

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

Classification (1/2)

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 (2/2)

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
⇒ 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: $expr ::= X$ (variable)
 | c (constant)
 | $\diamond expr$ (unary operation)
 | $expr \diamond expr$ (binary operation)

Statements: $stat ::= \mathbf{skip}$ (do nothing)
 | $X \leftarrow expr$ (assignment)
 | $stat; stat$ (sequence)
 | **if** $expr$ **then** $stat$ **else** $stat$ (conditional)
 | **while** $expr$ **do** $stat$ (loop)
 | **local** X **in** $stat$ (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!)

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 Γ , we can prove Prop”

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

Proof tree: complete application of rules from axioms to conclusion

example in propositional calculus:

$$\frac{\frac{\frac{\dots}{\Gamma \vdash B}}{\Gamma, A \vdash B} \quad \frac{\dots}{\Gamma, A \vdash C}}{\Gamma, A \vdash B \wedge C}}{\Gamma \vdash A \rightarrow (B \wedge C)}$$

Typing jugements

Types

type ::= **int** (*integers*)
 | **bool** (*booleans*)

Hypotheses Γ :

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

Judgments:

- $\Gamma \vdash \text{stat}$

given the type assignments Γ
 stat is well-typed

- $\Gamma \vdash \text{expr} : \text{type}$

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

Expression typing

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

$$\frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \neg e : \mathbf{int}} \quad \frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \neg e : \mathbf{bool}}$$

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

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

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

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

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

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

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

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 $\mathbf{local } X \mathbf{ in } s$

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 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.

Expression (denotational) semantics with explicit errors

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

$$E[c] \rho \stackrel{\text{def}}{=} \{c\}$$

$$E[[c_1, c_2]] \rho \stackrel{\text{def}}{=} \{c \in \mathbb{Z} \mid c_1 \leq c \leq c_2\}$$

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

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

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

...

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

Statement (operational) semantics with explicit errors

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

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

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

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

(and similarly for if e then s₁ else s₂)

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

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

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

Type soundness

Operational semantics: maximal execution traces

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

Type soundness

s is well-typed $\implies \forall (\sigma_0, \dots, \sigma_n) \in t[s]: \sigma_n \neq \Omega_t$

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

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

Type declarations

Solution:

ask the programmer to **add type information** to all variable declarations

we change the syntax of declaration statements into:

$$\begin{array}{l} \text{stat} ::= \mathbf{local} \ X : \text{type} \ \mathbf{in} \ \text{stat} \\ \quad \quad | \quad \dots \end{array}$$

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

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

Type propagation in expressions

Given variable types, we assign a single type to each expression.

this also solves the indeterminacy in the typing of $e_1 = e_2 \dots$

Algorithm: propagation by induction on the syntax

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

$$\tau_e(\Gamma, c) \stackrel{\text{def}}{=} \mathbf{int} \quad \text{if } c \in \mathbb{Z}$$

$$\tau_e(\Gamma, c) \stackrel{\text{def}}{=} \mathbf{bool} \quad \text{if } c \in \mathbb{B}$$

$$\tau_e(\Gamma, X) \stackrel{\text{def}}{=} \Gamma(X)$$

$$\tau_e(\Gamma, -e) \stackrel{\text{def}}{=} \mathbf{int} \quad \text{if } \tau_e(\Gamma, e) = \mathbf{int}$$

$$\tau_e(\Gamma, \neg e) \stackrel{\text{def}}{=} \mathbf{bool} \quad \text{if } \tau_e(\Gamma, e) = \mathbf{bool}$$

$$\tau_e(\Gamma, e_1 \diamond e_2) \stackrel{\text{def}}{=} \mathbf{int} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \mathbf{int}, \diamond \in \{+, -, \times, /\}$$

$$\tau_e(\Gamma, e_1 \diamond e_2) \stackrel{\text{def}}{=} \mathbf{bool} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \mathbf{int}, \diamond \in \{=, \neq, <, \leq\}$$

$$\tau_e(\Gamma, e_1 \diamond e_2) \stackrel{\text{def}}{=} \mathbf{bool} \quad \text{if } \tau_e(\Gamma, e_1) = \tau_e(\Gamma, e_2) = \mathbf{bool}, \diamond \in \{=, \neq, \wedge, \vee\}$$

