CSE114A Lecture 15

agenda:

- types and type inference: intro
- Nano's type system

A mini-version of Nano:

Syntax:

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

(what's missing: Booleans, primitive functions, subtraction, etc., lists)

What types can expressions in this mini-Nano language have?

Syntax of types:

$$T ::= \text{Int} \mid T_1 \rightarrow T_2 \qquad \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$

Int, Int → Int, Int → (Int → Int), (Int → Int) → (Int → Int)

We want to define a typing relation for mini-Nano

$$\Gamma \vdash e :: T$$ ← "In the type environment Γ, e has type T."

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

← premise (above the line), ← conclusion (below the line)

$\Gamma = [(x, \text{Int}), (f, \text{Int} \rightarrow \text{Int})]$

(examples: 3 :: Int, 5000000 :: Int)

$$\frac{\Gamma \vdash e_1 :: \text{Int} \quad \Gamma \vdash e_2 :: \text{Int}}{\Gamma \vdash (e_1 + e_2) :: \text{Int}} \text{ TAdd}$$

← premises (above the line), ← conclusion (below the line)

(examples: 3 + 4 :: Int, (3 + 5) + 6 :: Int, x + 6 :: Int (as long as (x, Int) is in Γ))

anatomy of an inference rule:

$$\frac{\text{premise 1} \quad \text{premise 2} \quad \cdots \quad \text{premise N}}{\text{conclusion}}$$

If the premises are true, then so is the conclusion!

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

(Γ = type environment)

environment binding variables to values: [(x, 5), (f, closure...), (y, 7)]

type environment binding program variables to types: [(x, Int), (f, Int → Int), (y, Int)]

(traditional name: gamma) Γ

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

($\Gamma, (x, T_1)$: extending the type environment with knowledge that x has type T_1.)

$(\lambda x. e_1)\ e_2 \rightarrow_\beta e_1 [x := e_2]$

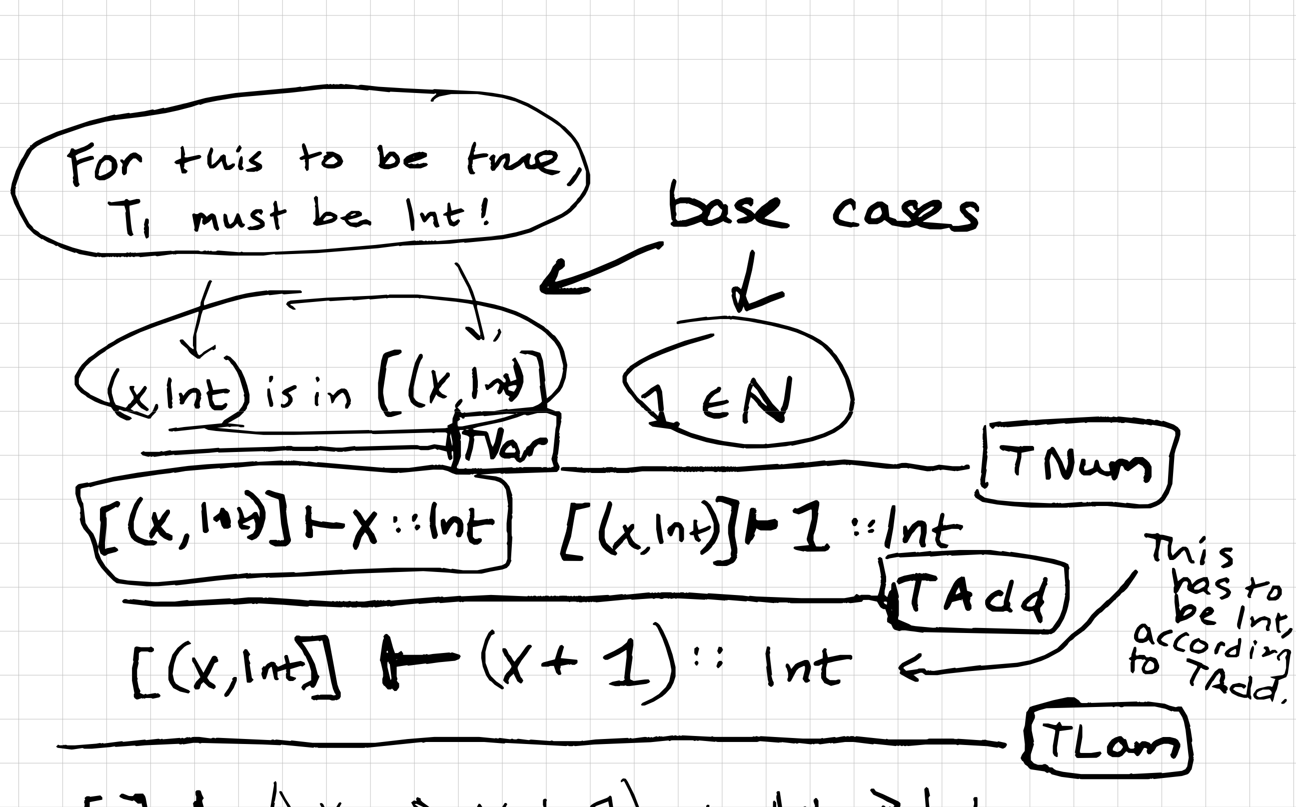
examples: \x → x + 1 :: Int → Int

(x + 1: how do we know what the type of an expression containing a variable should be?)

Four rules so far:

TNum, TAdd, TVar, TLam.

We can now construct a type derivation for programs like \x → x + 1:



What we just did together was mechanically apply the four typing rules to determine the type of an expression. This is what we want to implement!

