Introduction to Polymorphic Type Systems

From STLB to parametric and ad-hoc polymorphism

Mickaël Laurent

October 2, 2025

Charles University, Prague

https://mlaurent.ovh/courses/types/intro.pdf

Why static typing?

- Type safety: *well-typed programs do not go wrong* (Robin Milner)
  - What does *wrong* mean here?
  - Out-of-bound reads/writes are usually not catched by the type system
- Compilation/execution:
  - Function calls can be resolved statically (no need for a *dynamic dispatch*)
  - Rust: we statically know when memory should be freed (no need for a GC)
- Ecosystem:
  - Type information is useful for the programmer (documentation)
  - Type information is useful for the IDE (refactoring, completion, etc.)

Multiple flavors of types

- More focused on expressivity (not in this lecture):
  - Dependent types (e.g. proof assistants: Lean, Rocq)

    ⇒ Type-checking is not decidable, the programmer must annotate manually
  - Refinement types (e.g. verification-aware languages: F*, Dafny, Why3)

    ⇒ Type-checking is (usually) not decidable,

    but can be partly automated using SMT-solvers
- More focused on usability:
  - Type systems of mainstream static languages (e.g. Rust, OCaml)

    ⇒ Type-checking is decidable, and we may even have type inference
  - Static type-checkers for dynamic languages (e.g. TypeScript, MyPy, PyRight)

    ⇒ Got more attention in the last 10 years (my research area)

Simply Typed Lambda Calculus (STLC)

Reminder: λ-calculus

Constants	c	$::=$	$\texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid 2 \mid 3 \mid \dots$
Expressions	e	$::=$	$c \mid x \mid \lambda x.e \mid e\,e \mid (e, e) \mid \pi_1 e \mid \pi_2 e \mid e\,\texttt{?}\,e\,\texttt{:}\,e$
Values	v	$::=$	$c \mid \lambda x.e \mid (v, v)$

- **Expressions** e are programs, composed of: constants c, variables x, functions ($\lambda x.e$), function application ($e\ e$), pairs ((e, e)), pair projections (π_1, π_2), and conditionals ($e\,\texttt{?}\,e\,\texttt{:}\,e$).
- **Values** v are results (fully-reduced expressions): constants, functions, and pairs of values.
- All values are expressions, but not all expressions are values.

$$\begin{array}{lrcl} \textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid e\,e \mid (e,e) \mid \pi_1 e \mid \pi_2 e \mid e\,\texttt{?}\,e\,\texttt{:}\,e \\ \textbf{Values} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array}$$

Call-by-value semantics:

- When we have an application, we first reduce (*evaluate*) the argument, then we enter the function (β-reduction),
- When we have a projection, we first reduce the argument, then we project.

$(\lambda x.e)v$	$\rightsquigarrow$	$e\{v/x\}$	β-reduction	*(function application)*
$\pi_1(v_1, v_2)$	$\rightsquigarrow$	v_1	Left projection	*(return first element of a pair)*
$\pi_2(v_1, v_2)$	$\rightsquigarrow$	v_2	Right projection	*(return second element of a pair)*
$\texttt{true}\,\texttt{?}\,e_1\,\texttt{:}\,e_2$	$\rightsquigarrow$	e_1	Conditional (1)	*(take first branch)*
$\texttt{false}\,\texttt{?}\,e_1\,\texttt{:}\,e_2$	$\rightsquigarrow$	e_2	Conditional (2)	*(take second branch)*

Example

We assume we have an integer comparison function `leq`:

$$\texttt{leq}\,(n_1, n_2) \rightsquigarrow \begin{cases} \texttt{true} & \text{if } n_1 \leq n_2 \\ \texttt{false} & \text{otherwise} \end{cases}$$

Apply `max` $\equiv \lambda x.\, \lambda y.\, \texttt{leq}\,(x, y)\,\texttt{?}\,y\,\texttt{:}\,x$ to 42 and 24 and write the reduction steps.

$$\begin{aligned}
& (\lambda x.\, \lambda y.\, \texttt{leq}\,(x, y)\,\texttt{?}\,y\,\texttt{:}\,x)\ 42\ 24 && (\beta\text{-reduction}) \\
\rightsquigarrow\ & (\lambda y.\, \texttt{leq}\,(42, y)\,\texttt{?}\,y\,\texttt{:}\,42)\ 24 && (\beta\text{-reduction}) \\
\rightsquigarrow\ & \texttt{leq}\,(42, 24)\,\texttt{?}\,24\,\texttt{:}\,42 && (\text{semantics of } \texttt{leq}) \\
\rightsquigarrow\ & \texttt{false}\,\texttt{?}\,24\,\texttt{:}\,42 && (\text{conditional}) \\
\rightsquigarrow\ & 42
\end{aligned}$$

Exercise

Currified function it takes its different parameters (x_1, x_2, ...) successively:
$\lambda x_1.\, \lambda x_2.\, \dots\, x_1 \dots x_2 \dots$. A currified function can be partially applied.

