CS 456

Programming Languages
Fall 2024

Week 6

Type Inference

Inference

- How should we fill in these type annotations?

λx:□. **if** x **then** x **else** false

λx:□. λy:□. **if** x **then** y + 1 else y

(λx:□. λy:□. **if** x **then** y **else** y) true 1

λx:□. λy:□. **if** x **then** y **else** y

Type Inference

- More interesting question is how to avoid annotations if possible?
- **Today**: A **type inference** algorithm *infers* the **principal type** of a term missing some type annotations.
  - Such algorithms are key to OCaml's type system:

```
fold f acc [ ] = acc
fold f acc (x :: xs) = f x (fold f acc xs)


map (fun x -> x + 4) [1; 2]
```

Type Variables

- First step: extend STLC with Type Variables:

  $n \in \mathbb{N}$ $\quad\quad\quad\quad$ $X_? \in \mathrm{TypeVariables}$

  $T ::= \mathrm{Nat} \mid \mathrm{Bool} \mid T \to T \mid X_?$

  $t ::= x \mid \lambda x : T.\ t \mid t\ t \mid \mathtt{n} \mid t + t$
  $\mid \mathrm{true} \mid \mathrm{false} \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t$

- Typing rules and Operational Semantics are **same** as before:

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \textsc{TApp} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textsc{TVar}$$

$$\frac{\Gamma[x \mapsto T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2}\ \textsc{TAbs}$$

...

Type Inference

- Two ways to interpret a term with type variables:
  1. Are *all* instantiations well-typed terms?
     - λx:$Y_?$ → Bool. λy: $Y_?$. x y : ($Y_?$ → Bool) → $Y_?$ → Bool
  2. Is *some* instantiation a well-typed term?
     - λx:$X_?$. λy: $Y_?$. x (x y) : □
- Represent 'missing' type annotations with Type Variables:
  - ✗ λx. λy:Bool. **if** x y **then** false **else** true
  - ✓ λx:$X_?$. λy:Bool. **if** x y **then** false **else** true
- **<u>Our Goal</u>**: Build a **well-typed term** by filling or **substituting** in **concrete types** for type variables:
  - λx:Bool → Bool. λy:Bool. **if** x y **then** false **else** true

Type Substitution

- A **type substitution** is a mapping, γ, from variables to types:
  - $[Y_? \mapsto Bool, X_? \mapsto Bool \to Bool]$
  - $[X_? \mapsto Bool \to Bool, Y_? \mapsto X_?]$
- We apply a type substitution to a type T like so:

$$\gamma(X_?) \equiv T \quad \text{if } (X_? \mapsto T) \in \gamma$$
$$\gamma(X_?) \equiv X_? \quad \text{if } X_? \notin \gamma$$
$$\gamma(Bool) \equiv Bool \qquad \gamma(Nat) \equiv Nat$$
$$\gamma(T_1 \to T_2) \equiv \gamma(T_1) \to \gamma(T_1)$$

- Examples Application:

$$(Y_? \to X_?)[X_? \mapsto Bool \to Bool, Y_? \mapsto Bool] \equiv Bool \to (Bool \to Bool)$$
$$(Y_? \to X_?)[X_? \mapsto Bool \to Bool, Y_? \mapsto X_?] \equiv X_? \to (Bool \to Bool)$$

Type Substitution

- **Theorem**: Type substitution preserves typing: for every type substitution γ, if $\Gamma \vdash e : T$, then $\gamma(\Gamma) \vdash \gamma(e) : \gamma(T)$.

- A **solution** for a context Γ and term e is a type T and a substitution γ such that:

$$\gamma(\Gamma) \vdash \gamma(e) : \gamma(T)$$

- For the empty context, $\lambda x{:}X_?.\ \lambda y{:}Y_?.\ x\ (x\ y)$, a solution is:
  - **Type**: $X_? \to Y_? \to Y_?$
  - **Substitution**: $[X_? \mapsto Y_? \to Y_?]$

Concept Check

- A solution for a context Γ and term e is a type T and a substitution γ such that:

$$\gamma(\Gamma) \vdash \gamma(e) : \gamma(T)$$

- Can you find two solutions for the empty context and the term:

$$\lambda x{:}X_?.\ \lambda y{:}Y_?.\ \lambda z{:}Z_?.\ \textbf{if}\ y\ \textbf{then}\ x\ z\ \textbf{else}\ z$$

Type Inference

Algorithm InferType(Γ, e_{in})

Input: Typing Context Γ, Untyped Lambda term e_{in}

Output: Well-typed STLC term or ill-typed

1. e_1 ← **annotate** all lambda abstractions in e_{in} with fresh Type Variables;
2. (T, ξ) ← **calculate type and constraints** that *any* solution for Γ and e_1 must satisfy
3. γ ← **find solution** to ξ, **or** report none exists (⊥)
4. **if** γ == ⊥ **then return** ill-typed
5. **return** $\gamma(\Gamma) \vdash \gamma(e_1) : \gamma(T)$

Type Inference

Algorithm InferType(Γ, e_{in})

Input: Typing Conte...

e_{in}

...esh

...constraints that *any*

...e_1 must satisfy

3. ...find solution to ξ, **or** report none exists (⊥)
4. **if** γ == ⊥ **then return** ill-typed

★ Since typing does not affect dynamic behavior, e_{in} is guaranteed to not get stuck if InferType returns a well-typed term!

