

CSE 114A

Foundations of Programming Languages

Polymorphism and Type Inference

Leilani H. Gilpin

UC Santa Cruz

Hello World!

- B.S in Computer Science, B.S. in Mathematics at UC San Diego
- M.S. in Computational Mathematics from Stanford University (2013), Ph.D. in EECS from MIT (2020).
- Industry experience
 - Xerox PARC
 - INRIA (France)
 - Sony AI
- Research: The methodologies and technologies for complex systems to explain themselves.
- Hobbies: swimming, cooking, knitting, and org-mode
- Usually teach: CSE 140/240 (AI), CSE 146/246 (AI ethics), CSE 242/142 (machine learning)



Scheme and Lisp in AI

```
(define (tire-object diameter-input pressure #!optional given-name)
  (physob (list diameter-input)
    (lambda ()
      (let-cells (rotation friction (diameter diameter-input))
        (copier pressure friction)
        (inverter pressure rotation)
        (define (insides-name message)
          (case message
            ((diameter) diameter)
            ((pressure) pressure)
            ((rotation) rotation)
            ((friction) friction)
            (else #f)))
          insides-name))
      (if (default-object? given-name)
        `(:,type ,(map name diameter-input))
        given-name)))
```

Hello World!



Swimming and water polo

```
* Tasks
** DONE Start new org file for tutorial
   CLOSED: [2006-05-04 Thu 11:29]
** DONE Outline document
   CLOSED: [2006-05-04 Thu 11:43]
** DONE Write introduction
   CLOSED: [2006-05-04 Thu 11:55]
** TODO Finish document
   SCHEDULED: <2006-05-04 Thu>

* Introduction

Org-mode is a personal information management and outlining tool for
Emacs. This document is intended to give the reader a "feel" for
org-mode and to teach basic operations via step-by-step narration and
plentiful screenshots. It is not intended to replace the manual.

** Obtaining org-mode

You can download org-mode from the org-mode home page. The following
commands will download version 4.27 and unpack it into your ELISP
directory.

: cd #ELISP
BU:-- Orgtutorial.org Top L9 (Org Fill)----Thu May 4 1:20PM 1.17 [#dataf
```

Org mode

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) $(\lambda x \rightarrow x) + 1$
- (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 $T \rightarrow T$, 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:

$T ::= \text{Int}$ -- *integers*
| $T_1 \rightarrow T_2$ -- *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] $n :: \text{Int}$

$e_1 :: \text{Int}$ $e_2 :: \text{Int}$ -- *premises*
[T-Add] -----
 $e_1 + e_2 :: \text{Int}$ -- *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] $G \vdash n :: \text{Int}$

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

[T-Var] $G \vdash x :: T \quad \text{if } x:T \text{ in } G$

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

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

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

Typing rules

$G \vdash e :: T$

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

An expression e is **well-typed** in G if we can derive $G \vdash e :: T$ 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] ----- [T-Num]

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

[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: $[\] \vdash \mathbf{let\ } x = 1 \mathbf{\ in\ } x + 2 :: \mathit{Int}$

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

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

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

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 $[] \vdash \lambda x \rightarrow x :: \text{Int} \rightarrow \text{Int}$
- or $[] \vdash \lambda x \rightarrow x :: (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$
- 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 $\text{Int} \rightarrow \text{Int}$
- in the second application, id needs to have type $(\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$
- 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**: $\text{forall } a . a \rightarrow a$

- where a is a (bound) type variable
- also called a **type scheme**
- Haskell also has polymorphic types, but you don't usually write $\text{forall } a$.

We can **instantiate** this scheme into different types by replacing a in the body with some type, e.g.

- instantiating with Int yields $\text{Int} \rightarrow \text{Int}$
- instantiating with $\text{Int} \rightarrow \text{Int}$ yields $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}$
- 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 $\backslash x \rightarrow X$ comes out as $a \rightarrow a$
3. We can **generalize** this type to **forall** $a . a \rightarrow a$
4. When we apply `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 \rightarrow Int**

Let's formalize this intuition as a type system!