Uncurrified function it takes its different parameters (x_1, x_2, ...) all at once
using a pair/tuple: $\lambda x.\, \dots\, \pi_1 x \dots \pi_2 x \dots$.

Write an uncurrified version of `max` $\equiv \lambda x.\, \lambda y.\, \texttt{leq}\,(x, y)\,\texttt{?}\,y\,\texttt{:}\,x$,
apply it to 42 and 24 and write the reduction steps.

Simple Monomorphic Types

$$\begin{array}{llll} \textbf{Base Types} & b & ::= & \texttt{bool} \mid \texttt{int} \mid \ldots \\ \textbf{Types} & s, t & ::= & b \mid t \to t \mid t \times t \end{array}$$

We have a type constructor for each kind of value of our language:

$$\textbf{Values} \quad v \quad ::= \quad c \mid \lambda x.e \mid (v, v)$$

- Base types (b) represent constants,
- Arrows ($\to$) represent λ-abstractions,
- Products ($\times$) represent pairs.

Examples:

- Currified `max` has type $\texttt{int} \to \texttt{int} \to \texttt{int}$
  ($\to$ is associative to the right, so $\texttt{int} \to \texttt{int} \to \texttt{int} \;\equiv\; \texttt{int} \to (\texttt{int} \to \texttt{int})$)
- Uncurrified `max` has type $(\texttt{int} \times \texttt{int}) \to \texttt{int}$

Typing rules

$$[\text{BoolF}]\ \frac{}{\texttt{false} : \texttt{bool}} \qquad [\text{BoolT}]\ \frac{}{\texttt{true} : \texttt{bool}} \qquad [\text{Int}]\ \frac{}{n : \texttt{int}}$$

$$[\text{Pair}]\ \frac{e_1 : t_1 \qquad e_2 : t_2}{(e_1, e_2) : t_1 \times t_2} \qquad [\text{Cond}]\ \frac{e : \texttt{bool} \qquad e_1 : t \qquad e_2 : t}{e\ ?\ e_1 : e_2 : t}$$

$$[\text{App}]\ \frac{e_1 : s \to t \qquad e_2 : s}{e_1\ e_2 : t} \qquad [\text{LProj}]\ \frac{e : t_1 \times t_2}{\pi_1 e : t_1} \qquad [\text{RProj}]\ \frac{e : t_1 \times t_2}{\pi_2 e : t_2}$$

- A statement $e : t$ (meaning e *has type* t) is called a judgment,
- Over the line, we have the premises,
- Under the line, we have the conclusion,
- A rule with no premise ([BoolF], [BoolT], [Int]) is sometimes called an axiom.

What to do for λ-abstractions and variables?

$$[\text{Abs}]\ \frac{\textbf{???}}{\lambda x.e : s \to t} \qquad [\text{Var}]\ \frac{\textbf{???}}{x : t}$$

We need a notion of type environment (or *type context*) to store the type of the variables that are in the current scope.

$$\textbf{Type Environments} \quad \Gamma \quad ::= \quad \varnothing \mid x : t, \Gamma$$

We can freely reorder bindings in an environment.

We add environments to our judgments, $\Gamma \vdash e : t$, which reads: under the environment Γ, the expression e has type t.

$$[\text{Abs}]\ \frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t} \qquad [\text{Var}]\ \frac{}{\Gamma, x : t \vdash x : t}$$

$$[\text{BoolF}]\ \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \qquad [\text{BoolT}]\ \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad [\text{Int}]\ \frac{}{\Gamma \vdash n : \texttt{int}}$$

$$[\text{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\text{Cond}]\ \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e\,?\,e_1 \mathbin{\textbf{:}} e_2 : t}$$

$$[\text{App}]\ \frac{\Gamma \vdash e_1 : s \to t \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ e_2 : t} \qquad [\text{LProj}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\text{RProj}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}$$

Exercise

We consider our built-in comparison `leq` to be a variable of type $(\texttt{int} \times \texttt{int}) \to \texttt{bool}$ in the current scope.

