

CSE 114

# Introduction to Functional Programming

*Polymorphism and Type Inference*

# Roadmap

---

## Last week:

How do we *implement* a tiny functional language?

1. *Interpreter*: how do we *evaluate* a program given its AST?

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



<http://tiny.cc/cse116-nanotype-ind>

# Reminder: Nano2

---

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

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



<http://tiny.cc/cse116-nanotype-grp>

# QUIZ

---

*Answer: D.*

A adds a function;

B applies a number;

C defines  $f$  to take an `Int` and then passes in a function;

E requires a type `T` that is equal to `T -> T`, which doesn't exist.

# 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 ::= Int      -- integers
    | T1 -> T2 -- 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-Num]  n :: Int
```

```
[T-Add]  e1 :: Int    e2 :: Int    -- premises
         -----
         e1 + e2 :: Int    -- conclusion
```

```
[T-Var]  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*

$G = x_1:T_1, x_2:T_2, \dots, x_n:T_n$

Our *typing relation* should include the context  $G$ :

$G \vdash e :: T$  (e has type  $T$  in context  $G$ )

# Typing rules: take 2

---

[T-Num]  $G \vdash n :: \text{Int}$

[T-Add] 
$$\frac{G \vdash e_1 :: \text{Int} \quad G \vdash e_2 :: \text{Int}}{G \vdash e_1 + e_2 :: \text{Int}}$$

[T-Var]  $G \vdash x :: T$       **if**  $x:T$  **in**  $G$

[T-Abs] 
$$\frac{G, x:T_1 \vdash e :: T_2}{G \vdash \lambda x. e :: T_1 \rightarrow T_2}$$

[T-App] 
$$\frac{G \vdash e_1 :: T_1 \rightarrow T_2 \quad G \vdash e_2 :: T_1}{G \vdash e_1 e_2 :: T_2}$$

[T-Let] 
$$\frac{G \vdash e_1 :: T_1 \quad G, x:T_1 \vdash e_2 :: T_2}{G \vdash \text{let } x = e_1 \text{ in } e_2 :: T_2}$$

# Typing rules

---

$G \mid - e :: T$

An expression  $e$  **has type**  $T$  in  $G$  if we can derive  $G \mid - e :: T$  using these rules

An expression  $e$  is **well-typed** in  $G$  if we can derive  $G \mid - e :: T$  for some type  $T$

- and **ill-typed** otherwise

# Examples

---

Example 1:

Let's derive:  $[\ ] \vdash (\lambda x \rightarrow x) \ 2 \ :: \text{Int}$

[T-Var] -----

$[x:\text{Int}] \vdash x \ :: \text{Int}$

[T-Abs] -----

$[\ ] \vdash \lambda x \rightarrow x \ :: \text{Int} \rightarrow \text{Int}$

----- [T-Num]

$[\ ] \vdash 2 \ :: \text{Int}$

[T-App] -----

$[\ ] \vdash (\lambda x \rightarrow x) \ 2 \ :: \text{Int}$

But we *cannot* derive:  $[\ ] \vdash 1 \ 2 \ :: T$  for any type  $T$

- Why?
- **T-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:  $[\ ] \mid - \text{let } x = 1 \text{ in } x + 2 :: \text{Int}$

$$\begin{array}{c} \text{[T-Var]} \text{-----} \qquad \qquad \qquad \text{-----} \text{[T-Num]} \\ \qquad \qquad \qquad x:\text{Int} \mid - x :: \text{Int} \qquad x:\text{Int} \mid - 2 :: \text{Int} \\ \text{[T-Num]} \text{-----} \qquad \qquad \qquad \text{-----} \text{[T-Add]} \\ \qquad \qquad \qquad [\ ] \mid - 1 :: \text{Int} \qquad x:\text{Int} \mid - x + 2 :: \text{Int} \\ \text{[T-Let]} \text{-----} \\ \qquad \qquad \qquad [\ ] \mid - \text{let } x = 1 \text{ in } x + 2 :: \text{Int} \end{array}$$

But we *cannot* derive:  $[\ ] \mid - \text{let } x = \backslash y \rightarrow y \text{ in } x + 2 :: T$  for any type  $T$

The  $\text{[T-Var]}$  rule above will fail to derive  $x :: \text{Int}$

# Examples

---

## Example 3:

We cannot derive:  $[\ ] \vdash (\lambda x \rightarrow x\ x) :: T$  for any type  $T$

We cannot find any type  $T$  to fill in for  $x$ , because it has to be equal to  $T \rightarrow T$

# A note about typing rules

---

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

- examples?