$$\tau_e(\Gamma, e) \stackrel{\text{def}}{=} \Omega_t \quad \text{otherwise}$$

Ω_t indicates a **type error**

Type propagation in statements

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

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

$$\tau_s(\Gamma, \mathbf{skip}) \stackrel{\text{def}}{=} \text{true}$$

$$\tau_s(\Gamma, (s_1; s_2)) \stackrel{\text{def}}{=} \tau_s(\Gamma, s_1) \wedge \tau_s(\Gamma, s_2)$$

$$\tau_s(\Gamma, X \leftarrow e) \stackrel{\text{def}}{=} \tau_e(\Gamma, e) = \Gamma(X)$$

$$\tau_s(\Gamma, \mathbf{if } e \mathbf{ then } s_1 \mathbf{ else } s_2) \stackrel{\text{def}}{=} \tau_s(\Gamma, s_1) \wedge \tau_s(\Gamma, s_2) \wedge \tau_e(\Gamma, e) = \mathbf{bool}$$

$$\tau_s(\Gamma, \mathbf{while } e \mathbf{ do } s) \stackrel{\text{def}}{=} \tau_s(\Gamma, s) \wedge \tau_e(\Gamma, e) = \mathbf{bool}$$

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

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

<i>type constr</i>	::=	<i>type expr</i> = <i>type expr</i>	(<i>type equality</i>)
<i>type expr</i>	::=	int	(<i>integers</i>)
		bool	(<i>booleans</i>)
		t_X	(<i>type variable for $X \in \mathbb{V}$</i>)

Generating type constraints for expressions

Principle: similar to type propagation

$\tau_e : \text{expr} \rightarrow (\text{type expr} \times \mathcal{P}(\text{type constr}))$

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

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
- 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

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

$$\tau_s(\text{skip}) \stackrel{\text{def}}{=} \emptyset$$

$$\tau_s(s_1; s_2) \stackrel{\text{def}}{=} \tau_s(s_1) \cup \tau_s(s_2)$$

$$\tau_s(X \leftarrow e) \stackrel{\text{def}}{=} C \cup \{t_X : t\}$$

$$\tau_s(\text{if } e \text{ then } s_1 \text{ else } s_2) \stackrel{\text{def}}{=} \tau_s(s_1) \cup \tau_s(s_2) \cup C \cup \{t = \mathbf{bool}\}$$

$$\tau_s(\text{while } e \text{ do } s) \stackrel{\text{def}}{=} \tau_s(s) \cup C \cup \{t = \mathbf{bool}\}$$

$$\tau_s(\text{local } X \text{ in } s) \stackrel{\text{def}}{=} \tau_s(s)$$

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

- 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 $\Gamma \in \mathbb{V} \rightarrow \text{type}$
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 ...`

each class can be assigned independently either type **bool** or **int**

however, we can prove that these variables are in fact never initialized

⇒ 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:

<i>type</i>	::=	int bool	(base types)
		<i>type</i> → <i>type</i>	(functions)
		{ <i>a</i> ₁ : <i>type</i> ₁ , ..., <i>a</i> _{<i>n</i>} : <i>type</i> _{<i>n</i>} }	(record)
<i>term</i>	::=	{ <i>a</i> ₁ = <i>term</i> ₁ , ..., <i>a</i> _{<i>n</i>} = <i>term</i> _{<i>n</i>} }	(record creation)
		<i>term</i> . <i>a</i>	(record member)
		...	

(*a*, *a*_{*i*} ∈ \mathcal{A} , where \mathcal{A} is a set of record labels)

Structural subtyping rules: defining <:

$\frac{}{\vdash t <: t}$	(reflexivity)	$\frac{\vdash t_2 <: t_1 \quad \vdash t'_1 <: t'_2}{\vdash t_1 \rightarrow t'_1 <: t_2 \rightarrow t'_2}$	(functions)
$t_1 <: t'_1 \quad \dots \quad t_i <: t'_i$			
$\frac{}{\{a_1 : t_1, \dots, a_i : t_i, \dots, a_{i+j} : t_{i+j}\} <: \{a_1 : t_1, \dots, a_i : t'_i\}}$			(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:

$$\overline{\Gamma \vdash X : t} \quad ((X:t) \in \Gamma) \quad \overline{\Gamma \vdash c : \text{type}(c)} \quad (\text{constants})$$