Try to derive the type $\texttt{int} \to (\texttt{int} \to \texttt{int})$ for the function

$$\texttt{max} \quad \equiv \quad \lambda x.\, \lambda y.\, \texttt{leq}\,(x, y)\,\texttt{?}\,y\,\texttt{:}\,x$$

under the initial environment $\quad \Gamma \quad = \quad \texttt{leq} : (\texttt{int} \times \texttt{int}) \to \texttt{bool}.$

$$[\text{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\text{Cond}]\ \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e\,\texttt{?}\,e_1\,\texttt{:}\,e_2 : t}$$

$$[\text{App}]\ \frac{\Gamma \vdash e_1 : s \to t \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ e_2 : t} \qquad [\text{Abs}]\ \frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t} \qquad [\text{Var}]\ \frac{}{\Gamma, x : t \vdash x : t}$$

For concision, we define:

$$\begin{array}{rcl}
\Gamma & = & \mathtt{leq} : (\mathtt{int} \times \mathtt{int}) \to \mathtt{bool} \\
\Gamma' & = & \mathtt{leq} : (\mathtt{int} \times \mathtt{int}) \to \mathtt{bool}, \quad x : \mathtt{int} \\
\Gamma'' & = & \mathtt{leq} : (\mathtt{int} \times \mathtt{int}) \to \mathtt{bool}, \quad x : \mathtt{int}, \quad y : \mathtt{int}
\end{array}$$

Typing derivation for `max`:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma'' \vdash \mathtt{leq}\,(x, y) : \mathtt{bool}}\,[\text{App}] \qquad \dfrac{}{\Gamma'' \vdash y : \mathtt{int}} \qquad \dfrac{}{\Gamma'' \vdash x : \mathtt{int}}}{\Gamma'' \vdash \mathtt{leq}\,(x, y)\,?\,y : x : \mathtt{int}}\,[\text{Cond}]}{\Gamma' \vdash \lambda y.\, \mathtt{leq}\,(x, y)\,?\,y : x : \mathtt{int} \to \mathtt{int}}\,[\text{Abs}]}{\Gamma \vdash \lambda x.\, \lambda y.\, \mathtt{leq}\,(x, y)\,?\,y : x : \mathtt{int} \to (\mathtt{int} \to \mathtt{int})}\,[\text{Abs}]$$

Type safety

Theorem (Type safety)

If $\varnothing \vdash e : t$, then either:

- $e \leadsto^\infty$ *(e diverges), or*
- $e \leadsto^* v$ *with* $\varnothing \vdash v : t$ *(e reduces to a value of the same type)*

In particular, this means that a well-typed expression cannot get stuck (e.g. $42\;42$).

How to prove this theorem?

Lemma (Type preservation)

If $\Gamma \vdash e : t$ and $e \leadsto e'$, then $\Gamma \vdash e' : t$.

Lemma (Progress)

If $\varnothing \vdash e : t$, then either e is a value or $\exists e'.\ e \leadsto e'$.

Towards an algorithm

$$[\text{App}]\ \frac{\Gamma \vdash e_1 : s \to t \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ e_2 : t} \qquad [\text{LProj}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\text{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

Each rule of our type system is structural: it only applies to one expression constructor.

- [App] only applies on applications,
- [LProj] only applies on left projections,
- [Pair] only applies on pairs, etc.

This makes our type system syntax-directed: we know which rule to apply just by looking at the syntax of the expression we are typing.

However, the rule [Abs] is not analytic: the domain s of the λ-abstraction does not appear in the initial expression or environment.

$$\text{[Abs]}\ \frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t}$$

While this domain s can be deduced if we know the resulting type (type checking), this is not the case if we want to do type inference (i.e. find the type of the expression).

We will write the input of the algorithm in green, and its output in blue.

Type checking Finding a derivation for the judgment $\Gamma \vdash e : t$
(i.e. construct a derivation tree by recursively building the premises)

Type inference Finding a type t and a derivation for the judgment $\Gamma \vdash e : t$.

Unification

We will turn our type system into an inference algorithm based on unification.

To that purpose, we add type variables to the syntax of types:

Base Types	b	$::=$	$\texttt{bool} \mid \texttt{int} \mid \ldots$
Monomorphic Types	s, t	$::=$	$b \mid t \to t \mid t \times t \mid \alpha$

Definition (Unifier)

For two terms e_1 and e_2, we say that a type substitution ψ is a unifier for e_1 and e_2 if and only if $e_1\psi \equiv e_2\psi$.

