Polymorphism and Type Inference

Owen Arden
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

Roadmap

Past two weeks:
How do we implement a tiny functional language?
1. Interpreter: how do we evaluate a program given its AST?
2. Parser: how do we convert strings to ASTs?

This week: adding types
How do we check statically if our programs “make sense”?
1. Type system: formalizing the intuition about which expressions have which types
2. Type inference: computing the type of an expression

Reminder: Nano2

```
e ::= n | x       -- numbers, vars
    | e1 + e2    -- arithmetic
    | \x -> e    -- abstraction
    | e1 e2      -- application
    | let x = e1 in e2 -- Let binding
```
**Reminder: Nano2**

Which one of these Nano2 programs is well-typed? *

- (A) \( (\lambda x \cdot x) + 1 \)
- (B) \( 1 \ 2 \)
- (C) \( \text{let } f = \lambda x \cdot x + 1 \text{ in } f \ (\lambda y \cdot y) \)
- (D) \( \lambda x \rightarrow \lambda y \rightarrow x \ y \)
- (E) \( \lambda x \rightarrow x \ x \)

http://tiny.cc/cmps112-nanotype-ind

**Reminder: Nano2**

Which one of these Nano2 programs is well-typed? *

- (A) \( (\lambda x \cdot x) + 1 \)
- (B) \( 1 \ 2 \)
- (C) \( \text{let } f = \lambda x \cdot x + 1 \text{ in } f \ (\lambda y \cdot y) \)
- (D) \( \lambda x \rightarrow \lambda y \rightarrow x \ y \)
- (E) \( \lambda y \rightarrow \lambda y \rightarrow 1 + 2 \)
- (F) \( \lambda x \rightarrow x \ x \)

http://tiny.cc/cmps112-nanotype-grp

**QUIZ**

Answer: D.

A adds a function;  
B applies a number;  
C defines \( f \) to take an \( \text{Int} \) and then passes in a function;  
E requires a type \( T \) that is equal to \( T \rightarrow T \), which doesn’t exit.
Type system for Nano2

A type system defines what types an expression can have.

To define a type system we need to define:

- the syntax of types: what do types look like?
- the static semantics of our language (i.e. the typing rules): assign types to expressions

Type system: take 1

Syntax of types:

\[ T ::= \text{Int} \quad \text{-- integers} \]
\[ \mid T1 \rightarrow T2 \quad \text{-- function types} \]

Now we want to define a typing relation \( e :: T \) (e has type T).

We define this relation inductively through a set of typing rules:

\[ [T\text{-Num}] \quad n :: \text{Int} \]
\[ [T\text{-Add}] \quad \text{--+ premises} \]
\[ \quad e1 :: \text{Int} \quad e2 :: \text{Int} \]
\[ \quad \text{--+ conclusion} \]
\[ \quad e1 + e2 :: \text{Int} \]
\[ [T\text{-Var}] \quad x :: ???? \]

What is the type of a variable?

We have to remember what type of expression it was bound to!

Type Environment

An expression has a type in a given type environment (also called context), which maps all its free variables to their types.

\[ G = x1:T1, \ x2:T2, \ldots, \ xn:Tn \]

Our typing relation should include the context G:

\[ G |- e :: T \quad \text{(e has type T in context G)} \]
Typing rules: take 2

\[
\begin{align*}
&T-\text{Num} \quad \Gamma \vdash n :: \text{Int} \\
&T-\text{Add} \quad \Gamma \vdash e_1 :: \text{Int} \quad \Gamma \vdash e_2 :: \text{Int} \\
&T-\text{Var} \quad \Gamma \vdash x :: T & \text{ if } x : T \text{ in } \Gamma \\
&T-\text{Abs} \quad \Gamma \vdash e :: T_1 \Rightarrow T_2 \\
&T-\text{App} \quad \Gamma \vdash e_1 :: T_1 \Rightarrow T_2 \quad \Gamma \vdash e_2 :: T_2 \\
&T-\text{Let} \quad \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 :: T_2
\end{align*}
\]

Typing rules

\[
\begin{align*}
\Gamma \vdash e :: T
\end{align*}
\]

An expression \( e \) has type \( T \) in \( \Gamma \) if we can derive \( \Gamma \vdash e :: T \) using these rules.