$$\frac{\Gamma \cup \{X : t\} \vdash m : t'}{\Gamma \vdash \mathbf{fun} X \rightarrow m : t \rightarrow t'} \quad \frac{\Gamma \vdash m : t \rightarrow t' \quad \Gamma \vdash n : t}{\Gamma \vdash m n : t'} \quad (\text{functions})$$

$$\frac{\Gamma \vdash m : \{a_1 : t_1, \dots, a_n : t_n\}}{\Gamma \vdash m.a_j : t_j} \quad (\text{record member})$$

$$\frac{\Gamma \vdash m_1 : t_1 \quad \dots \quad \Gamma \vdash m_n : t_n}{\Gamma \vdash \{a_1 = m_1, \dots, a_n = m_n\} : \{a_1 : t_1, \dots, a_n : t_n\}} \quad (\text{record creation})$$

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

...

(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:

<i>expr</i>	::=	<i>X</i>	<i>stat</i>	::=	skip
		<i>c</i>			<i>X</i> ← <i>expr</i>
		[<i>c</i> ₁ , <i>c</i> ₂]			<i>stat</i> ; <i>stat</i>
		◇ <i>expr</i>			if <i>expr</i> then <i>stat</i> else <i>stat</i>
		<i>expr</i> ◇ <i>expr</i>			while <i>expr</i> do <i>stat</i>
					local <i>X</i> in <i>stat</i>

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 (abstraction)
 - ⇒ the semantics expresses the absence of dynamic type error
 - Ω_t never occurs in any computation
- the semantics on types is computable, always terminates
 - ⇒ we have a static analysis

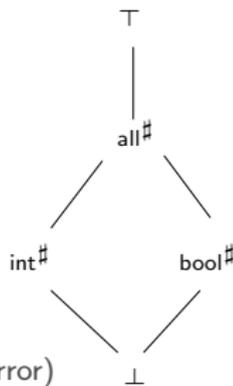
Type lattice

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

- we should distinguish integers, booleans, and type errors Ω_t
- but not value errors Ω_v nor non-initialization ω from valid values
- a type in $\mathbb{I}^\#$ 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 $\mathbb{I}^\#$
- $\mathbb{I}^\#$ should be **closed under \cap**
 \implies every $I \subseteq \mathbb{I} \cup \{\omega\}$ has a best approximation: $\alpha(I) \stackrel{\text{def}}{=} \cap \{t \in \mathbb{I}^\# \mid I \subseteq t\}$

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

- $\text{int}^\# \stackrel{\text{def}}{=} \mathbb{Z} \cup \{\Omega_v, \omega\}$
 - $\text{bool}^\# \stackrel{\text{def}}{=} \mathbb{B} \cup \{\Omega_v, \omega\}$
 - $\text{all}^\# \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \{\Omega_v, \omega\}$ (no information, no type error)
 - $\perp \stackrel{\text{def}}{=} \{\Omega_v, \omega\}$ (value error, non-initialization)
 - $\top \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \{\Omega_t, \Omega_v, \omega\}$ (no information, possible type error)
- $\implies (\mathbb{I}^\#, \subseteq, \cup, \cap, \perp, \top)$ forms a complete lattice



Abstract denotational semantics of expressions

$E^\# \llbracket \text{expr} \rrbracket : \mathcal{E}^\# \rightarrow \mathbb{I}^\#$ where $\mathcal{E}^\# \stackrel{\text{def}}{=} \mathbb{V} \rightarrow \mathbb{I}^\#$

$E^\# \llbracket c \rrbracket \rho$	$\stackrel{\text{def}}{=}$	$\text{int}^\#$	if $c \in \mathbb{Z}$
$E^\# \llbracket c \rrbracket \rho$	$\stackrel{\text{def}}{=}$	$\text{bool}^\#$	if $c \in \mathbb{B}$
$E^\# \llbracket [c_1, c_2] \rrbracket \rho$	$\stackrel{\text{def}}{=}$	$\text{int}^\#$	if $c_1 \leq c_2$
$E^\# \llbracket [c_1, c_2] \rrbracket \rho$	$\stackrel{\text{def}}{=}$	\perp	if $c_1 > c_2$
$E^\# \llbracket X \rrbracket \rho$	$\stackrel{\text{def}}{=}$	$\rho(X)$	
$E^\# \llbracket \circ e \rrbracket \rho$	$\stackrel{\text{def}}{=}$	$\circ^\# (E^\# \llbracket e \rrbracket \rho)$	
$E^\# \llbracket e_1 \diamond e_2 \rrbracket \rho$	$\stackrel{\text{def}}{=}$	$(E^\# \llbracket e_1 \rrbracket \rho) \diamond^\# (E^\# \llbracket e_2 \rrbracket \rho)$	

