CSE114A - lecture 16!

Today: type inference and polymorphism

$$\Gamma \vdash e :: T$$

gamma, type environment — expr — type

ex: $[(x, Int), (y, Int \to Int), \ldots]$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n :: Int} [\text{T-Num}] \qquad \frac{\Gamma \vdash e_1 :: Int \quad \Gamma \vdash e_2 :: Int}{\Gamma \vdash e_1 + e_2 :: Int} [\text{T-Add}]$$

$$\frac{\text{if } (x, T) \text{ is in } \Gamma}{\Gamma \vdash x :: T} [\text{T-Var}] \qquad \frac{\Gamma, (x, T_1) \vdash e :: T_2}{\Gamma \vdash \backslash x \to e :: T_1 \to T_2} [\text{T-Lam}]$$

$$\frac{\Gamma \vdash e_1 :: T_1 \to T_2 \quad \Gamma \vdash e_2 :: T_1}{\Gamma \vdash e_1 \ e_2 :: T_2} [\text{T-App}]$$

$$\frac{\Gamma \vdash e_1 :: T_1 \qquad \Gamma, (x, T_1) \vdash e_2 :: T_2}{\Gamma \vdash \text{let } \underbrace{x = e_1} \text{ in } e_2 :: \boxed{T_2}} [\text{T-Let}]$$

$$\boxed{\Gamma \vdash e :: T}$$

we've defined this relation.

An expression is well-typed in Γ if we can derive $\Gamma \vdash e :: T$ for some type T.

If we can't do that, the expression is ill-typed.

In general, the idea here is to figure out the types of expressions without evaluating any expressions — rather, just by using the typing rules.

example: let x = 1 in x + 2

$$\dfrac{\dfrac{1 \in \mathbb{Z}}{[\,] \vdash 1 :: \boxed{\text{Int}}}\,[\text{T-Num}] \qquad \dfrac{\dfrac{(x,\text{Int}) \text{ is in } [(x,\text{Int})]}{[(x,\text{Int})] \vdash x :: \boxed{\text{Int}}}\,[\text{T-Var}] \quad \dfrac{2 \in \mathbb{Z}}{[(x,\text{Int})] \vdash 2 :: \boxed{\text{Int}}}\,[\text{T-Num}]}{[(x,\text{Int})] \vdash x+2 :: \boxed{\text{Int}}}\,[\text{T-Add}]}{[\,] \vdash \text{let } x=1 \text{ in } x+2 :: \boxed{\text{Int}}}\,[\text{T-Let}]$$

---

example: (\x → x) 2

$$\dfrac{\dfrac{\dfrac{(x, \boxed{\text{Int}}) \text{ is in } [(x,\text{Int})]}{[(x,\text{Int})] \vdash x :: \boxed{\text{Int}}}\,[\text{T-Var}]}{[\,] \vdash \backslash x \to x :: \boxed{\text{Int} \to \text{Int}}}\,[\text{T-Lam}] \qquad \dfrac{2 \in \mathbb{Z}}{[\,] \vdash 2 :: \text{Int}}\,[\text{T-Num}]}{[\,] \vdash (\backslash x \to x)\ 2 :: \boxed{\text{Int}}}\,[\text{T-App}]$$

---

example: (\x → x x) is ill-typed.

$$\dfrac{\dfrac{\dfrac{(x,T_1) \text{ is in } [(x,T_1)]}{[(x,T_1)] \vdash x :: \boxed{T_1}}\,[\text{T-Var}] \quad \dfrac{(x,T_1) \text{ is in } [(x,T_1)]}{[(x,T_1)] \vdash x :: \boxed{T_1}}\,[\text{T-Var}]}{[(x,T_1)] \vdash x\ x :: \boxed{\text{:(}}}\,[\text{T-App}]}{[\,] \vdash (\backslash x \to x\ x) :: \boxed{\text{:(}}}\,[\text{T-Lam}]$$

What about \x → x ? What's its type?

we could use the rules to derive
* Int → Int,
* (Int → Int) → (Int → Int),

or T → T for any T.

What we really want, though, is to derive a single, most general type for every expresssion.

```
let f = \x -> x in
  let y = f 5 in
    f (\z -> z + y)
```

What should the type here be?
Int → Int...

but the typing rules (so far) say this is ill-typed!

\z → z+5

The problem here is that our typing rules forced us to pick only one type for f, even though different uses of it call for different types.

\x → x :: forall a . a → a

This is the real type of \x → x, also known as a type scheme or just a polytype.

We can instantiate a type scheme into different types, by replacing the bound type variable (in this case, a) with some type.

instantiating forall a . a → a with Int gives us Int → Int.

instantiating forall a . a → a with Int → Int gives us (Int → Int) → (Int → Int)

we need to tweak our rules to allow this.