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

- and ill-typed otherwise.

Examples

Example 1:

Let's derive: \( \Gamma \vdash (\lambda x \to x) \ 2 :: \text{Int} \)

\[
\begin{align*}
&T-\text{Var} \quad \Gamma \vdash x :: \text{Int} \\
&T-\text{Abs} \quad \Gamma \vdash \lambda x \to x :: \text{Int} \to \text{Int} \quad \Gamma \vdash 2 :: \text{Int} \\
&T-\text{App} \quad \Gamma \vdash (\lambda x \to x) \ 2 :: \text{Int}
\end{align*}
\]

But we cannot derive: \( \Gamma \vdash 1 \ 2 :: T \) for any type \( T \)

- Why?
- \( T-\text{App} \) only applies when LHS has a function type, but there's no rule to derive a function type for \( 1 \).
Examples

Example 2:

Let's derive: [] |- \ let x = 1 in x + 2 :: Int

[T-Var]----------------- [T-Nun]
x:Int |- x :: Int x:Int |- 2 :: Int
[T-Nun]----------------- [T-Add]
() |- 1 :: Int x:Int |- x + 2 :: Int
[T-Let]-----------------
() |- let x = 1 in x + 2 :: Int

But we cannot derive: [] |- \ let x = \ y -> y in x + 2 :: T for any type T

The [T-Var] rule above will fail to derive x :: Int

Examples

Example 3:

We cannot derive: [] |- (\x :> x x) :: T for any type T

We cannot find any type T to fill in for x, because it has to be equal to T :> T

A note about typing rules

According to these rules, an expression can have zero, one, or many types

- examples?
  1 2 has no types; 3 has one type (Int)

\x :> x has many types:

- we can derive [] |- \x :> x :: Int :> Int
- or [] |- \x :> x :: (Int :> Int) :> (Int :> Int)
- or T :> T for any concrete T

We would like every well-typed expression to have a single most general type!

- most general type = allows most uses
- infer type once and reuse later
QUIZ

Is this program well-typed according to your intuition and according to our rules? *

\[
\text{let } id = \lambda x \to x \text{ in }
ex
\text{let } y = id 5 \text{ in }
\text{id } (\lambda z \to z + y)
\]

- (A) Me: okay, rules: okay
- (B) Me: okay, rules: nope
- (C) Me: nope, rules: okay
- (D) Me: nope, rules: nope

http://tiny.cc/cmps112-typed-ind

QUIZ

Is this program well-typed according to your intuition and according to our rules? *

\[
\text{let } id = \lambda x \to x \text{ in }
ex
\text{let } y = id 5 \text{ in }
\text{id } (\lambda z \to z + y)
\]

- (A) Me: okay, rules: okay
- (B) Me: okay, rules: nope
- (C) Me: nope, rules: okay
- (D) Me: nope, rules: nope

http://tinyc.cc/cmps112-typed-grp

QUIZ

Answer: B.
Double identity

\[
\begin{align*}
\text{let } \text{id} &= \lambda x \to x \text{ in} \\
\text{let } y &= \text{id} \ 5 \text{ in} \\
\text{id} (\lambda z \to z + y)
\end{align*}
\]

Intuitively this program looks okay, but our type system rejects it:

- in the first application, \( \text{id} \) needs to have type \( \text{Int} \to \text{Int} \)
- in the second application, \( \text{id} \) needs to have type \( (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) \)
- the type system forces us to pick just one type for each variable, such as \( \text{id} : \)

What can we do?