Type system: take 3

Syntax of types

-- Mono-types

$T ::= \text{Int}$ *-- integers*
| $T1 \rightarrow T2$ *-- 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, $G = [z: \text{Int}, \text{id}: \text{forall } a . a \rightarrow a]$

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**

$G \vdash e1 :: S$ $G, x:S \vdash e2 :: T$

[T-Let] -----

$G \vdash \mathbf{let} \ x = e1 \ \mathbf{in} \ e2 :: T$

Typing rules

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

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

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

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

Typing rules

The rest of the rules are the same:

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

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

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

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

Examples

Example 1

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

[T-Var] -----

$[x:a] \vdash x :: a$

[T-Abs] -----

$[\] \vdash \lambda x . x :: a \rightarrow a$

[T-Gen] ----- not (a in FTV([\]))

$[\] \vdash \lambda x . 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: $G1 \vdash \text{id } 5 :: \text{Int}$ where $G1 = [\text{id} : (\text{forall } a . a \rightarrow a)]$:

[T-Var]-----

$G1 \vdash \text{id} :: \text{forall } a . a \rightarrow a$

[T-Inst]----- [T-Num]

$G1 \vdash \text{id} :: \text{Int} \rightarrow \text{Int}$ $G1 \vdash 5 :: \text{Int}$

[T-App]-----

$G1 \vdash \text{id } 5 :: \text{Int}$

Examples

Example 3

Finally, we can derive:

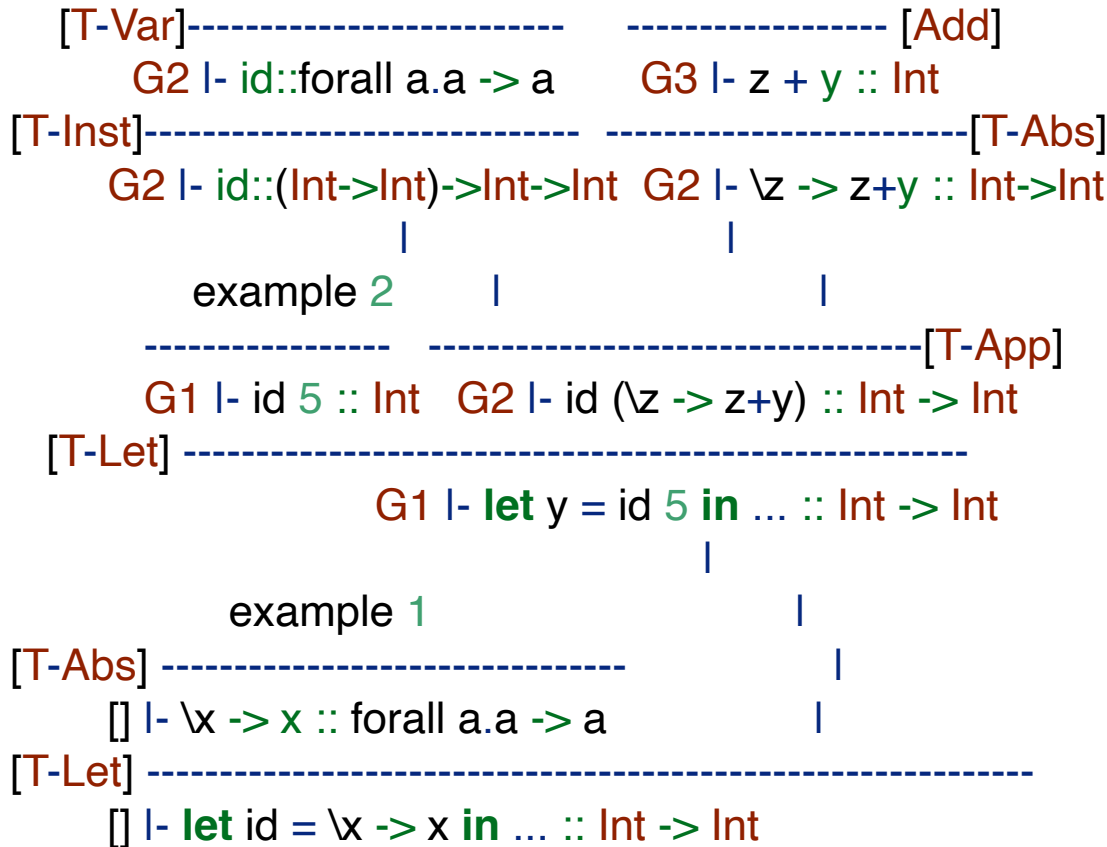
(**let** id = $\lambda x \rightarrow x$ **in**