Definition (Most general unifier)

For two terms e_1 and e_2, we say that a type substitution ψ is a most general unifier for e_1 and e_2 if and only if: (i) ψ is a unifier for e_1 and e_2, and (2) for any unifier ψ' for e_1 and e_2, there exists ψ'' such that $\psi' = \psi'' \circ \psi$.

Example: for two terms $e_1 \equiv \alpha \to \beta$ and $e_2 \equiv \texttt{int} \to \beta$,

- $\{\alpha \rightsquigarrow \texttt{int}\}$ is a most general unifier for e_1 and e_2,
- $\{\alpha \rightsquigarrow \texttt{int},\ \beta \rightsquigarrow \texttt{int}\}$ is a unifier for e_1 and e_2.

Property (Principality of unification)

If there exists a unifier for e_1 and e_2,
then there exists a most general unifier for e_1 and e_2.

Definition (Unification)

The unification function $\mathrm{mgu}(e_1, e_2)$ returns a most general unifier for e_1 and e_2 if it exists, and is not defined otherwise.

The unification function $\mathrm{mgu}(e_1, e_2)$ can be computed in linear time
(cf. `unify` algorithm from your previous lectures).

Exercise

- What is mgu($\texttt{int} \times \beta,\ \ \alpha \times (\alpha \to \texttt{int})$)?
- When there exists a most general unifier, is it always **unique**?
- What is mgu($\alpha,\ \ \beta$)?
- What is mgu($\alpha \to \texttt{int},\ \ \alpha$)?

Type inference

To infer the domain of a λ-abstraction, we initially type it with a fresh type variable α, and then substitute it on-the-fly when required, using unification.

To do so, we extend our typing judgements (green=input, blue=output):

$$\Gamma \vdash e : t \dashv \psi$$

It reads: under the typing environment Γ, the expression e can be typed t provided that we apply the substitution ψ to our context (i.e. to Γ).

In particular, if we get $\Gamma \vdash e : t \dashv \psi$, then $\Gamma\psi \vdash e : t$ should be derivable.

Axiomatic rules just return the identity substitution, noted $\varnothing$:

$$[\text{BoolF}]\ \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool} \dashv \varnothing} \qquad [\text{BoolT}]\ \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool} \dashv \varnothing} \qquad [\text{Int}]\ \frac{}{\Gamma \vdash n : \texttt{int} \dashv \varnothing}$$

$$[\text{Var}]\ \frac{}{\Gamma, x : t \vdash x : t \dashv \varnothing}$$

Note: if no rule can be applied (e.g. we want type x but there is no binding for x in our environment Γ), then the type inference algorithm fails (we get a static type error).

$$[\text{Abs}]\ \frac{\Gamma, x : \alpha \vdash e : t \dashv \psi}{\Gamma \vdash \lambda x.e : (\alpha\psi) \to t \dashv \psi}\ \alpha \text{ fresh}$$

To type $\lambda x.e$, we assume x has a fresh type α and recursively call our algorithm on the body e, yielding a type t and a substitution ψ. The substitution ψ must be applied to our context (in particular to α), so the resulting type for our λ-abstraction is $(\alpha\psi) \to t$.

$$[\mathrm{App}]\ \frac{\Gamma \vdash e_1 : t_1 \dashv \psi_1 \qquad \Gamma\psi_1 \vdash e_2 : t_2 \dashv \psi_2 \qquad \psi = \mathrm{mgu}(t_1\psi_2, t_2 \to \alpha)}{\Gamma \vdash e_1\ e_2 : \alpha\psi \dashv \psi \circ \psi_2 \circ \psi_1}\ \alpha \text{ fresh}$$

To type an application $e_1\ e_2$, we first recursively type e_1, yiedling a type t_1 and a substitution ψ_1. We then type e_2 under the updated context $\Gamma\psi_1$, yielding a type t_2 and a substitution ψ_2.

At this point, the argument has type t_2, and the function has type $t_1\psi_2$. We thus use unification to solve the constraint

$$t_1\psi_2 \equiv t_2 \to \alpha$$

(with α a fresh type variable representing the type of the result),
yielding a substitution ψ. The type of the result of the application is now $\alpha\psi$.