Polymorphic types

Intuitively, we can describe the type of \( \text{id} \) like this:

- it’s a function type where
- the argument type can be any type \( T \)
- the return type is then also \( T \)

Polymorphic types

We formalize this intuition as a polymorphic type: \( \forall a \ . \ a \to a \)
- where \( a \) is a (bound) type variable
- also called a type scheme
- Haskell also has polymorphic types, but you don’t usually write \( \forall a \) .

We can instantiate this scheme into different types by replacing \( a \) in the body with some type, e.g.
- instantiating with \( \text{Int} \) yields \( \text{Int} \to \text{Int} \)
- instantiating with \( \text{Int} \to \text{Int} \) yields \( (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \)
- etc.
Inference with polymorphic types

With polymorphic types, we can derive \( e : \text{Int} \rightarrow \text{Int} \) where \( e \) is

\[
\begin{align*}
\text{let} & \quad \text{id} = \lambda x \rightarrow x \text{ in} \\
\text{let} & \quad y = \text{id} 5 \text{ in} \\
\text{id} & \quad (\lambda z \rightarrow z + y)
\end{align*}
\]

At a high level, inference works as follows:

1. When we have to pick a type \( T \) for \( x \), we pick a fresh type variable \( a \)
2. So the type of \( \lambda x \rightarrow x \) comes out as \( a \rightarrow a \)
3. We can generalize this type to \( \forall a . \ a \rightarrow a \)
4. When we apply \text{id} the first time, we instantiate this polymorphic type with \text{Int}
5. When we apply \text{id} the second time, we instantiate this polymorphic type with \( \text{Int} \rightarrow \text{Int} \)

Let's formalize this intuition as a type system!

Type system: take 3

Syntax of types

-- Mono-types
\[ T ::= \text{Int} \quad \text{-- integers} \]
\[ \text{T1} \rightarrow \text{T2} \quad \text{-- function types} \]
\[ a \quad \text{-- NEW: type variable} \]

-- NEW: Poly-types (type schemes)
\[ S ::= T \quad \text{-- mono-type} \]
\[ \forall a . S \quad \text{-- polymorphic type} \]

Type Environment

The type environment now maps variables to poly-types: \( G : \text{Var} \rightarrow \text{Poly} \)

- example, \( G = [z: \text{Int}, \text{id}: \forall a . a \rightarrow a] \)

Type Substitutions

We need a mechanism for replacing all type variables in a type with another type

A type substitution is a finite map from type variables to types: \( U : \text{TVar} \rightarrow \text{Type} \)

- example: \( U_1 = [a / \text{Int}, b / (c \rightarrow c)] \)

To apply a substitution \( U \) to a type \( T \) means replace all type vars in \( T \) with whatever they are mapped to in \( U \)

- example 1: \( U_1 (a \rightarrow a) = \text{Int} \rightarrow \text{Int} \)
- example 2: \( U_1 \text{Int} = \text{Int} \)
QUIZ

What is the result of the following substitution application? *

\[ [a / \text{Int}, b / c \rightarrow c] (b \rightarrow d \rightarrow b) \]

- (A) \( c \rightarrow d \rightarrow c \)
- (B) \( (c \rightarrow c) \rightarrow d \rightarrow (c \rightarrow c) \)
- (C) Error: no mapping for type variable d
- (D) Error: type variable a is unused

http://tiny.cc/cmps112-subst-ind

---

QUIZ

What is the result of the following substitution application? *

\[ [a / \text{Int}, b / c \rightarrow c] (b \rightarrow d \rightarrow b) \]

- (A) \( c \rightarrow d \rightarrow c \)
- (B) \( (c \rightarrow c) \rightarrow d \rightarrow (c \rightarrow c) \)
- (C) Error: no mapping for type variable d
- (D) Error: type variable a is unused

http://tiny.cc/cmps112-subst-grp

---

QUIZ

(B) \( (c \rightarrow c) \rightarrow d \rightarrow (c \rightarrow c) \)