At a high level, type inference works as follows. (for this example)

1) when we have to pick a type T for a variable x, we pick a fresh type variable a.
2) So, the type of \x → x comes out as a → a
3) we generalize this to forall a. a → a

F = \x → x

4) when we apply f, we instantiate it with Int or Int → Int as needed for each application.

Let's make this happen:

$T ::= \text{Int} \mid T_1 \to T_2 \mid a$

type variables.

we now also have type schemes, aka polytypes

$S ::= T \mid \text{forall } a \,.\, S$

[(z, Int), (f, forall a. a → a)]

Type environments can now contain polytypes.

we now need a notion of <u>type substitutions</u>.

A type substitution maps type variables to types.

example: [("a", Int), ("b", c → c)]

<u>Applying</u> a type substitution to a type T means replacing all the type variables in T with whatever the type substitution binds them to.

[("a", Int), ("b", c → c)] (a → a)
would give me: Int → Int

[("a", Int) ("b", c → c)] (b → c)
would give me: (c → c) → c

and so on.

we'll change the T-Var rule and the T-Let rule to talk about type schemes.

$$\frac{(x, \boxed{S}) \text{ is in } \Gamma}{\Gamma \vdash x :: \boxed{S}} \text{[T-Var]}$$

$$\frac{\Gamma \vdash e_1 :: \boxed{S} \qquad \Gamma, (x, \boxed{S}) \vdash e_2 :: T}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 :: T} \text{[T-Let]}$$

Two new rules that let us
- <u>generalize</u> a type into a type scheme
- <u>instantiate</u> a type scheme into a type

$$\frac{\Gamma \vdash e :: \text{forall } a.\ S}{\Gamma \vdash e :: S} \text{[T-Inst]}$$

but with the type substitution [("a", T)] applied for some T.

e.g. forall a. a → a
could become Int → Int.

$$\frac{\Gamma \vdash e :: S}{\Gamma \vdash e :: \text{forall } a.\ S} \text{[T-Gen]}$$

where a is a fresh type variable not occurring in Γ.

Let's do an example:
what's the type of \x → x ?
(we want it to be forall a. a → a)

$$\cfrac{\cfrac{\cfrac{(x, a) \text{ is in } [(x, a)]}{[(x, a)] \vdash x :: a} \text{[T-Var]}}{[\,] \vdash \backslash x \to x :: a \to a} \text{[T-Lam]}}{[\,] \vdash \backslash x \to x :: \boxed{\text{forall } a.\ a \to a}\ ?} \text{[T-Gen]}$$

let f = \x → x in
let y = f 5 in
f (\z → z+y)

should have type:
Int → Int
Let's try to derive this.

(f, forall...) is in [(f, forall...), (y, Int)]
——————————————— [T-Var]
[(f, forall...), (y, Int)] ⊢ f :: forall a. a → a
——————————————— [T-Inst]
[(f, forall...), (y, Int)] ⊢ f :: $\boxed{(\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})}$

(easy use of T-Lam and T-Add)
———————————————
[(f, forall...), (y, Int)] ⊢ \z → z+y :: Int → Int

——————————————— [T-App]
[(f, forall...), (y, Int)] ⊢ f(\z → z+y) :: $\boxed{\text{Int} \to \text{Int}}$

(f, forall...) is in [(f, forall...)]
——————————————— [T-Var]
[(f, forall...)] ⊢ f :: forall a...
——————————————— [T-Inst]
[(f, forall...)] ⊢ f :: $\boxed{\text{Int} \to \text{Int}}$

5 ∈ ℤ
——————————————— [T-Num]
[(f, forall...)] ⊢ 5 :: Int

——————————————— [T-App]
[(f, forall a. a → a)] ⊢ f 5 :: $\boxed{\text{Int}}$

——————————————— [T-Let]
[(f, forall a. a → a)] ⊢ let y = f 5 in ... :: $\boxed{\text{Int} \to \text{Int}}$

(previous example)
———————————————
[] ⊢ \x → x :: forall a. a → a

——————————————— [T-Let]
[] ⊢ let f = \x → x in ... :: $\boxed{\text{Int} \to \text{Int}}$

<u>Key idea</u> of the previous typing derivation:
We used the [T-Inst] rule twice. Once we used it to instantiate forall a. $a \to a$ with Int, resulting in $Int \to Int$. And once we used it to instantiate forall a. $a \to a$ with $Int \to Int$, resulting in $(Int \to Int) \to (Int \to Int)$.

The binding (f, forall a. $a \to a$) was in our type environment, and every time we used f from the type environment (with the [T-Var] rule), we used [T-Inst] to instantiate it as needed!