau_e(\Gamma, e_1 \diamond e_2) & \stackrel{\text{def}}{=} & \textbf{int} & \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \textbf{int},\ \diamond \in \{+, -, \times, /\} \\
\tau_e(\Gamma, e_1 \diamond e_2) & \stackrel{\text{def}}{=} & \textbf{bool} & \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \textbf{int},\ \diamond \in \{=, \neq, <, \leq\} \\
\tau_e(\Gamma, e_1 \diamond e_2) & \stackrel{\text{def}}{=} & \textbf{bool} & \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \textbf{bool},\ \diamond \in \{=, \neq, \wedge, \vee\} \\
\tau_e(e) & \stackrel{\text{def}}{=} & \Omega_t & \text{otherwise}
\end{array}
$$

Ω_t indicates a type error

Type propagation in statements

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

$$\tau_s : ((\mathbb{V} \rightharpoonup \mathit{type}) \times \mathit{stat}) \to \mathbb{B}$$

$$\begin{array}{lcl}
\tau_s(\Gamma, \textbf{skip}) & \stackrel{\text{def}}{=} & \text{true} \\
\tau_s(\Gamma, (s_1; s_2)) & \stackrel{\text{def}}{=} & \tau_s(\Gamma, s_1) \wedge \tau_s(\Gamma, s_2) \\
\tau_s(\Gamma, X \leftarrow e) & \stackrel{\text{def}}{=} & \tau_e(\Gamma, e) = \Gamma(X) \\
\tau_s(\Gamma, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2) & \stackrel{\text{def}}{=} & \tau_s(\Gamma, s_1) \wedge \tau_s(\Gamma, s_2) \wedge \tau_e(\Gamma, e) = \textbf{bool} \\
\tau_s(\Gamma, \textbf{while } e \textbf{ do } s) & \stackrel{\text{def}}{=} & \tau_s(\Gamma, s) \wedge \tau_e(\Gamma, e) = \textbf{bool} \\
\tau_s(\Gamma, \textbf{local } X : t \textbf{ in } s) & \stackrel{\text{def}}{=} & \tau_s(\Gamma[X \mapsto t], s)
\end{array}$$

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

Theorem

$\tau_s(\emptyset, s) = \text{true} \iff \emptyset \vdash s$ 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 checking)

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

$$
\begin{array}{lcll}
\textit{type const} & ::= & \textit{type expr} = \textit{type expr} & (\textit{type equality}) \\
\textit{type expr} & ::= & \textbf{int} & (\textit{integers}) \\
 & | & \textbf{bool} & (\textit{booleans}) \\
 & | & t_X & (\textit{type variable for } X \in \mathbb{V})
\end{array}
$$

Generating type constraints for expressions

Principle: similar to type propagation

$\tau_e : expr \to (type\ expr \times \mathcal{P}(type\ const))$

$$
\begin{array}{llll}
\tau_e(c) & \stackrel{\text{def}}{=} & (\textbf{int}, \emptyset) & \text{if } c \in \mathbb{Z} \\
\tau_e(c) & \stackrel{\text{def}}{=} & (\textbf{bool}, \emptyset) & \text{if } c \in \mathbb{B} \\
\tau_e(X) & \stackrel{\text{def}}{=} & (t_X, \emptyset) & \\
\tau_e(-e_1) & \stackrel{\text{def}}{=} & (\textbf{int},\ C_1 \cup \{t_1 = \textbf{int}\}) & \\
\tau_e(\neg e_1) & \stackrel{\text{def}}{=} & (\textbf{bool},\ C_1 \cup \{t_1 = \textbf{bool}\}) & \\
\tau_e(e_1 \diamond e_2) & \stackrel{\text{def}}{=} & (\textbf{int},\ C_1 \cup C_2 \cup \{t_1 = \textbf{int}, t_2 = \textbf{int}\}) & \text{if } \diamond \in \{+,-,\times,/\} \\
\tau_e(e_1 \diamond e_2) & \stackrel{\text{def}}{=} & (\textbf{bool},\ C_1 \cup C_2 \cup \{t_1 = \textbf{int}, t_2 = \textbf{int}\}) & \text{if } \diamond \in \{<,\leq\} \\
\tau_e(e_1 \diamond e_2) & \stackrel{\text{def}}{=} & (\textbf{bool},\ C_1 \cup C_2 \cup \{t_1 = \textbf{bool}, t_2 = \textbf{bool}\}) & \text{if } \diamond \in \{\wedge,\vee\} \\
\tau_e(e_1 \diamond e_2) & \stackrel{\text{def}}{=} & (\textbf{bool},\ C_1 \cup C_2 \cup \{t_1 = t_2\}) & \text{if } \diamond \in \{=,\neq\}
\end{array}
$$