Answer: B

---
Typing rules

We need to change the typing rules so that:

1. Variables (and their definitions) can have polymorphic types

\[
\begin{align*}
[T-\text{Var}] & \quad G \mid \ x : S \quad \text{if } x:S \text{ in } G \\
G \mid & \ e1 : S \quad G, x:S \mid \ e2 : T \\
[T-\text{Let}] & \quad \text{------------------------------------} \\
G \mid & \ \text{let } x = e1 \text{ in } e2 : T \\
\end{align*}
\]

Typing rules

2. We can instantiate a type scheme into a type

\[
\begin{align*}
G \mid & \ e : \text{forall } a . S \\
[T-\text{Inst}] & \quad \text{------------------------} \\
G \mid & \ e : [a / T] S \\
\end{align*}
\]

3. We can generalize a type with free type variables into a type scheme

\[
\begin{align*}
G \mid & \ e : S \\
[T-\text{Gen}] & \quad \text{------------------------} \quad \text{if not } (a \text{ in FTV}(G)) \\
G \mid & \ e : \text{forall } a . S \\
\end{align*}
\]

Typing rules

The rest of the rules are the same:

\[
\begin{align*}
[T-\text{Num}] & \quad G \mid n : \text{Int} \\
G \mid & \ e1 : \text{Int} \quad G \mid e2 : \text{Int} \\
[T-\text{Add}] & \quad \text{-----------------------------------} \\
G \mid & \ e1 + e2 : \text{Int} \\
G, x:T1 \mid & \ e : T2 \\
[T-\text{Abs}] & \quad \text{-----------------------------------} \\
G \mid & \ \backslash x \rightarrow e : T1 \rightarrow T2 \\
G \mid & \ e1 : T1 \rightarrow T2 \quad G \mid \ e2 : T1 \\
[T-\text{App}] & \quad \text{-----------------------------------} \\
G \mid & \ e1 e2 : T2
\end{align*}
\]
Examples

Example 1
Let's derive: \( \lambda x : a. a \to a \)

\[
\begin{align*}
[T\text{-Var}] & \quad \vdash x : a \\
[T\text{-Abs}] & \quad \vdash \lambda x : a. x \to a \\
[T\text{-Gen}] & \quad \vdash \lambda x : a. x \to a \quad \text{not (a in FTV(\[]))}
\end{align*}
\]

Can we derive: \( \lambda x : a. x \to a \)?

No! The side condition of \( T\text{-Gen} \) is violated because \( a \) is present in the context.

Examples

Example 2
Let's derive: \( G1 \vdash id 5 : \text{Int} \) where \( G1 = [id : (\forall a. a \to a)] \):

\[
\begin{align*}
[T\text{-Var}] & \quad \vdash id : \forall a. a \to a \\
[T\text{-Inst}] & \quad \vdash id : \text{Int} \to \text{Int} \quad \vdash 5 : \text{Int} \\
[T\text{-App}] & \quad \vdash id 5 : \text{Int}
\end{align*}
\]

Examples

Example 3
Finally, we can derive:

\[
\begin{align*}
& (\text{let id} = \lambda x : a. x \in) \\
& \quad (\text{let y} = \text{id} 5 \in) \\
& \quad \quad (\text{id} (\lambda z : a. z + y)) : \text{Int} \to \text{Int}
\end{align*}
\]
**Examples**

- **easy**
  
  **[T-Var]**
  
  \[ G_2 \models \text{id} :: \forall a \ . \ a \to a \]
  
  \[ G_3 \models z + y :: \text{Int} \]
  
- **Add**
  
  \[ G_2 \models \text{id} :: \forall \text{Int} \to \text{Int} \to \text{Int} \]
  
  \[ G_2 \models \text{_z} :: \text{Int} \to \text{Int} \]
  
