CSE114A Lecture 16

agenda:

- recap of lecture 15
  - typing judgements & typing rules
  - Nano's type system
  - type derivations
- How the above connects to hw5 (essentially implementing type inference using these typing rules!) $\longleftarrow$ part 3 of hw5
- <u>unification</u> (part 2 of hw5) and how it plays into the above
- do a little hacking on parts 1 & 2 of hw5

Recap of last time:

$$\boxed{\Gamma \overset{\text{turnstile, sometimes pronounced "entails"}}{\vdash} e :: T} \leftarrow \text{"typing judgment"}$$

"In type env Γ, e has type T"

"gamma entails that e has type T".

Typing rules for (mini-) Nano:

$e ::= n \mid x \mid e_1 + e_2 \mid \backslash x \to e \mid e_1\ e_2 \mid \text{let } x = e_1 \text{ in } e_2$

✓ ✓ ✓ ✓ ✓ ✓

$$\frac{n \in \mathbb{N}}{\Gamma \vdash n :: \text{Int}}\ \boxed{\text{TNum}}$$

$$\frac{b \in \{\text{True}, \text{False}\}}{\Gamma \vdash b :: \text{Bool}}\ \boxed{\text{TBool}}$$

$$\frac{\boxed{\Gamma \vdash e_1 :: \text{Int}} \quad \boxed{\Gamma \vdash e_2 :: \text{Int}}}{\boxed{\Gamma \vdash \boxed{e_1 + e_2} :: \text{Int}}}\ \boxed{\text{TPlus}}$$

Notice that the judgment below the line depends on two judgments above the line. This suggests a need for recusive calls on subexpressions in your implementation of <u>infer</u>.

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x :: T}\ \boxed{\text{TVar}}$$

$$\frac{\Gamma(x, T_1) \vdash e :: T_2}{\Gamma \vdash (\backslash x \to e) :: T_1 \to T_2}\ \boxed{\text{TLam}}$$

example:

$\backslash x \to \underline{x + 1} :: \boxed{\text{Int} \to \text{Int}}$

$$\frac{\Gamma \vdash e_1 :: T_1 \to T_2 \quad \Gamma \vdash e_2 :: T_1}{\Gamma \vdash \underline{e_1\ e_2} :: T_2}\ \boxed{\text{TApp}}$$

$$\frac{\Gamma \vdash e_1 :: T_1 \quad \boxed{\overset{\text{extended type environment with } (x, T_1).}{\boxed{\Gamma, (x, T_1)}} \vdash e_2 :: T_2}}{\Gamma \vdash \text{let } \underline{x} = \textcircled{e_1} \text{ in } e_2 :: T_2}\ \boxed{\text{TLet}}$$

eval $\boxed{\text{let } x = \textcircled{5} \text{ in } \underbrace{\boxed{x + 2}}_{\text{body}}} \Rightarrow 7$

$$\boxed{\text{let } f = \underbrace{\boxed{\backslash x \to x + 2}}_{\text{Int} \to \text{Int}} \text{ in } \underbrace{\boxed{f\ 3}}_{\text{Int}}} :: \text{Int}$$

let's do a typing derivation involving let

$$\cfrac{\cfrac{\overset{\checkmark}{1 \in \mathbb{N}}}{[\,] \vdash 1 :: \boxed{Int}}\,\text{TNum} \quad \cfrac{\cfrac{\overset{\checkmark}{(x,Int) \in [(x,Int)]}}{[(x,Int)] \vdash x :: Int}\,\text{TVar} \quad \cfrac{\overset{\checkmark}{2 \in \mathbb{N}}}{[(x,Int)] \vdash 2 :: Int}\,\text{TNum}}{[(x,\boxed{Int})] \vdash \underbrace{x+2} :: \boxed{Int}}\,\text{TPlus}}{[\,] \vdash \underbrace{let\ x = 1\ in\ x+2} :: \boxed{Int}}\,\boxed{\text{TLet}}$$

↑ empty type env.

What about an expression that shouldn't type check?

x cannot be type Int and also have a function type!

must be the same

$$\cfrac{\cfrac{[(y,\square)] \vdash y :: \square}{[\,] \vdash \backslash y \to y :: \square \to \square}\,\text{TLam} \quad \cfrac{\boxed{[(x,\square\to\square)] \vdash x :: Int} \quad [(x,\square\to\square)] \vdash 3 :: Int}{[(x,\square\to\square)] \vdash x+3 :: \square}}{[\,] \vdash let\ x = \underbrace{\backslash y \to y}\ in\ \underbrace{x+3} :: \square}\,\text{TLet}$$

must be the same

Notice that to determine if an expression is well-typed, we have to ensure that various constraints on subexpressions hold. In particular, we may have to ensure that different subexpressions have the same type.

We're going to do this using something called <u>unification</u> (part 2 of hw5)

We need the concept of a <u>substitution</u>: a mapping of type variables to types.

[(a, Int), (b, Int → Int), (c, Bool)]

↑ Substitution mapping a, b, and c to concrete types.

[(a, Int) (b, c → c)]

↑ Type variables may occur in the types on the right hand side of these pairs, as well.

<u>Unification</u> is finding a substitution that makes two types "the same".

Type 1	Type 2	Substitution that unifies them
Int → Bool	a → b	[(a, Int), (b, Bool)]

(we say that [(a, Int), (b, Bool)] unifies Int → Bool and a → b because if we <u>apply</u> the substitution to the types, we end up with Int → Bool for both.

Applying a substitution to a type just means replacing every type variable in the type with whatever the substitution maps it to.

Type 1	Type 2	Substitution that unifies them
Int → Bool	a → b	most general unifier → [(a, Int), (b, Bool)] (or [(a, Int), (b, Bool), (c, Int)])
Int → a	b → Bool	[(a, Bool), (b, Int)]
Int → a	c	[(c, Int → a)] ↑ perfectly fine to have type variables occuring here.
a	b	[(a, b)] or [(b, a)]
Int	Bool	nope ☹
Int → a	Int	nope ☹
a → a	a	nope ☹
[Int]	[a]	[(a, Int)]
a	[a]	nope ☹
a	[Bool]	[(a, [Bool])]

Int → a
<u>b → Bool</u>

Apply [(a, Bool), (b, Int)] to both:

Int → Bool
Int → Bool

So these unify!