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

- 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
- $e_1 = e_2$ and $e_1 \neq e_2$ reduce to type equality

Generating type constraints for statements

$$\tau_s : stat \to \mathcal{P}(type\ const)$$

$$\begin{array}{lcl}
\tau_s(\textbf{skip}) & \stackrel{\text{def}}{=} & \emptyset \\
\tau_s(s_1; s_2) & \stackrel{\text{def}}{=} & \tau_s(s_1) \cup \tau_s(s_2) \\
\tau_s(X \leftarrow e) & \stackrel{\text{def}}{=} & C \cup \{ t_X : t \} \\
\tau_s(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2) & \stackrel{\text{def}}{=} & \tau_s(s_1) \cup \tau_s(s_2) \cup C \cup \{ t = \textbf{bool} \} \\
\tau_s(\textbf{while } e \textbf{ do } s) & \stackrel{\text{def}}{=} & \tau_s(s) \cup C \cup \{ t = \textbf{bool} \} \\
\tau_s(\textbf{local } X \textbf{ in } s) & \stackrel{\text{def}}{=} & \tau_s(s)
\end{array}$$

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

- we return a set of constraints to satisfy to ensure it is well typed
- for simplicity, scoping in **local** $X \in s$ is not handled
  $\Longrightarrow$ we assign a single type for all the local variables with the same name

Solving type constraints

$\tau_s(s)$ is a set of equalities between type variables and constants **int**, **bool**

Solving algorithm: compute equivalence classes by unification

consider $T = \{\textbf{int}, \textbf{bool}\} \cup \{\, t_X \mid X \in \mathbb{V}\,\}$

- 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: $\mathcal{O}(|\tau_s(s)| \times \alpha(|T|))$ time cost)
- if **int** and **bool** end up in the same equivalence class
  the program is not typable

  otherwise, there exists type assignments $\Gamma \in \mathbb{V} \to \textit{type}$
  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** $\cdots$

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

  however, we can prove that these variables are in fact never initialized
  $\Longrightarrow$ 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 $t_1 <: t_2$) $\stackrel{\text{def}}{\iff}$ 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* $\implies$ *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:

$$
\begin{array}{rcll}
\textit{type} & ::= & \textbf{int} \mid \textbf{bool} & \textit{(base types)} \\
 & \mid & \textit{type} \to \textit{type} & \textit{(functions)} \\
 & \mid & \{a_1 : \textit{type}_1, \ldots, a_n : \textit{type}_n\} & \textit{(record)}
\end{array}
$$

$$
\begin{array}{rcll}
\textit{term} & ::= & \{a_1 = \textit{term}_1, \ldots, a_n = \textit{term}_n\} & \textit{(record creation)} \\
 & \mid & \textit{term}.a & \textit{(record member)} \\
 & \mid & \cdots &
\end{array}
$$

($a, a_i \in \mathcal{A}$, where $\mathcal{A}$ is a set of record labels)

Structural subtyping rules: defining $<:$

$$\frac{}{\vdash t <: t} \text{ (reflexivity)} \qquad \frac{\vdash t_2 <: t_1 \qquad \vdash t_1' <: t_2'}{\vdash t_1 \to t_1' <: t_2 \to t_2'} \text{ (functions)}$$

$$\frac{t_1 <: t_1' \qquad \cdots \qquad t_i <: t_i'}{\{a_1 : t_1, \ldots, a_i : t_i, \ldots, a_{i+j} : t_{i+j}\} <: \{a_1 : t_1, \ldots, a_i : t_i'\}} \text{ (record)}$$

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:

$$\frac{}{\Gamma \vdash X : t} \; ((X{:}t)\in\Gamma) \qquad \frac{}{\Gamma \vdash c : \mathit{type}(c)} \; (\mathit{constants})$$

$$\frac{\Gamma \cup \{X : t\} \vdash m : t'}{\Gamma \vdash \textbf{fun}\, X \to m : t \to t'} \qquad \frac{\Gamma \vdash m : t \to t' \qquad \Gamma \vdash n : t}{\Gamma \vdash m\; n : t'} \; (\text{functions})$$

$$\frac{\Gamma \vdash m : \{a_1 : t_1, \ldots, a_n : t_n\}}{\Gamma \vdash m.a_i : t_i} \; (\text{record member})$$

$$\frac{\Gamma \vdash m_1 : t_1 \qquad \cdots \qquad \Gamma \vdash m_n : t_n}{\Gamma \vdash \{a_1 = m_1, \ldots, a_n = m_n\} : \{a_1 : t_1, \ldots, a_n : t_n\}} \; (\text{record creation})$$

$$\frac{\Gamma \vdash m : t \qquad t <: t'}{\Gamma \vdash m : t'} \; (\text{subtyping})$$

...