- **example 2**
  
  \[ G_1 \models \text{id} \cdot 5 :: \text{Int} \]
  
  \[ G_2 \models \text{id} \cdot (\text{z} \to \text{z+y}) :: \text{Int} \to \text{Int} \]
  
- **[T-Abs]**
  
  \[ G_1 \models \text{let} \cdot y = \text{id} \cdot 5 \text{ in } \ldots :: \text{Int} \to \text{Int} \]
  
- **[T-Let]**
  
  \[ [] \models \text{let} \cdot \text{id} = \lambda x \to x \text{ in } \ldots :: \text{Int} \to \text{Int} \]

- **example 1**
  
  \[ G_1 = [\text{id} : (\forall a \ . a \to a)] \]
  
  \[ G_2 = [\text{y} : \text{Int}, \text{id} : (\forall a \ . a \to a)] \]
  
  \[ G_3 = [\text{z} : \text{Int}, \text{y} : \text{Int}, \text{id} : (\forall a \ . a \to a)] \]

---

**Type inference algorithm**

Our ultimate goal is to implement a Haskell function `infer` which

- given a context \( G \) and an expression \( e \)
- returns a type \( T \) such that \( G \models e :: T \)
- or reports a type error if \( e \) is ill-typed in \( G \)

---

**Representing types**

First, let’s define a Haskell datatype to represent Nano2 types:

```haskell
data Type
  = TInt
  | Type => Type
  | TVar String

data Poly = Mono Type
  | Forall TVar Poly

type TVar = String

type TEnv = [(Id, Poly)]
  -- type environment

type Subst = [(String, Type)]
  -- type substitution
```
Inference: main idea

Let's implement infer like this:

1. Depending on what kind of expression e is, find a typing rule that applies to it
2. If the rule has premises, recursively call infer to obtain the types of sub-expressions
3. Combine the types of sub-expression according to the conclusion of the rule
4. If no rule applies, report a type error

---

Inference: main idea

let infer :: TypeEnv -> Expr -> Type
infer _ (ENum _) = TInt
infer tEnv (EVar var) = lookup var tEnv
infer tEnv (EAdd e1 e2) =
  if t1 == TInt && t2 == TInt
  then return TInt
  else throw "type error: + expects Int operands"
where
  t1 = infer tEnv e1
  t2 = infer tEnv e2
...  

This doesn't quite work (for other cases). Why?

Inference: tricky bits

The trouble is that our typing rules are nondeterministic!
- When building derivations, sometimes we had to guess how to proceed

Problem 1: Guessing a type

-- oh, now we know!

[T-Var]

[x:?] |- x :: Int   [x:?] |- 1 :: Int

[T-Add]

[x:?] |- x + 1 :: ?? -- what should "?" be?

[T-Abs]

[] |- (\x -> x + 1) :: ? -> ??
Inference: tricky bits

Problem 1: Guessing a type
So, if we want to implement
\[
\text{infer } t\text{Env } (\text{ELam } x \text{ e}) = tX \Rightarrow tBody
\]
where
\[
\begin{align*}
t\text{Env'} &= \text{extendTEnv } x \times t\text{Env} \\
tX &= ??? -- \text{what do we put here?} \\
tBody &= \text{infer } t\text{Env'} e
\end{align*}
\]
...

Problem 2: Guessing when to generalize
In the derivation for
\[
\begin{align*}
(\text{let } id &= \lambda x \rightarrow x \text{ in} \\
\text{let } y &= id \text{ in} \\
id (\lambda z \rightarrow z + y) :& \text{ Int } \rightarrow \text{ Int}
\end{align*}
\]
we had to guess that the type of id should be generalized into
\[
\forall a . a \rightarrow a
\]
Let’s deal with problem 1 first

Constraint-based type inference
-- oh, now we know!