- an abstract environment $\rho \in \mathcal{E}^\#$ assigns a type to each variable
- we return \perp when using a non-initialized variable ($\rho(X) = \perp$) 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 $\diamond^\# : (\mathbb{I}^\# \times \mathbb{I}^\#) \rightarrow \mathbb{I}^\#$ (defined in the next slide)

Abstract operators

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

$$\neg^\# x \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } x = \perp \\ \text{int}^\# & \text{if } x = \text{int}^\# \\ \top & \text{if } x \in \{\text{bool}^\#, \text{all}^\#, \top\} \end{cases} \quad \neg^\# x \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } x = \perp \\ \text{bool}^\# & \text{if } x = \text{bool}^\# \\ \top & \text{if } x \in \{\text{int}^\#, \text{all}^\#, \top\} \end{cases}$$

$$x +^\# y \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } x = \perp \vee y = \perp \\ \text{int}^\# & \text{if } x = y = \text{int}^\# \\ \top & \text{otherwise} \end{cases} \quad x \vee^\# y \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } x = \perp \vee y = \perp \\ \text{bool}^\# & \text{if } x = y = \text{bool}^\# \\ \top & \text{otherwise} \end{cases}$$

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

and other operators are similar:

$$\neg^\# \stackrel{\text{def}}{=} x^\# \stackrel{\text{def}}{=} /^\#, \wedge^\# \stackrel{\text{def}}{=} \vee^\#, \leq^\# \stackrel{\text{def}}{=}} <^\#, \text{ and } \neq^\# \stackrel{\text{def}}{=} =^\#$$

the operators are strict

the operators propagate type errors

the operators create new type errors

(return \perp if one argument is \perp)

(return \top if one argument is \top)

(return \top)

Abstract denotational semantics of statements

We consider the complete lattice $(\mathbb{V} \rightarrow \mathbb{I}^\#, \underline{\subseteq}, \dot{\cup}, \dot{\cap}, \dot{\perp}, \dot{\top})$
(point-wise lifting)

$S^\# \llbracket \text{stat} \rrbracket : \mathcal{E}^\# \rightarrow \mathcal{E}^\#$ where $\mathcal{E}^\# \stackrel{\text{def}}{=} \mathbb{V} \rightarrow \mathbb{I}^\#$

$$S^\# \llbracket \text{skip} \rrbracket \rho \stackrel{\text{def}}{=} \rho$$

$$S^\# \llbracket s_1; s_2 \rrbracket \stackrel{\text{def}}{=} S^\# \llbracket s_2 \rrbracket \circ S^\# \llbracket s_1 \rrbracket$$

$$S^\# \llbracket X \leftarrow e \rrbracket \rho \stackrel{\text{def}}{=} \begin{cases} \dot{\top} & \text{if } \rho = \dot{\top} \vee E^\# \llbracket e \rrbracket \rho = \top \\ \dot{\perp} & \text{if } E^\# \llbracket e \rrbracket \rho = \perp \\ \rho[X \mapsto E^\# \llbracket e \rrbracket \rho] & \text{otherwise} \end{cases}$$

- 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-initialized variable results in $\dot{\perp}$
we can have $\rho(X) = \perp$ and $\rho(Y) \neq \perp$, if X is not initialized but Y is,
however, $X \leftarrow X + 1$ will output \perp where Y maps to \perp

Abstract denotational semantics of statements

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

$$S^\# \llbracket \text{if } e \text{ then } s_1 \text{ else } s_2 \rrbracket \rho \stackrel{\text{def}}{=} \begin{cases} \dot{\top} & \text{if } \rho = \dot{\top} \vee E^\# \llbracket e \rrbracket \rho \notin \{\text{bool}^\#, \perp\} \\ \dot{\perp} & \text{if } E^\# \llbracket e \rrbracket \rho = \perp \\ (S^\# \llbracket s_1 \rrbracket \rho) \dot{\cup} (S^\# \llbracket s_2 \rrbracket \rho) & \text{otherwise} \end{cases}$$