(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:

$$\begin{array}{rcl} \textit{expr} & ::= & X \\ & | & c \\ & | & [c_1, c_2] \\ & | & \diamond\, \textit{expr} \\ & | & \textit{expr} \diamond \textit{expr} \end{array} \qquad \begin{array}{rcl} \textit{stat} & ::= & \textbf{skip} \\ & | & X \leftarrow \textit{expr} \\ & | & \textit{stat}; \textit{stat} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stat} \textbf{ else } \textit{stat} \\ & | & \textbf{while } \textit{expr} \textbf{ do } \textit{stat} \\ & | & \textbf{local } X \textbf{ in } \textit{stat} \end{array}$$

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)

  $\Longrightarrow$ the semantics expresses the absence of dynamic type error (Ω_t never occurs in any computation)
- the semantics on types is computable, always terminates

  $\Longrightarrow$ we have a static analysis

Type lattice

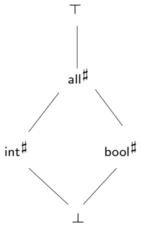
Types $\mathbb{I}^\sharp$: representative subsets of $\mathbb{I} \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \{\Omega_t, \Omega_v\}$:

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

We define a finite lattice $\mathbb{I}^\sharp \stackrel{\text{def}}{=} \{\text{int}^\sharp, \text{bool}^\sharp, \text{all}^\sharp, \bot, \top\}$ where

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

  $\Longrightarrow$ $(\mathbb{I}^\sharp, \subseteq, \cup, \cap, \bot, \top)$ forms a complete lattice



Abstract denotational semantics of expressions

$\mathsf{E}^\sharp[\![\, expr \,]\!] : \mathcal{E}^\sharp \to \mathbb{T}^\sharp$ where $\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{T}^\sharp$

$$
\begin{array}{lcll}
\mathsf{E}^\sharp[\![\, c \,]\!]\,\rho & \stackrel{\text{def}}{=} & \mathsf{int}^\sharp & \text{if } c \in \mathbb{Z} \\
\mathsf{E}^\sharp[\![\, c \,]\!]\,\rho & \stackrel{\text{def}}{=} & \mathsf{bool}^\sharp & \text{if } c \in \mathbb{B} \\
\mathsf{E}^\sharp[\![\, [c_1, c_2] \,]\!]\,\rho & \stackrel{\text{def}}{=} & \mathsf{int}^\sharp & \text{if } c_1 \leq c_2 \\
\mathsf{E}^\sharp[\![\, [c_1, c_2] \,]\!]\,\rho & \stackrel{\text{def}}{=} & \bot & \text{if } c_1 > c_2 \\
\mathsf{E}^\sharp[\![\, X \,]\!]\,\rho & \stackrel{\text{def}}{=} & \rho(X) & \\
\mathsf{E}^\sharp[\![\, \circ\, e \,]\!]\,\rho & \stackrel{\text{def}}{=} & \circ^\sharp\, (\mathsf{E}^\sharp[\![\, e \,]\!]\,\rho) & \\
\mathsf{E}^\sharp[\![\, e_1 \diamond e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & (\mathsf{E}^\sharp[\![\, e_1 \,]\!]\,\rho)\ \diamond^\sharp\ (\mathsf{E}^\sharp[\![\, e_2 \,]\!]\,\rho) &
\end{array}
$$

- an abstract environment $\rho \in \mathcal{E}^\sharp$ 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 $\circ^\sharp : \mathbb{T}^\sharp \to \mathbb{T}^\sharp$
  and abstract binary operators $\diamond^\sharp : (\mathbb{T}^\sharp \times \mathbb{T}^\sharp) \to \mathbb{T}^\sharp$
  (defined in the next slide)