\[
\begin{align*}
[T-\text{Var}] &- \text{----------------} \\
[x?:?] &- x : \text{ Int} \quad [x:] &- 1 : \text{ Int}
\end{align*}
\]

\[
\begin{align*}
[T-\text{Add}] &- \text{----------------} \\
[x?:?] &- x + 1 : ? ? -- \text{ what should } ? \text{ be?}
\end{align*}
\]

\[
\begin{align*}
[T-\text{Abs}] &- \text{----------------} \\
[] &- (\lambda x \rightarrow x + 1) : ? \rightarrow ? ?
\end{align*}
\]

Main idea:
1. Whenever you need to “guess” a type, don’t:
   - just return a fresh type variable
   - fresh = not used anywhere else in the program
2. Whenever a rule imposes a constraint on a type (i.e. says it should have certain form):
   - try to find the right substitution for the free type vars to satisfy the constraint
   - this step is called unification
Example

Let's infer the type of \( \lambda x \to x + 1 \):

<table>
<thead>
<tr>
<th>Step</th>
<th>Expression</th>
<th>Subst</th>
<th>Inferred type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \lambda x \to x + 1 )</td>
<td>[ ]</td>
<td>[ T-Abs ] [ ]</td>
</tr>
<tr>
<td>2</td>
<td>([x:a0]) ( x + 1 )</td>
<td>([ T-Add ] ) ( a0 )</td>
<td>( a0 )</td>
</tr>
<tr>
<td>3</td>
<td>( x )</td>
<td>([ T-Var ] )</td>
<td>( a0 )</td>
</tr>
<tr>
<td>4</td>
<td>( x + 1 )</td>
<td>unify ( a0 ) ( \text{Int} )</td>
<td>( a0 / \text{Int} )</td>
</tr>
<tr>
<td>5</td>
<td>([x:\text{Int}]) ( 1 )</td>
<td>([ T-Num ] )</td>
<td>( \text{Int} )</td>
</tr>
<tr>
<td>6</td>
<td>( x + 1 )</td>
<td>unify ( \text{Int} ) ( \text{Int} )</td>
<td>( \text{Int} )</td>
</tr>
<tr>
<td>7</td>
<td>( x + 1 )</td>
<td>( \text{Int} )</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>( \lambda x \to x + 1 )</td>
<td>( \text{Int} \to \text{Int} )</td>
<td></td>
</tr>
</tbody>
</table>

Unification

The unification problem: given two types \( T_1 \) and \( T_2 \), find a type substitution \( U \) such that \( U \ T_1 = U \ T_2 \).

Such a substitution is called a unifier of \( T_1 \) and \( T_2 \)

Examples:

\begin{align*}
\text{The unifier of:} \\
a & \quad \text{and} \quad \text{Int} & \quad \text{is} & \quad [a / \text{Int}] \\
a \to a & \quad \text{and} \quad \text{Int} \to \text{Int} & \quad \text{is} & \quad [a / \text{Int}] \\
a \to \text{Int} & \quad \text{and} \quad \text{Int} \to b & \quad \text{is} & \quad [a / \text{Int}, b / \text{Int}] \\
\text{Int} & \quad \text{and} \quad \text{Int} & \quad \text{is} & \quad [\] \\
a & \quad \text{and} \quad a & \quad \text{is} & \quad [\] \\
\text{Int} & \quad \text{and} \quad \text{Int} \to \text{Int} & \quad \text{cannot unify!} \\
\text{Int} & \quad \text{and} \quad a \to a & \quad \text{cannot unify!} \\
a & \quad \text{and} \quad a \to a & \quad \text{cannot unify!}
\end{align*}
QUIZ

What is the unifier of the following two types? *

1. a -> Int -> Int
2. b -> c

- (A) Cannot unify
- (B) [a / Int, b / Int -> Int, c / Int]
- (C) [a / Int, b / Int, c / Int -> Int]
- (D) [b / a, c / Int -> Int]
- (E) [a / b, c / Int -> Int]

http://tiny.cc/cmps112-unify-ind

QUIZ

What is the unifier of the following two types? *

1. a -> Int -> Int
2. b -> c

- (A) Cannot unify
- (B) [a / Int, b / Int -> Int, c / Int]
- (C) [a / Int, b / Int, c / Int -> Int]
- (D) [b / a, c / Int -> Int]
- (E) [a / b, c / Int -> Int]

http://tiny.cc/cmps112-unify-grp

QUIZ

(C), (D) and (E) are all unifiers!