$$[\text{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \dashv \psi_1 \qquad \Gamma\psi_1 \vdash e_2 : t_2 \dashv \psi_2}{\Gamma \vdash (e_1, e_2) : (t_1\psi_2) \times t_2 \dashv \psi_2 \circ \psi_1}$$

$$[\text{LProj}]\ \frac{\Gamma \vdash e : t \dashv \psi \qquad \psi' = \text{mgu}(t, \alpha_1 \times \alpha_2)}{\Gamma \vdash \pi_1 e : \alpha_1\psi' \dashv \psi' \circ \psi}\ \alpha_1, \alpha_2 \text{ fresh}$$

$$[\text{RProj}]\ \frac{\Gamma \vdash e : t \dashv \psi \qquad \psi' = \text{mgu}(t, \alpha_1 \times \alpha_2)}{\Gamma \vdash \pi_2 e : \alpha_2\psi' \dashv \psi' \circ \psi}\ \alpha_1, \alpha_2 \text{ fresh}$$

$$[\text{Cond}]\ \frac{\begin{array}{c}\Gamma \vdash e : s \dashv \psi \qquad \psi' = \text{mgu}(s, \texttt{bool}) \\ (\Gamma\psi)\psi' \vdash e_1 : t_1 \dashv \psi_1 \qquad ((\Gamma\psi)\psi')\psi_1 \vdash e_2 : t_2 \dashv \psi_2 \qquad \psi'' = \text{mgu}(t_1\psi_2, t_2)\end{array}}{\Gamma \vdash e\ ?\ e_1 : e_2 : t_2\psi'' \dashv \psi'' \circ \psi_2 \circ \psi_1 \circ \psi' \circ \psi}$$

These typing rules are a reformulation of Algorithm W in the context of STLC.

Exercise

Derive a type for $\lambda x.\ (\pi_2 x, \pi_1 x)$ under the empty environment $\varnothing$.

$$[\text{Abs}]\ \frac{\Gamma, x : \alpha \vdash e : t \dashv \psi}{\Gamma \vdash \lambda x.e : (\alpha\psi) \rightarrow t \dashv \psi}\ \alpha \text{ fresh}$$

$$[\text{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \dashv \psi_1 \qquad \Gamma\psi_1 \vdash e_2 : t_2 \dashv \psi_2}{\Gamma \vdash (e_1, e_2) : (t_1\psi_2) \times t_2 \dashv \psi_2 \circ \psi_1}$$

$$[\text{LProj}]\ \frac{\Gamma \vdash e : t \dashv \psi \qquad \psi' = \text{mgu}(t, \alpha_1 \times \alpha_2)}{\Gamma \vdash \pi_1 e : \alpha_1\psi' \dashv \psi' \circ \psi}\ \alpha_1, \alpha_2 \text{ fresh}$$

$$[\text{RProj}]\ \frac{\Gamma \vdash e : t \dashv \psi \qquad \psi' = \text{mgu}(t, \alpha_1 \times \alpha_2)}{\Gamma \vdash \pi_2 e : \alpha_2\psi' \dashv \psi' \circ \psi}\ \alpha_1, \alpha_2 \text{ fresh}$$

Lack of polymorphism

What happens if we try to infer the type of the identity function $\lambda x.x$?

$$\varnothing \vdash \lambda x.x : \alpha \to \alpha \dashv \varnothing$$

We obtain the type $\alpha \to \alpha$. Let's add it to our context:

$$\Gamma \quad = \quad \texttt{id} : \alpha \to \alpha$$

Now, what happens if we try to type (`id 42`, `id false`)?

- The first application `id 42` substitutes α by `int`, yielding a new environment where `id` : `int` → `int`,
- Then, the second application `id false` fails.

Parametric polymorphism (Hindley-Milner)

Compositionality

A type system should be compositional: each definition of your program is typed sequentially, only once, without knowing how it will be used by later definitions.

However, a definition may be use multiple times with arguments of different types:

```
let id x = x (* Type: 'a -> 'a *)
(* ... *)
let foo =
  id 42,  (* 'a should be substituted by int *)
  id true (* 'a should be substituted by bool *)
(* ... *)
let bar x =
  id x    (* 'a should be substituted by the type of x *)
```

⇒ we need a way to use different instantiations of a definition

Let-bindings

In order to model sequential definitions, we add let-bindings to our syntax:

$$\begin{array}{llll} \textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid e\,e \mid (e,e) \mid \pi_1 e \mid \pi_2 e \mid e\,\texttt{?}\,e\,\texttt{:}\,e \\ & & & \mid \texttt{let}\,x = e\,\texttt{in}\,e \\ \textbf{Values} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array}$$

Semantics: we first reduce the definition, then

$$\texttt{let}\,x = v\,\texttt{in}\,e \rightsquigarrow e\{v/x\} \quad \text{Let-reduction}$$

Typing rule (first attempt, no parametric polymorphism):

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\,x = e_1\,\texttt{in}\,e_2 : t}$$

Type Schemes

We want a way to give a polymorphic type to a let-definition, that is, a type that can be instantiated in different ways at different locations.