Abstract operators

The abstract operators $\circ^\sharp$, $\diamond^\sharp$ are defined as:

$$-^\sharp x \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } x = \bot \\ \text{int}^\sharp & \text{if } x = \text{int}^\sharp \\ \top & \text{if } x \in \{\text{bool}^\sharp, \text{all}^\sharp, \top\} \end{cases} \qquad \neg^\sharp x \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } x = \bot \\ \text{bool}^\sharp & \text{if } x = \text{bool}^\sharp \\ \top & \text{if } x \in \{\text{int}^\sharp, \text{all}^\sharp, \top\} \end{cases}$$

$$x +^\sharp y \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } x = \bot \vee y = \bot \\ \text{int}^\sharp & \text{if } x = y = \text{int}^\sharp \\ \top & \text{otherwise} \end{cases} \qquad x \vee^\sharp y \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } x = \bot \vee y = \bot \\ \text{bool}^\sharp & \text{if } x = y = \text{bool}^\sharp \\ \top & \text{otherwise} \end{cases}$$

$$x <^\sharp y \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } x = \bot \vee y = \bot \\ \text{bool}^\sharp & \text{if } x = y = \text{int}^\sharp \\ \top & \text{otherwise} \end{cases} \qquad x =^\sharp y \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } x = \bot \vee y = \bot \\ \text{bool}^\sharp & \text{if } x = y \in \{\text{int}^\sharp, \text{bool}^\sharp\} \\ \top & \text{otherwise} \end{cases}$$

and other operators are similar:
$-^\sharp \stackrel{\text{def}}{=} \times^\sharp \stackrel{\text{def}}{=} /^\sharp$, $\wedge^\sharp \stackrel{\text{def}}{=} \vee^\sharp$, $\leq^\sharp \stackrel{\text{def}}{=} <^\sharp$, and $\neq^\sharp \stackrel{\text{def}}{=} =^\sharp$

the operators are strict (return $\bot$ if one argument is $\bot$)
the operators propagate type errors (return $\top$ if one argument is $\top$)
the operators create new type errors (return $\top$)

Abstract denotational semantics of statements

We consider the complete lattice $(\mathbb{V} \to \mathbb{I}^\sharp, \dot\subseteq, \dot\cup, \dot\cap, \dot\bot, \dot\top)$

(point-wise lifting)

$\mathsf{S}^\sharp[\![\, stat \,]\!] : \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ where $\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{I}^\sharp$

$$\mathsf{S}^\sharp[\![\, \mathbf{skip} \,]\!]\,\rho \stackrel{\text{def}}{=} \rho$$

$$\mathsf{S}^\sharp[\![\, s_1; s_2 \,]\!] \stackrel{\text{def}}{=} \mathsf{S}^\sharp[\![\, s_2 \,]\!] \circ \mathsf{S}^\sharp[\![\, s_1 \,]\!]$$

$$\mathsf{S}^\sharp[\![\, X \leftarrow e \,]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} \dot\top & \text{if } \rho = \dot\top \vee \mathsf{E}^\sharp[\![\, e \,]\!]\,\rho = \top \\ \dot\bot & \text{if } \mathsf{E}^\sharp[\![\, e \,]\!]\,\rho = \bot \\ \rho[X \mapsto \mathsf{E}^\sharp[\![\, e \,]\!]\,\rho] & \text{otherwise} \end{cases}$$

- the possibility of a type error is denoted by $\dot\top$ and is propagated
  (we never construct ρ where $\rho(X) = \top$ and $\rho(Y) \neq \top$)
- using a non-initalized variable results in $\dot\bot$
  (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 $\dot\bot$ where Y maps to $\bot$)

Abstract denotational semantics of statements

$$S^\sharp[\![\, \textbf{local}\ X\ \textbf{in}\ s \,]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} \dot{\top} & \text{if } \rho = \dot{\top} \\ S[\![\, s \,]\!]\,(\rho[X \mapsto \bot]) & \text{otherwise} \end{cases}$$

$$S^\sharp[\![\, \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,]\!]\,\rho \stackrel{\text{def}}{=}$$

$$\begin{cases} \dot{\top} & \text{if } \rho = \dot{\top} \vee E^\sharp[\![\, e \,]\!]\,\rho \notin \{\text{bool}^\sharp, \bot\} \\ \dot{\bot} & \text{if } E^\sharp[\![\, e \,]\!]\,\rho = \bot \\ (S^\sharp[\![\, s_1 \,]\!]\,\rho)\,\dot{\cup}\,(S^\sharp[\![\, s_2 \,]\!]\,\rho) & \text{otherwise} \end{cases}$$