But somehow (D) and (E) are better than (C)

- they make the least commitment required to make these types equal
- this is called the most general unifier
Infer: take 2

Let's add constraint-based typing to infer!

-- Now has to keep track of current substitution!

infer :: Subst -> TypeEnv -> Expr -> (Subst, Type)
infer sub (ENum _) = (sub, TInt)
infer sub tEnv (EVar var) = (sub, lookup var tEnv)

-- Lambda case: simply generate fresh type variable!
infer sub tEnv (ELam x e) = (sub1, tX' :=> tBody)

where
    tEnv' = extendTEnv x tX tEnv
tX = freshTV -- we'll get to this
(sub1, tBody) = infer sub tEnv' e
tX' = apply sub1 tX

-- Add case: recursively infer types of operands
-- and enforce constraint that they are both Int

infer sub tEnv (EAdd e1 e2) = (sub4, TInt)

where
    (sub1, t1) = infer sub tEnv e1 -- 1. infer type of e1
    sub2 = unify sub1 t1 Int -- 2. constraint: t1 is Int
    tEnv' = apply sub2 tEnv -- 3. apply subst to context
    (sub3, t2) = infer sub2 tEnv' e2 -- 4. infer e2 type in new ctx
    sub4 = unify sub3 t2 Int -- 5. constraint: t2 is Int

Why are all these steps necessary? Can't we just return (sub, TInt)?

QUIZ

Which of these programs will type-check if we skip step 3? *

infr sub tEnv (EAdd e1 e2) = (sub4, TInt)

where
    (sub1, t1) = infer sub tEnv e1 -- 1. infer type of e1
    sub2 = unify sub1 t1 Int -- 2. enforce constraint: t1 is Int
    tEnv' = apply sub2 tEnv -- 3. apply substitution to context
    (sub3, t2) = infer sub2 tEnv' e2 -- 4. infer e2 type in new ctx
    sub4 = unify sub3 t2 Int -- 5. enforce constraint: t2 is Int

(A) 1 + 3
(B) 1 + 2 3
(C) (x -> x) + 1
(D) 1 + (x -> x)
(E) \( \lambda x \rightarrow x \times x \times 5 \)

http://tiny.cc/cmps112-infer-ind
QUIZ

Which of these programs will type-check if we skip step 3? *

inference sub tEnv (EAP 1 e2) = (sub4, tBody)

where

\( (\text{sub1}, t1') = \text{inference sub tEnv e1} \quad \text{-- 1. infer type of e1} \)

\( \text{sub1} = \text{unify sub} t1 \text{ Int} \quad \text{-- 2. enforce constraint: t1 is Int} \)

\( t1' = \text{apply sub2 tEnv} \quad \text{-- 3. apply substitution to context} \)

\( (\text{sub2}, t2') = \text{inference sub tEnv' e2} \quad \text{-- 4. infer type of e2 in new context} \)

\( \text{sub4} = \text{unify sub3 t2 Int} \quad \text{-- 5. enforce constraint: t2 is Int} \)

- (A) 1 2 + 3
- (B) 1 + 2 3
- (C) (x -> x) + 1
- (D) 1 + (x -> x)
- (E) \( \lambda x \rightarrow x + x \times 5 \)

http://tiny.cc/cmps112-infer-qrp

QUIZ

Answer: E.
A fails in step 1 (LHS is ill-typed);
B fails in step 4 (RHS is ill-typed);
C fails in step 2 (LHS is not Int);
D fails in step 5 (RHS is not Int);
finally, E should fail because LHS and RHS by themselves are fine, but not together!