We have a couple typing rules to go, for dealing with applications, and let-expressions.

$$\frac{\Gamma \vdash e_1 :: T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 :: T_1}{\Gamma \vdash e_1\ e_2 :: T_2} \text{ TApp}$$

examples: (\x → x + 1) 2 :: Int (with (\x → x + 1) :: Int → Int and 2 :: Int)

$$\text{modus ponens}: \frac{A \rightarrow B \quad A}{B}$$

Notice the analogy to the TApp rule!

Curry-Howard correspondence

$$\frac{\Gamma \vdash e_1 :: T_1 \quad \Gamma, (x, T_1) \vdash e_2 :: T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 :: T_2} \text{ TLet}$$

($\Gamma, (x, T_1)$: extended type environment that knows about the type of x.)

examples: let x = 5 in 1 + x :: Int

let x = 5 in \y → y + x :: Int → Int (5: Int)

We could construct a typing derivation for the above programs, if we wanted.

We now have 6 typing rules:

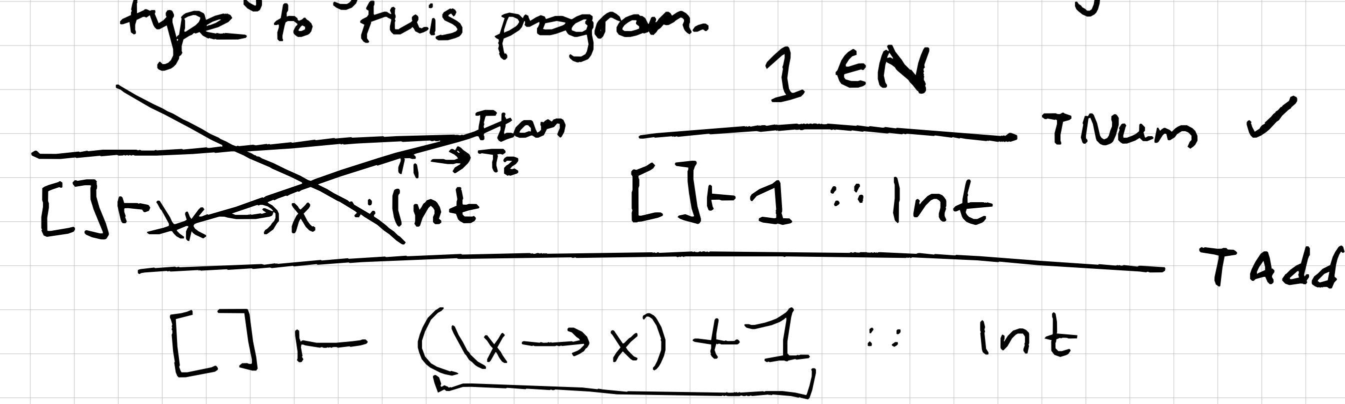
TInt, TAdd, TVar, TLam, TApp, TLet.

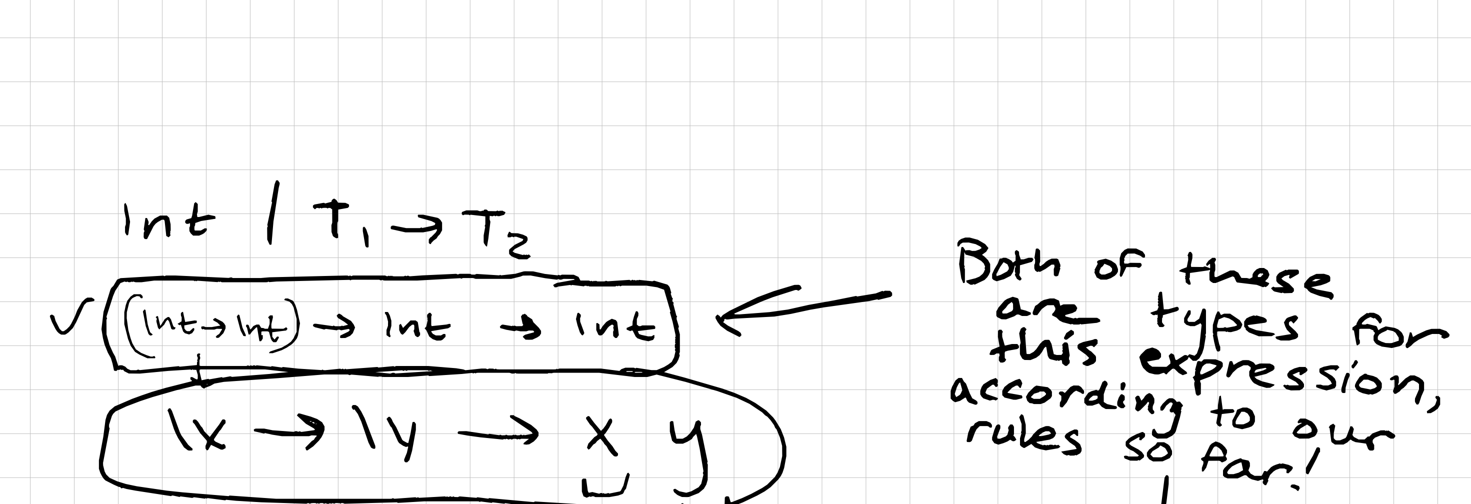
Quiz: why is (\x → x) + 1 ill-typed?

Because you can't add functions and numbers.

More formally,

We cannot use the rules to construct a typing derivation that would give a type to this program.





→ This is actually an expression with a polymorphic type.

\x → x :: a → a (a: type variable)

Int → Int

(Int → Int) → (Int → Int)

Bool → Bool