- returns an error $\dot{\top}$ if e is not boolean
- merges the types inferred from s_1 and s_2

  if $(S^\sharp[\![\, s_1 \,]\!]\,\rho)(X) = \text{int}^\sharp$ and $(S^\sharp[\![\, s_2 \,]\!]\,\rho)(X) = \text{bool}^\sharp$, we get $X \mapsto \text{all}^\sharp$
  (i.e., depending on the branch taken, X may be an integer or a boolean)

Notes:
constructing ρ such that $\rho(X) = \text{all}^\sharp$ is not a type error
but a type error is generated if X is used when $\rho(X) = \text{all}^\sharp$

Abstract denotational semantics of statements

$\mathsf{S}^\sharp[\![\, \textbf{while } e \textbf{ do } s \,]\!]\,\rho \stackrel{\text{def}}{=} \mathsf{S}^\sharp[\![\, e \,]\!]\,(\mathrm{lfp}\, F)$

where $F(x) \stackrel{\text{def}}{=} \rho \,\dot{\cup}\, \mathsf{S}^\sharp[\![\, s \,]\!]\,(\mathsf{S}^\sharp[\![\, e \,]\!]\, x)$

and $\mathsf{S}^\sharp[\![\, e \,]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} \dot{\top} & \text{if } \rho = \dot{\top} \vee \mathsf{E}^\sharp[\![\, e \,]\!]\,\rho \notin \{\mathrm{bool}^\sharp, \bot\} \\ \dot{\bot} & \text{if } \mathsf{E}^\sharp[\![\, e \,]\!]\,\rho = \bot \\ \rho & \text{otherwise} \end{cases}$

- similar to tests $\mathsf{S}^\sharp[\![\, \textbf{if } e \textbf{ then } s \,]\!]$, but with a fixpoint
- the sequence $X_0 \stackrel{\text{def}}{=} \dot{\bot}$, $X_{i+1} \stackrel{\text{def}}{=} X_i \,\dot{\cup}\, F(X_i)$ is:
  - increasing: $X_i \,\dot{\subseteq}\, X_{i+1}$ (due to $\dot{\cup}$)
  - converges in finite time (because $\mathbb{V} \to \mathbb{I}^\sharp$ has bounded height)
  - its limit X_δ satisfies $X_\delta = X_\delta \,\dot{\cup}\, F(X_\delta)$
    and so $F(X_\delta) \,\dot{\subseteq}\, X_\delta$
    $\Longrightarrow X_\delta$ is a post-fixpoint of F

$\Longrightarrow \mathsf{S}^\sharp[\![\, s \,]\!]$ can be computed in finite time

Soundness

Consider a standard (non abstract) denotational semantics:
$\mathsf{S}[\![\, s \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ where $\mathcal{E} \stackrel{\text{def}}{=} \{\Omega_t, \Omega_v\} \cup (\mathbb{V} \to (\mathbb{Z} \cup \mathcal{B} \cup \{\omega\}))$

Soundness theorem

$$\Omega_t \in \mathsf{S}[\![\, s \,]\!](\lambda X.\omega) \implies \mathsf{S}^\sharp[\![\, s \,]\!]\,\dot{\bot} = \dot{\top}$$

Proof sketch:
every set of environments R can be over-approximated by a function $\alpha_{\mathcal{E}}(R) \in \mathbb{V} \to \mathbb{I}^\sharp$

$$\alpha_{\mathcal{E}}(R) \stackrel{\text{def}}{=} \begin{cases} \dot{\top} & \text{if } \Omega_t \in R \\ \lambda X.\alpha_{\mathbb{I}}(\{\, \rho(X) \mid \rho \in R \setminus \{\Omega_t, \Omega_v\}\,\}) & \text{otherwise} \end{cases}$$

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

$$\alpha_{\mathbb{I}}(V) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } V \subseteq \{\omega\} \\ \text{int}^\sharp & \text{else if } V \subseteq \mathbb{Z} \cup \{\omega\} \\ \text{bool}^\sharp & \text{else if } V \subseteq \mathbb{B} \cup \{\omega\} \\ \text{any}^\sharp & \text{otherwise} \end{cases}$$

we can then prove by induction on s that $\forall R$: $(\alpha \circ \mathsf{S}[\![\, s \,]\!])(R) \mathrel{\dot{\subseteq}} (\mathsf{S}^\sharp[\![\, s \,]\!] \circ \alpha)(R)$
we conclude by noting that $\alpha(\lambda X.\omega) = \dot{\bot}$ and $\Omega_t \in \alpha(x) \iff x = \dot{\top}$

$\implies \mathsf{S}^\sharp[\![\, s \,]\!]$ can find statically all dynamic typing errors!