- returns an error $\dot{\top}$ if e is not boolean
- merges the types inferred from s_1 and s_2
 - if $(S^\# \llbracket s_1 \rrbracket \rho)(X) = \text{int}^\#$ 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 ρ 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

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

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

$$\text{and } S^\#[e] \rho \stackrel{\text{def}}{=} \begin{cases} \dagger & \text{if } \rho = \dagger \vee E^\#[e] \rho \notin \{\text{bool}^\#, \perp\} \\ \perp & \text{if } E^\#[e] \rho = \perp \\ \rho & \text{otherwise} \end{cases}$$

- similar to tests $S^\#[\mathbf{if} \ e \ \mathbf{then} \ s]$, but with a **fixpoint**
- the sequence $X_0 \stackrel{\text{def}}{=} \perp$, $X_{i+1} \stackrel{\text{def}}{=} X_i \dot{\cup} F(X_i)$ is:
 - increasing: $X_i \dot{\subseteq} X_{i+1}$ (due to $\dot{\cup}$)
 - **converges in finite time** (because $\forall \rightarrow \mathbb{N}^\#$ has bounded height)
 - its limit X_δ satisfies $X_\delta = X_\delta \dot{\cup} F(X_\delta)$
and so $F(X_\delta) \dot{\subseteq} X_\delta$
 $\implies X_\delta$ is a **post-fixpoint of F**

$\implies S^\#[s]$ can be computed in finite time

Soundness

Consider a standard (non abstract) denotational semantics:

$S[[s]] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E})$ where $\mathcal{E} \stackrel{\text{def}}{=} \{\Omega_t, \Omega_v\} \cup (\mathbb{V} \rightarrow (\mathbb{Z} \cup \mathbb{B} \cup \{\omega\}))$

Soundness theorem

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

Proof sketch:

every set of environments R can be over-approximated by a function $\alpha_{\mathcal{E}}(R) \in \mathbb{V} \rightarrow \mathbb{I}^\#$

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

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

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

we can then prove by induction on s that $\forall R: (\alpha \circ S[[s]])(R) \dot{\subseteq} (S^\#[[s]] \circ \alpha)(R)$

we conclude by noting that $\alpha(\lambda X.\omega) = \dot{\perp}$ and $\Omega_t \in \alpha(x) \iff x = \dot{\top}$

$\implies S^\#[[s]]$ can find statically all dynamic type errors!

Incompleteness

The typing analysis is not complete

in general: $S^\sharp[[s]] \dot{\perp} = \dot{\top} \not\Rightarrow \Omega_t \in S[[s]](\lambda X.\omega)$

Examples: correct programs that are reported as incorrect

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

$\Rightarrow S^\sharp[[s]]$ 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
- $S^\# \llbracket s \rrbracket$ can assign **different** types to X at different program points

example: “ $X \leftarrow 10; \dots; X \leftarrow \text{true}$ ” is not well typed
but its execution has no type error and $S^\# \llbracket s \rrbracket \perp \neq \top$

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 $X \leftarrow \text{true}$); •”
is not well typed as X cannot store values of type either **int** or **bool** at •
but its execution has not type error and $S^\# \llbracket s \rrbracket \perp \neq \top$

\implies

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 $S^\# \llbracket s \rrbracket X$ with $X \cup S^\# \llbracket s \rrbracket X$)

Polymorphism and relationality

Problem: imprecision of the type analysis

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

- $S[[P]]$ has no type error as X and Y always hold values of the same type
- $S^\#[[P]] \perp = \dagger$: **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: $\mathcal{E}^\# \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{V} \rightarrow \mathbb{I}^\#)$ (instead of $\mathcal{E}^\# \stackrel{\text{def}}{=} \mathbb{V} \rightarrow \mathbb{I}^\#$)

e.g. $\{ [X \mapsto \text{int}^\#, Y \mapsto \text{int}^\#], [X \mapsto \text{bool}^\#, Y \mapsto \text{bool}^\#] \}$
 on which $X =^\# Y$ 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

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.)

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 POPL, 316–331, ACM, 1997.

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