Constraint-Based Typing

- **Key Idea$_1$**: record a set of ***constraints*** about how variables are used, and figure out how to solve them **later**
- Types constrain how things can be used:
  - The condition of an **if** expression must have type bool
  - Only expressions of type nat can be added together
- Formally, we define a new typing algorithm with the following judgement:

$$\Gamma \vdash e : T \mid \varnothing$$

Constraint-Based Typing

- Here are the rules for this type system:
  - Expressions which don't 'use' anything don't impose any new constraints:

$$\frac{}{\Gamma \vdash \textbf{true} : \text{Bool} \mid \varnothing}\ \text{CTTRUE}$$

$$\frac{\Gamma(x) : T}{\Gamma \vdash x : T \mid \varnothing}\ \text{CTVAR}$$

$$\frac{}{\Gamma \vdash \textbf{false} : \text{Bool} \mid_{\varnothing} \varnothing}\ \text{CTFALSE}$$

$$\frac{n \in \mathbb{N}}{\Gamma \vdash \textbf{n} : \text{Nat} \mid \varnothing}\ \text{CTNUM}$$

$$\frac{\Gamma,[x \mapsto T_1] \vdash t : T_2 \mid C}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \rightarrow T_2 \mid C}\ \text{CTABS}$$

Constraint-Based Typing

Standard rule

$$\frac{\Gamma \vdash e_1 : nat \quad \Gamma \vdash e_2 : nat}{\Gamma \vdash e_1 + e_2 : nat} \text{ TADD}$$

Constraint-based

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \quad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : nat \mid C_1 \cup C_2 \cup \{T_1 = nat, T_2 = nat\}}$$

Constraint-Based Typing

Standard rule

$$\frac{\Gamma \vdash e_c : \text{Bool} \qquad \Gamma \vdash e_t : T \qquad \Gamma \vdash e_e : T}{\Gamma \vdash \text{if } e_c \text{ then } e_t \text{ else } e_e : T} \quad \text{TIF}$$

Constraint-based

$C = C_c \cup C_t \cup C_e \cup \{T_c = \text{Bool},\ T_t = T_e\}$

Type variables in C do not overlap

$$\frac{\Gamma \vdash e_c : T_c \mid C_c \qquad \Gamma \vdash e_t : T_t \mid C_t \qquad \Gamma \vdash e_e : T_e \mid C_e}{\Gamma \vdash \text{if } e_c \text{ then } e_t \text{ else } e_e : T_t \mid C} \quad \text{CTIF}$$

Constraint-Based Typing

Standard rule

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2} \quad \text{TAPP}$$

Constraint-based

Type Variables in $FV(T_2)$, $FV(T_1)$, C_1, C_2, t_1, t_2 and Γ don't overlap

$X \notin FV(T_2), FV(T_1), C_1, C_2, t_1, t_2$ or Γ $\qquad C = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad \Gamma \vdash t_2 : T_2 \mid C_2}{\Gamma \vdash t_1\, t_2 : X \mid C} \quad \text{CTApp}$$

Concept Check

What is the constrained type for:

$$\lambda x{:}X.\ \lambda y{:}Y.\ \lambda z{:}Z.\ x\ (y\ z)$$

Implicit Type Annotations

- These rules gives us an algorithm for type reconstruction for an expression e in the (unannotated) lambda calculus:
  - Add a (fresh) type variable to every lambda term in e
  - Use constraint-based typing rules to gather constraints
  - Find a solution

- An alternative: Add a typing rule for unannotated lambda terms

$$\frac{X \notin T_1 \text{ or } C \qquad \Gamma, [x \mapsto X] \vdash t : T_2 \mid C}{\Gamma \vdash \lambda x.t : X \to T_2 \mid C} \quad \text{CTABS}$$

Solving Constraints

- Note that this algorithm never fails: it *always* returns a set of constraints:
  - $\vdash(\lambda x{:}\mathrm{Bool}.x)(\lambda y{:}\mathrm{Bool}.y) : Z_? \mid \{\mathrm{Bool}\to \mathrm{Bool} = \mathrm{Bool}\to\mathrm{Bool}\to Z_?\}$
  - $\vdash\lambda x{:}X_?.\ x\ x : X_?\to Z_? \mid \{X_? = X_?\to Z_?\}$
- Step 2 is to find a solution to the results of constraint-based rules
  - A solution to $\Gamma \vdash e{:}T \mid C$ is a type S and a substitution γ such that γ is ***consistent*** with C and γ T = S.
  - A substitution is **consistent** with a constraint if it applying makes both sides of the equation exactly the *same*, i.e. unifies them.

Solving Constraints

- Step 2 is to find a solution to the results of constraint-based rules
  - A solution to Γ ⊢ e: T | C is a type S and a substitution γ such that γ is **consistent** with C and γ T = S.
- A solution to:

```
λx:X?. λy:Y?. λz:Z?. (x z) (y z) : X?→Y?→Z?→R?
|  {X?=Z?→Q?, Y?=Z?→P?, Q?=P?→R?} is:
[X?↦ Z?→P?→ R?, Y? ↦ Z?→ P?] and the type
(Z?→P?→ R?) → (Z?→P?) → Z? → R?.
```

Concept Check

- What is a solution to the constraints generated by:

```
(λx: X?. λy: Y?. if x then y + 1 else y)
```

Sensibility of Approach

- Let's take a step back and ask when this makes sense.
  - How does this relate to the original type system?
- **Theorem**: Constraint typing is sound. That is, if $\Gamma \vdash e: T \mid C$, then any solution S and γ must also be a solution for Γ and e.
- **Theorem**: Constraint typing is complete. That is, if S and γ are a solution for e and Γ and $\Gamma \vdash e: T \mid C$, then if γ and the type variables in C do not overlap, there must exist some solution for the original typing derivation, γ_2 and S'.
- **Theorem**: Constraint typing is sane: there is a solution to $\Gamma \vdash e: T \mid C$ if and only if there is a solution to Γ and e.