Incompleteness

The typing analysis is not complete
in general: $\mathsf{S}^\sharp[\![\, s \,]\!]\,\dot\bot = \dot\top \not\Longrightarrow\ \Omega_t \in \mathsf{S}[\![\, s \,]\!]\,(\lambda X.\omega)$

Examples: correct programs that are reported as incorrect

- $P \stackrel{\text{def}}{=} X \leftarrow 10;\ \textbf{if}\ X < 0\ \textbf{then}\ X \leftarrow X + \text{true}$
  the erroneous assignment $X \leftarrow X + \text{true}$ is never executed: $\mathsf{S}[\![\, P \,]\!]\, R = \emptyset$
  but $\mathsf{S}^\sharp[\![\, P \,]\!]\,\dot\bot = \dot\top$ as $\mathsf{S}^\sharp[\![\, P \,]\!]$ cannot prove that the branch is never executed
- $P \stackrel{\text{def}}{=} X \leftarrow 10;\ (\textbf{while}\ X > 0\ \textbf{do}\ X \leftarrow X + 1);\ X \leftarrow X + \text{true}$
  similarly, $X \leftarrow X + \text{true}$ is never executed
  but $\mathsf{S}^\sharp[\![\, P \,]\!]$ cannot express (and so cannot infer) non-termination

$\Longrightarrow \mathsf{S}^\sharp[\![\, s \,]\!]$ can report spurious typing errors

(checking exactly $\Omega_t \in \mathsf{S}[\![\, s \,]\!]\, R$ 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
- $\mathsf{S}^\sharp[\![\, s \,]\!]$ 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 $\mathsf{S}^\sharp[\![\, s \,]\!]\,\dot{\bot} \neq \dot{\top}$

The analysis takes "dead variables" into account

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

example: "(**if** $[0,1] = 0$ **then** $X \leftarrow 10$; **else** $X \leftarrow$ true); $\bullet$"
is not well typed as X cannot store values of type either **int** or **bool** at $\bullet$
but its execution has not type error and $\mathsf{S}^\sharp[\![\, s \,]\!]\,\dot{\bot} \neq \dot{\top}$

$\Longrightarrow$

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 $\mathsf{S}^\sharp[\![\, s \,]\!]\,X$ with $X \,\dot{\cup}\, \mathsf{S}^\sharp[\![\, s \,]\!]\,X$)

Polymorphism and relationality

Problem: imprecision of the type analysis

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

- $\mathsf{S}[\![\, P \,]\!]$ has no type error as X and Y always hold values of the same type
- $\mathsf{S}^\sharp[\![\, P \,]\!]\, \dot{\bot} = \dot{\top}$: incorrect type error

  $\mathsf{S}^\sharp[\![\, P \,]\!]$ gives the environment $[X \mapsto \text{all}^\sharp,\ Y \mapsto \text{all}^\sharp]$
  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: $\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{V} \to \mathbb{I}^\sharp)$ (instead of $\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{I}^\sharp$)

e.g. $\{\, [X \mapsto \text{int}^\sharp,\ Y \mapsto \text{int}^\sharp],\ [X \mapsto \text{bool}^\sharp,\ Y \mapsto \text{bool}^\sharp] \,\}$
on which $X =^\sharp Y$ gives $\text{bool}^\sharp$ 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 ($|\mathcal{E}^\sharp|$ is larger)
  but still decidable and sound

Conclusion

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.)

Bibliography

Courses and references on typing:

[Cardelli97] **L. Cardelli**. *Type systems* In Handbook of Computer Science and Engineering, Chapter 103. CRC Press, 1997.

[Pierce02] **B. Pierce**. *Types and programming languages* In MIT Press, 2002.

Research articles and surveys:

[Cardelli88] **L. Cardelli**. *A semantics of multiple inheritance* In Information and Computation 76, 138-164, 1988.

[Cousot97] **P. Cousot**. *Types as abstract interpretation* In In POPL, 316–331, ACM, 1997.

[Milner78] **R. Milner**. *A theory of type polymorphism in programming* In J. Comput. Syst. Sci., 17:348–375, 1978.