To that purpose, we define a notion of type scheme:

Base Types	b	$::=$	$\texttt{bool} \mid \texttt{int} \mid \ldots$
Types	s, t	$::=$	$b \mid t \to t \mid t \times t \mid \alpha$
Type Schemes	σ	$::=$	$\forall \vec{\alpha}.\, t$

where $\vec{\alpha}$ is a set of type variables.

Intuitively, a type scheme $\forall \vec{\alpha}.\, t$ represents a set of types:
it represents all the instances of t obtained by substituting the type variables in $\vec{\alpha}$.

$$\forall \vec{\alpha}.\, t \quad \simeq \quad \{ t\psi \mid \psi \in \textbf{Substs},\ \mathrm{dom}(\psi) \subseteq \vec{\alpha} \}$$

In particular, if $\vec{\alpha} = \varnothing$, then $\forall \vec{\alpha}.\, t \simeq \{t\}$.

New Type Environments

We update our type environments Γ to associate variables to type-schemes (instead of types):

$$\begin{array}{llll} \textbf{Type Schemes} & \sigma & ::= & \forall\vec{\alpha}.\ t \\ \textbf{Type Environments} & \Gamma & ::= & \varnothing \mid x : \sigma, \Gamma \end{array}$$

For instance, if $(x : \forall\alpha.\ \alpha \to \alpha) \in \Gamma$,
it means that the variable x can be typed with any type in this set:

$$\{(\alpha \to \alpha)\psi \mid \psi \in \textbf{Substs},\ \mathrm{dom}(\psi) \subseteq \{\alpha\}\} \;=\; \{t \to t \mid t \in \textbf{Types}\}$$

Note: our type environments now associate variables to type-schemes,
but our typing rules still derive a type ($\Gamma \vdash e : t$), not a type-scheme.

New Typing rules

As a variable can now be associated to multiple types, we have to modify the [Var] typing rule: it has to choose one type among those captured by the type-scheme.

$$[\text{Var}]\ \frac{}{\Gamma, x : t \vdash x : t} \quad \longrightarrow \quad [\text{Var}]\ \frac{}{\Gamma, x : \forall \vec{\alpha}.\ t \vdash x : t\psi}\ \text{dom}(\psi) \subseteq \vec{\alpha}$$

We also need to modify the typing rules that add a binding in the environment Γ:

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2\ : t} \quad \longrightarrow \quad \textbf{???}$$

$$[\text{Abs}]\ \frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t} \quad \longrightarrow \quad \textbf{???}$$

Should we allow any type scheme as a domain in [Abs] rules?

$$[\text{Abs}]\ \frac{\Gamma, x : \forall\vec{\alpha}.\ s \vdash e : t}{\Gamma \vdash \lambda x.e : (\forall\vec{\alpha}.\ s) \to t}$$

$(\forall\vec{\alpha}.\ s) \to t$ is not a type...

$\Rightarrow$ We cannot accept quantification in the domain of a λ-abstraction.

Our polymorphism is prenex: there cannot be quantifications inside of a type constructor.

$$[\text{Abs}]\ \frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t} \quad\longrightarrow\quad [\text{Abs}]\ \frac{\Gamma, x : \forall\varnothing.\ s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t}$$

What about [Let] rules?

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\, x = e_1\ \texttt{in}\ e_2 \ : t}$$

Let's say e_1 is the identity $\lambda y.y$, for which we derive the type $\alpha \to \alpha$.
When typing the rest of the program (i.e., e_2),
x may be called on arguments of different types
$\Rightarrow$ x should be associated to the type-scheme $\forall \alpha.\, \alpha \to \alpha$

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : \text{generalize}(s) \vdash e_2 : t}{\Gamma \vdash \texttt{let}\, x = e_1\ \texttt{in}\ e_2 \ : t}$$

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : \text{generalize}(s) \vdash e_2 : t}{\Gamma \vdash \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 : t}$$

Should generalize(s) quantify over all type variables in s?

$$[\text{Abs}]\ \dfrac{[\text{Let}]\ \dfrac{[\text{Var}]\ \dfrac{}{\Gamma, y : \forall\varnothing.\ \alpha \vdash y : \alpha} \qquad [\text{Var}]\ \dfrac{}{\Gamma, y : \forall\varnothing.\ \alpha, x : \forall\alpha.\ \alpha \vdash x : \texttt{int}}}{\Gamma, y : \forall\varnothing.\ \alpha \vdash \texttt{let}\, x = y \,\texttt{in}\, x : \texttt{int}}}{\Gamma \vdash \lambda y.\ \texttt{let}\, x = y \,\texttt{in}\, x : \alpha \to \texttt{int}}$$

No, it is **unsound** to generalize type variables that are bound to the current environment Γ. We should only generalize type variables in $\text{fv}(s) \setminus \text{fv}(\Gamma)$.