1 2 has no types; 1 has one type (**Int**)

$\lambda x \rightarrow x$  has many types:

- we can derive  $[] \mid - \lambda x \rightarrow x :: \text{Int} \rightarrow \text{Int}$
- or  $[] \mid - \lambda x \rightarrow x :: (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$
- or  $T \rightarrow T$  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? \*

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> 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/cse116-typed-ind>



# QUIZ

---

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

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> 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/cse116-typed-grp>

# QUIZ

---

*Answer: B.*

# Double identity

---

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

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

- in the first application, `id` needs to have type `Int -> Int`
- in the second application, `id` needs to have type `(Int -> Int) -> (Int -> Int)`
- the type system forces us to pick *just one type* for each variable, such as `id :`

What can we do?

# Polymorphic types

---

Intuitively, we can describe the type of `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 -> 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 `Int` yields `Int -> Int`
- instantiating with `Int -> Int` yields `(Int -> Int) -> Int -> Int`
- etc.

# Inference with polymorphic types

---

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

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

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  $\lambda x . x$  comes out as  $a \rightarrow a$
3. We can generalize this type to  $\forall a . a \rightarrow a$
4. When we apply `id` the first time, we instantiate this polymorphic type with `Int`
5. When we apply `id` the second time, we instantiate this polymorphic type with `Int -> Int`

Let's formalize this intuition as a type system!

# Type system: take 3

---

## Syntax of types

```
-- Mono-types
T ::= Int      -- integers
    | T1 -> T2 -- function types
    | a        -- NEW: type variable

-- NEW: Poly-types (type schemes)
S ::= T        -- mono-type
    | forall a . S -- polymorphic type
```

where  $a \in TVar$ ,  $T \in Type$ ,  $S \in Poly$

## Type Environment

The type environment now maps variables to poly-types:  $G : Var \rightarrow Poly$

- example,  $G = [z: Int, id: forall a . a \rightarrow a]$

# Type system: take 3

---

## 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:  $U : TVar \rightarrow Type$

- example:  $U1 = [a / Int, b / (c \rightarrow c)]$

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:  $U1 (a \rightarrow a) = Int \rightarrow Int$
- example 2:  $U1 Int = Int$



# 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/cse116-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/cse116-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

[T-Var]  $G \vdash x :: S$                       **if**  $x:S$  **in**  $G$

[T-Let] 
$$\frac{G \vdash e1 :: S \quad G, x:S \vdash e2 :: T}{G \vdash \text{let } x = e1 \text{ in } e2 :: T}$$

# Typing rules

---

2. We can *instantiate* a type scheme into a type

$$\begin{array}{c} G \mid - e :: \text{forall } a . S \\ \text{[T-Inst]} \text{ -----} \\ G \mid - e :: [a / T] S \end{array}$$

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

$$\begin{array}{c} G \mid - e :: S \\ \text{[T-Gen]} \text{ ----- if not } (a \text{ in } \text{FTV}(G)) \\ G \mid - e :: \text{forall } a . S \end{array}$$

# Typing rules

---

The rest of the rules are the same:

[T-Num]  $G \vdash n :: \text{Int}$

[T-Add] 
$$\frac{G \vdash e_1 :: \text{Int} \quad G \vdash e_2 :: \text{Int}}{G \vdash e_1 + e_2 :: \text{Int}}$$

[T-Abs] 
$$\frac{G, x:T_1 \vdash e :: T_2}{G \vdash \lambda x. e :: T_1 \rightarrow T_2}$$

[T-App] 
$$\frac{G \vdash e_1 :: T_1 \rightarrow T_2 \quad G \vdash e_2 :: T_1}{G \vdash e_1 e_2 :: T_2}$$

# Examples

---

## Example 1

Let's derive:  $[\ ] \vdash \lambda x \rightarrow x :: \text{forall } a . a \rightarrow a$

[T-Var] -----  
           $[x:a] \vdash x :: a$

[T-Abs] -----  
           $[\ ] \vdash \lambda x \rightarrow x :: a \rightarrow a$

[T-Gen] ----- not (a in FTV([\ ]))  
           $[\ ] \vdash \lambda x \rightarrow x :: \text{forall } a . a \rightarrow a$

Can we derive:  $[x:a] \vdash x :: \text{forall } a . a$ ?