Fresh type variables

Now has to keep track of current substitution!

\[
\begin{align*}
\text{infer} & :: \text{Subst} \rightarrow \text{TypeEnv} \rightarrow \text{Expr} \rightarrow (\text{Subst}, \text{Type}) \\
\text{-- lambda case: simply generate fresh type variable!} \\
\text{infer } \text{tEnv } (E\text{Lam } x \ e) & = \text{tX } :\Rightarrow \text{tBody} \\
\text{where} & \\
\text{tEnv'} & = \text{extendTEnv } x \text{ tX } \text{tEnv} \\
\text{tX} & = \text{freshTV -- how do we do this?} \\
\text{tBody} & = \text{infer } \text{tEnv'} \ e
\end{align*}
\]

Intended behavior:

- First time we call freshTV it returns a0
- Second time it returns a1
- ... and so on

Can we do that in Haskell?

No, Haskell is pure. Have to thread the counter through ;(
Polymorphism: the final frontier

Back to double identity:

```haskell
let id = \x -> x in -- Must generalize the type of id
let y = id 5 in -- Instantiate with Int
id (\z -> z + y) -- Instantiate with (Int -> Int)
```

- When should we to generalize a type like `a -> a` into a polymorphic type like `forall a. a -> a`?
- When should we instantiate a polymorphic type like `forall a. a -> a` and with what?

### Polymorphism: the final frontier

Generalization and instantiation:

- Whenever we infer a type for a let-defined variable, generalize it!
  - It’s safe to do so, even when not strictly necessary
- Whenever we see a variable with a polymorphic type, instantiate it
  - With what type?
  - Well, what do we use when we don’t know what type to use?
  - Fresh type variables!

#### Example

Let’s infer the type of `let id = \x -> x in 5`:

<table>
<thead>
<tr>
<th>TEnv</th>
<th>Expression</th>
<th>Step</th>
<th>Subst</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>let id=\x-&gt;x in 5</td>
<td>[T-Let]</td>
<td>[]</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>\x-&gt;x</td>
<td>[T-Abs]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[x:a0]</td>
<td>x</td>
<td>[T-Var]</td>
<td></td>
<td>a0</td>
</tr>
<tr>
<td>[]</td>
<td>\x-&gt;x</td>
<td>[T-Var]</td>
<td></td>
<td>a0 -&gt; a0</td>
</tr>
<tr>
<td>[]</td>
<td>let id=\x-&gt;x in 5</td>
<td>generalize a0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tEnv</td>
<td>id</td>
<td>[T-Var]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>id</td>
<td>instantiate</td>
<td></td>
<td>a1 -&gt; a1</td>
</tr>
<tr>
<td>5</td>
<td>[T-Num]</td>
<td>Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>id</td>
<td>unify (a1-&gt;a1)</td>
<td>(Int-&gt;a2)</td>
<td>[a1/Int,a2/Int]</td>
</tr>
<tr>
<td>[]</td>
<td>let id=\x-&gt;x in 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>id</td>
<td></td>
<td></td>
<td>Int</td>
</tr>
</tbody>
</table>

Here `tEnv = [id : forall a0.a0->a0]`
## What we learned this week

Type system: a set of rules about which expressions have which types

Type environment (or context): a mapping of variables to their types

Polymorphic type: a type parameterized with type variables that can be instantiated with any concrete type

Type substitution: a mapping of type variables to types; you can apply a substitution to a type by replacing all its variables with their values in the substitution

Unifier of two types: a substitution that makes them equal; unification is the process of finding a unifier

---

## What we learned this week

Type inference: an algorithm to determine the type of an expression

Constraint-based type inference: a type inference technique that uses fresh type variables and unification

Generalization: turning a mono-type with free type variables into a polymorphic type (by binding its variables with a `forall`)

Instantiation: turning a polymorphic type into a mono-type by substituting type variables in its body with some types