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : \forall(\text{fv}(s) \setminus \text{fv}(\Gamma)).\ s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 : t}$$

Exercise

Find a derivation for the judgment $\varnothing \vdash \texttt{let}\, x = \lambda y.y \,\texttt{in}\, (x\ 42, x\ \texttt{true}) : \texttt{int} \times \texttt{bool}$.

$$[\mathrm{Int}]\ \frac{}{\Gamma \vdash n : \texttt{int}} \qquad [\mathrm{Bool}]\ \frac{}{\Gamma \vdash b : \texttt{bool}} \qquad [\mathrm{Var}]\ \frac{}{\Gamma, x : \forall \vec{\alpha}.\ t \vdash x : t\psi}\ \mathrm{dom}(\psi) \subseteq \vec{\alpha}$$

$$[\mathrm{Abs}]\ \frac{\Gamma, x : \forall \varnothing.\ s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t} \qquad [\mathrm{App}]\ \frac{\Gamma \vdash e_1 : s \to t \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ e_2 : t}$$

$$[\mathrm{LProj}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\mathrm{RProj}]\ \frac{e : t_1 \times t_2}{\pi_2 e : t_2} \qquad [\mathrm{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\mathrm{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : \forall (\mathrm{fv}(s) \setminus \mathrm{fv}(\Gamma)).\ s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 : t}$$

Type inference

Two rules are not analytic (i.e. we have to guess a type or substitution):

$$[\text{Abs}]\ \frac{\Gamma, x : \forall\varnothing.\ s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t}$$

$$[\text{Var}]\ \frac{}{\Gamma, x : \forall\vec{\alpha}.\ t \vdash x : t\psi}\ \text{dom}(\psi) \subseteq \vec{\alpha}$$

Similarly to STLC, we can infer the domain of a λ-abstractions and the instantiations of variables using unification.

$$\Gamma \vdash e : t \dashv \psi$$

$$[\text{App}]\ \frac{\Gamma \vdash e_1 : t_1 \dashv \psi_1 \qquad \Gamma\psi_1 \vdash e_2 : t_2 \dashv \psi_2 \qquad \psi = \text{mgu}(t_1\psi_2, t_2 \to \alpha)}{\Gamma \vdash e_1\ e_2 : \alpha\psi \dashv \psi \circ \psi_2 \circ \psi_1}\ \alpha \text{ fresh}$$

$$[\text{Abs}]\ \frac{\Gamma, x : \forall\varnothing.\ \alpha \vdash e : t \dashv \psi}{\Gamma \vdash \lambda x.e : (\alpha\psi) \to t \dashv \psi}\ \alpha \text{ fresh}$$

$$[\text{Var}]\ \frac{}{\Gamma, x : \forall\vec{\alpha}.\ t \vdash x : t\psi \dashv \varnothing}\ \psi \text{ maps type variables in } \vec{\alpha} \text{ to fresh ones}$$

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \dashv \psi_1 \qquad \Gamma\psi_1, x : \forall(\text{fv}(s) \setminus \text{fv}(\Gamma)).\ s \vdash e_2 : t \dashv \psi_2}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : t \dashv \psi_2 \circ \psi_1}$$

Exercise: implement a Hindley-Milner type system following these typing rules.

Recap

- **Simply Typed Lambda Calculus** (STLC):
  base types, arrows, products, type variables (but no quantifier)
  $\Rightarrow$ Inference of the domain of functions is possible using unification
- **Hindley-Milner** (HM): type environments can now quantify universally ($\forall$)
  over some type variables using type-schemes
  $\Rightarrow$ Type inference is almost unchanged
- **System F**: generalization of Hindley-Milner,
  where $\forall$ quantifiers can appear inside type constructors (in particular, arrows)
  $\Rightarrow$ Type inference is not decidable anymore (unless we add some restrictions)

Towards an imperative language

Our λ-calculus is pure, in particular let-variables cannot be reassigned.