No! The side condition of [T-Gen] is violated because  $a$  is present in the context

# Examples

---

## Example 2

Let's derive:  $G1 \vdash id\ 5 :: Int$  where  $G1 = [id : (forall\ a . a \rightarrow a)]$ :

```
[T-Var]-----  
      G1 ⊢ id :: forall a.a -> a  
[T-Inst]----- [T-Num]-----  
      G1 ⊢ id :: Int -> Int      G1 ⊢ 5 :: Int  
[T-App]-----  
      G1 ⊢ id 5 :: Int
```



# Examples

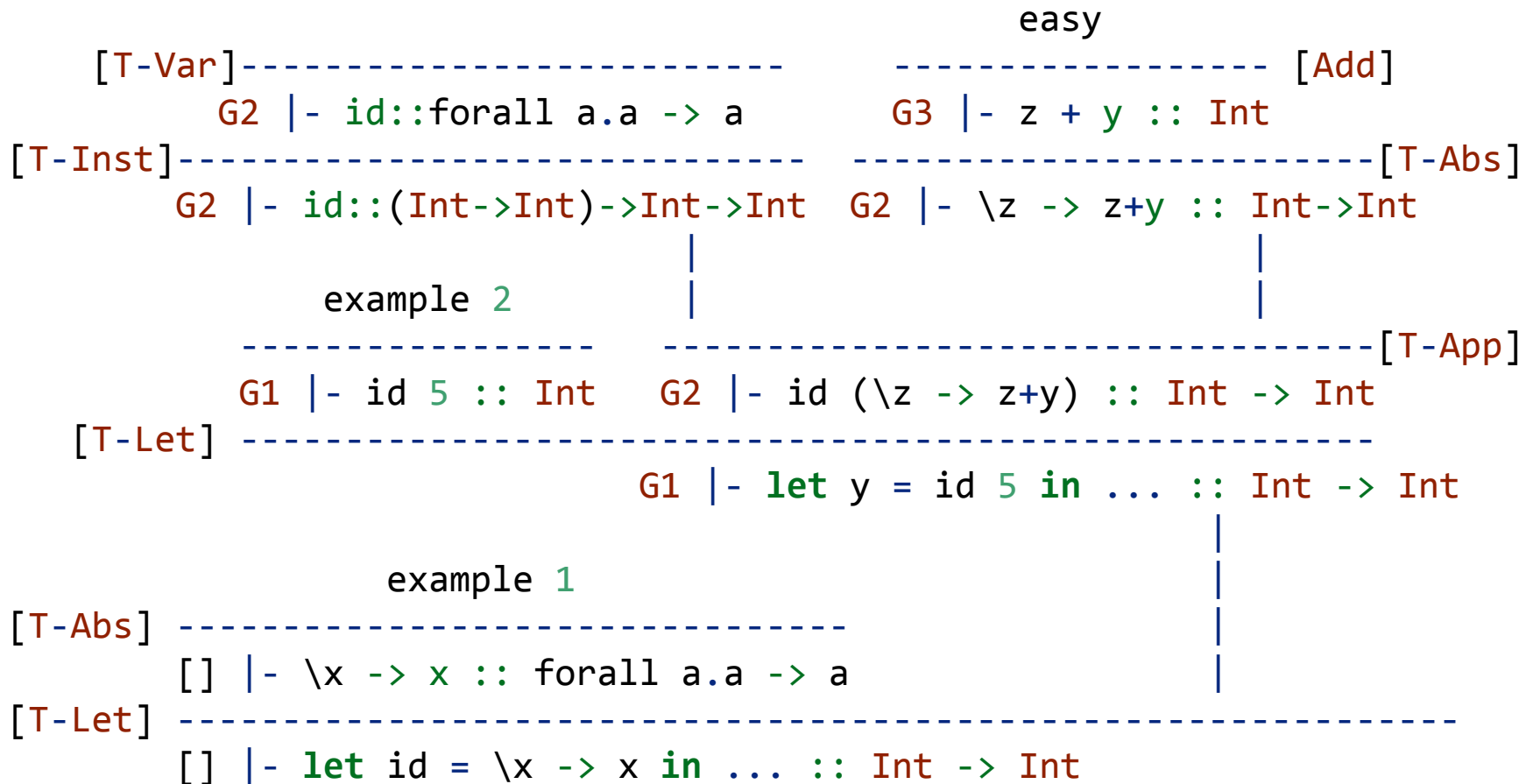
---

## Example 3

Finally, we can derive:

```
(let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)) :: Int -> Int
```

# Examples



- G1 = [id : (forall a . a -> a)]
- G2 = [y : Int, id : (forall a . a -> a)]
- G3 = [z : Int, y : Int, id : (forall a . a -> 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  $G \vdash e :: T$
- *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:

```
type Id    = String           -- program variables, "x"
type Tid  = String           -- type variables, "a"

data MonoType = TInt          -- Int
              | TFun MonoType MonoType -- T1 -> T2
              | TVar Tid        -- "a"

type TEnv  = [(Id, MonoType)] -- type environment
type Subst = [(Tid, MonoType)] -- 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

---

```
-- | This is not the final version!!!
infer :: TypeEnv -> Expr -> MonoType
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

```
infer tEnv (ELam x e) = TFun tX tBody
```

**where**

```
tX      = ??? -- what do we put here?
```

```
tEnv'   = extendTEEnv x tX tEnv
```

```
tBody   = infer tEnv' e
```

...