Let us extend our language and types with references (i.e. mutable memory cells):

$$
\begin{array}{llll}
\textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid e\,e \mid (e,e) \mid \pi_1 e \mid \pi_2 e \mid e\,\texttt{?}\,e\,\texttt{:}\,e \\
 & & & \mid \texttt{let}\,x\,\texttt{=}\,e\,\texttt{in}\,e \mid \texttt{ref}\,e \mid \texttt{!}e \mid x\,\texttt{:=}\,e \\
\textbf{Values} & v & ::= & c \mid \lambda x.e \mid (v,v) \\
\textbf{Types} & s,t & ::= & b \mid t \to t \mid t \times t \mid \alpha \mid \texttt{ref}\,t
\end{array}
$$

We can then encode mutable variables with references:

```
mut v = 15 ; v = v + 42 ; ...
```

$$\downarrow$$

```
let v = (ref 15) in let _ = (v := !v + 42) in ...
```

$$[\text{Ref}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{ref}\ e : \texttt{ref}\ t} \qquad [\text{Read}]\ \frac{\Gamma \vdash e : \texttt{ref}\ t}{\Gamma \vdash \texttt{!}e : t} \qquad [\text{Assign}]\ \frac{\Gamma \vdash x : \texttt{ref}\ t \qquad \Gamma \vdash e : t}{\Gamma \vdash x := e : t}$$

But we have to be careful not to generalize the type of expressions with side-effects:

```
let foo = ref (fun x -> x) in
(* foo: forall 'a. ref('a -> 'a) *)

foo := (fun i -> i + 42) ;
(* Typechecks by instantiating 'a with int *)

!foo false  (* Reduction stuck! false + 42 *)
(* Typechecks by instantiating 'a with false *)
```

$$[\text{Ref}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{ref}\ e : \texttt{ref}\ t} \qquad [\text{Read}]\ \frac{\Gamma \vdash e : \texttt{ref}\ t}{\Gamma \vdash\ !e : t} \qquad [\text{Assign}]\ \frac{\Gamma \vdash x : \texttt{ref}\ t \qquad \Gamma \vdash e : t}{\Gamma \vdash x := e : t}$$

Solution: only generalize values (value restriction).

$$[\text{LetGen}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : \forall(\text{fv}(s) \setminus \text{fv}(\Gamma)).\, s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t}\ \text{if } e_1 \text{ is a value}$$

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma, x : \forall\varnothing.\, s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t}\ \text{otherwise}$$

Subtyping and ad-hoc polymorphism

What is the type of this function?

```
def inv(x:float):
  if x == 0.0:
    return None
  else:
    return 1/x
```

$\texttt{float} \to (\texttt{float} \vee \texttt{none})$

What is the type of this one?

```
def f(x:Union[float, NoneType]):
  if x == None:
    return 0.0
  else:
    return x
```

$(\texttt{float} \vee \texttt{none}) \to \texttt{float}$

Subtyping

In dynamic languages, functions can manipulate data of heterogeneous types

$\Rightarrow$ Hence we need to be able to express the union of two types $t_1 \vee t_2$

$\Rightarrow$ We may also need to express an intersection, e.g. $\texttt{printable} \wedge \texttt{iterable}$

- The type `float` should be usable everywhere a $\texttt{float} \vee \texttt{none}$ is expected
- The type `none` should be usable everywhere a $\texttt{float} \vee \texttt{none}$ is expected

Formally, we define a subtyping relation $\leq$ such that:

- $\leq$ is reflexive $(t \leq t)$ and transitive $(t_1 \leq t_2$ and $t_2 \leq t_3$ implies $t_1 \leq t_3)$,
- for any t_1 and t_2, $t_1 \leq t_1 \vee t_2$ and $t_2 \leq t_1 \vee t_2$,
- other properties may be desirable, for instance idempotency $(t \simeq t \wedge t)$

Overloaded functions

What is the type of this function?

```
def inv(x):
  if isinstance(x, complex):
    return ...
  elif isinstance(x, float):
    return ...
```

$$(\texttt{complex} \vee \texttt{float}) \to (\texttt{complex} \vee \texttt{float})$$

or if we want to be more precise

$$(\texttt{complex} \to \texttt{complex}) \wedge (\texttt{float} \to \texttt{float})$$

Parametric polymorphism vs ad-hoc polymorphism

Parametric polymorphism: captures genericity of functions: when the function has the same behavior on (potentially infinitely many) different input types.
Implemented by quantifying on type variables (e.g. $\forall\alpha.\ \alpha \to \alpha$).

Ad-hoc polymorphism: captures overloading of functions: when the function has (finitely many) different behaviors depending on the input type.
Implemented by associating multiple signatures to a top-level function, or using intersection types (e.g. $(\texttt{int} \to \texttt{bool}) \wedge (\texttt{string} \to \texttt{int})$).

Existing approaches

To be continued ...

References i