# Inference: tricky bits

---

## Problem 2: Guessing when to generalize

In the derivation for

```
(let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)) :: Int -> Int
```

we had to *guess* that the type of `id` should be generalized into

```
forall a . a -> a
```

Let's deal with problem 1 first

# Constraint-based type inference

---

*-- 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) :: ? -> ??

## 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  $\lambda x. x + 1$ :

<i>-- TEnv</i>	<i>Expression</i>	<i>Step</i>	<i>Subst</i>	<i>Inferred type</i>
1 []	$\lambda x. x + 1$	[T-Abs]	[]	
2 [x:a0]	$x + 1$	[T-Add]		
3	$x$	[T-Var]		a0
4	$x + 1$	unify a0 Int	[a0/Int]	
5 [x:Int]	$1$	[T-Num]		Int
6	$x + 1$	unify Int Int		
7	$x + 1$			Int
8 []	$\lambda x. x + 1$			Int -> Int

# Example

---

1. Infer the type of  $(\lambda x \rightarrow x + 1)$  in  $[\ ]$  (apply  $[T\text{-Abs}]$ )
2. For the type of  $x$ , pick *fresh type variable* (say,  $a\theta$ ); infer the type of  $x + 1$  in  $[x:a\theta]$  (apply  $[T\text{-Add}]$ )
3. Infer the type of  $x$  in  $[x:a\theta]$  (apply  $[T\text{-Var}]$ ); result:  $a\theta$
4.  $[T\text{-Add}]$  *imposes a constraint*: its LHS must be of type  $\text{Int}$ , so *unify*  $a\theta$  and  $\text{Int}$  and update the *current substitution* to  $[a\theta / \text{Int}]$
5. *Apply* the current substitution  $[a\theta/\text{Int}]$  to the type environment  $[x:a\theta]$  to get  $[x:\text{Int}]$ . Infer the type of  $1$  in  $[x:\text{Int}]$  (apply  $[T\text{-Num}]$ ); result:  $\text{Int}$
6.  $[T\text{-Add}]$  *imposes a constraint*: its RHS must be of type  $\text{Int}$ , so *unify*  $\text{Int}$  and  $\text{Int}$ ; current substitution doesn't change
7. By conclusion of  $[T\text{-Add}]$ : return  $\text{Int}$  as the inferred type
8. By conclusion of  $[T\text{-Lam}]$ : return  $\text{Int} \rightarrow \text{Int}$  as the inferred type

# Unification

---

The unification problem: given types **T1** and **T2**, find a type substitution **U** such that

$$U \ T1 \ = \ U \ T2.$$

Such a substitution is called a *unifier* of **T1** and **T2**

## Examples:

The unifier of:

<code>a</code>	and	<code>Int</code>	is	<code>[a / Int]</code>
<code>a -&gt; a</code>	and	<code>Int -&gt; Int</code>	is	<code>[a / Int]</code>
<code>a -&gt; Int</code>	and	<code>Int -&gt; b</code>	is	<code>[a / Int, b / Int]</code>
<code>Int</code>	and	<code>Int</code>	is	<code>[]</code>
<code>a</code>	and	<code>a</code>	is	<code>[]</code>
<code>Int</code>	and	<code>Int -&gt; Int</code>	cannot unify!	
<code>Int</code>	and	<code>a -&gt; a</code>	cannot unify!	
<code>a</code>	and	<code>a -&gt; a</code>	cannot unify!	

# QUIZ

---

What is the unifier of the following two types? \*

1.  $a \rightarrow \text{Int} \rightarrow \text{Int}$

2.  $b \rightarrow c$

- (A) Cannot unify
- (B)  $[a / \text{Int}, b / \text{Int} \rightarrow \text{Int}, c / \text{Int}]$
- (C)  $[a / \text{Int}, b / \text{Int}, c / \text{Int} \rightarrow \text{Int}]$
- (D)  $[b / a, c / \text{Int} \rightarrow \text{Int}]$
- (E)  $[a / b, c / \text{Int} \rightarrow \text{Int}]$



<http://tiny.cc/cse116-unify-ind>



# QUIZ

---

What is the unifier of the following two types? \*

1.  $a \rightarrow \text{Int} \rightarrow \text{Int}$

2.  $b \rightarrow c$

- (A) Cannot unify
- (B)  $[a / \text{Int}, b / \text{Int} \rightarrow \text{Int}, c / \text{Int}]$
- (C)  $[a / \text{Int}, b / \text{Int}, c / \text{Int} \rightarrow \text{Int}]$
- (D)  $[b / a, c / \text{Int} \rightarrow \text{Int}]$
- (E)  $[a / b, c / \text{Int} \rightarrow \text{Int}]$



<http://tiny.cc/cse116-unify-grp>

# QUIZ

---

What is the unifier of the following two types? \*

1.  $a \rightarrow \text{Int} \rightarrow \text{Int}$

2.  $b \rightarrow c$

- (A) Cannot unify
- (B)  $[a / \text{Int}, b / \text{Int} \rightarrow \text{Int}, c / \text{Int}]$
- (C)  $[a / \text{Int}, b / \text{Int}, c / \text{Int} \rightarrow \text{Int}]$
- (D)  $[b / a, c / \text{Int} \rightarrow \text{Int}]$
- (E)  $[a / b, c / \text{Int} \rightarrow \text{Int}]$

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

# Infer: take 2

---

```
-- 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? \*

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

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



<http://tiny.cc/cse116-infer-ind>

# QUIZ

---

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

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

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



<http://tiny.cc/cse116-infer-grp>

# 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!



# Worked Example

---

```
infer sub tEnv (ELam x e) = (sub1, TFun tX' tBody)
```

where

```
tX          = freshTV
```

```
tEnv'       = extendTEnv x tX tEnv
```

```
(sub1, tBody) = infer sub tEnv' e
```

```
tX'         = apply sub1 tX
```

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

where

```
(sub1, t1) = infer sub tEnv e1
```

```
sub2       = unify sub1 t1 Int
```

```
tEnv'      = apply sub2 tEnv
```

```
(sub3, t2) = infer sub2 tEnv' e2
```

```
sub4       = unify sub3 t2 Int
```

# Fresh type variables

---

```
-- | Now has to keep track of current substitution!  
infer :: Subst -> TypeEnv -> Expr -> (Subst, MonoType)  
  
-- Lambda case: simply generate fresh type variable!  
infer tEnv (ELam x e) = TFun tX tBody  
  where  
    tEnv'   = extendTEEnv x tX tEnv  
    tX      = freshTV -- how do we do this?  
    tBody   = infer tEnv' e
```

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 :(

# Typing rules

---

2. We can *instantiate* a type scheme into a type

$$\begin{array}{c} G \mid - e :: \text{forall } a . S \\ \text{[T-Inst]} \text{ -----} \\ G \mid - e :: [a / T] S \end{array}$$

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

$$\begin{array}{c} G \mid - e :: S \\ \text{[T-Gen]} \text{ ----- if not } (a \text{ in } \text{FTV}(G)) \\ G \mid - e :: \text{forall } a . S \end{array}$$

# Polymorphism: the final frontier

---

Back to double identity:

```
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 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 id 5`:

--	<i>TEnv</i>	<i>Expression</i>	<i>Step</i>	<i>Subst</i>	<i>Type</i>
1	[]	<code>let id=\x-&gt;x in id 5</code>	[T-Let]	[]	
2		<code>\x-&gt;x</code>	[T-Abs]		
3	[x:a0]	<code>x</code>	[T-Var]		a0
4		<code>\x-&gt;x</code>			a0 -> a0
5	[]	<code>let id=\x-&gt;x in id 5</code>	generalize a0		
6	tEnv	<code>id 5</code>	[T-App]		
7		<code>id</code>	[T-Var]		
8		<code>id</code>	instantiate		a1 -> a1
9		<code>5</code>	[T-Num]		Int
10		<code>id 5</code>	unify (a1->a1) (Int->a2) [a1/Int, a2/Int]		
10		<code>id 5</code>			Int
11	[]	<code>let id=\x-&gt;x in id 5</code>			Int

Here tEnv = [id : forall a0.a0->a0]

# Representing types

---

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

```
type Id    = String           -- program variables, "x"
type Tid  = String           -- type variables, "a"

data MonoType = TInt          -- Int
              | TFun MonoType MonoType -- T1 -> T2
              | TVar Tid        -- "a"

data PolyType = TMono MonoType
              | TForall Tid PolyType -- forall a.

type TEnv  = [(Id, PolyType)] -- type environment
type Subst = [(Tid, MonoType)] -- type substitution
```



# What we learned

---

**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

---

**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