let y = id 5 **in**

id ($\lambda z \rightarrow z + y$) :: **Int** \rightarrow **Int**

Examples

easy



- 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:

```
data Type
```

```
= TInt      -- Int
```

```
| Type :=> Type  -- T1 -> T2
```

```
| TVar String   -- a, b, c
```

```
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

-- / This is not the final version!!!

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]-----

[] |- ($\lambda x \rightarrow x + 1$) :: ? \rightarrow ??

Inference: tricky bits

Problem 1: Guessing a type

So, if we want to implement

`infer tEnv (ELam x e) = tX :=> tBody`

where

`tEnv' = extendTEnv x tX tEnv`

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

`tBody = infer tEnv' e`

...

Inference: tricky bits

Problem 2: Guessing when to generalize

In the derivation for

(**let** id = $\lambda x \rightarrow x$ **in**

let y = id 5 **in**

id ($\lambda z \rightarrow z + y$) :: **Int** \rightarrow **Int**

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!

[T-Var]-----

$[x:?] \vdash x : \text{Int}$ $[x:?] \vdash 1 :: \text{Int}$

[T-Add]-----

$[x:?] \vdash x + 1 :: ??$ *-- what should "?" be?*

[T-Abs]-----

$[] \vdash (\lambda x . x + 1) :: ? \rightarrow ??$

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

Example

Let's infer the type of $\lambda x \rightarrow x + 1$:

<i>-- TEnv</i>	<i>Expression</i>	<i>Step</i>	<i>Subst</i>	<i>Inferred type</i>
1 []	$\lambda x \rightarrow x + 1$	[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 []	$\lambda x \rightarrow x + 1$		Int \rightarrow Int	

Example

1. Infer the type of $(\lambda x \rightarrow x + 1)$ in $[]$ (apply [T-Abs])
2. For the type of x , pick *fresh type variable* (say, a_0); infer the type of $x + 1$ in $[x:a_0]$ (apply [T-Add])
3. Infer the type of x in $[x:a_0]$ (apply [T-Var]); result: a_0
4. [T-Add] *imposes a constraint*: its LHS must be of type Int , so *unify* a_0 and Int and update the *current substitution* to $[a_0 / \text{Int}]$
5. *Apply* the current substitution $[a_0/\text{Int}]$ to the type environment $[x:a_0]$ to get $[x:\text{Int}]$. Infer the type of 1 in $[x:\text{Int}]$ (apply [T-Num]); result: Int
6. [T-Add] *imposes a constraint*: its RHS must be of type Int , so *unify* Int and Int ; current substitution doesn't change\
7. By conclusion of [T-Add]: return Int as the inferred type\
8. By conclusion of [T-Lam]: return $\text{Int} \rightarrow \text{Int}$ as the inferred type

Unification

The **unification** problem: given two 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:

a and **Int** is $[a / \text{Int}]$

a -> a and **Int -> Int** is $[a / \text{Int}]$

a -> Int and **Int -> b** is $[a / \text{Int}, b / \text{Int}]$

Int and **Int** is $[]$

a and **a** is $[]$

Int and **Int -> Int** cannot unify!

Int and **a -> a** cannot unify!

a and **a -> a** 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

(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!

-- I 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' = extendTEEnv x tX tEnv

tX = freshTV *-- we'll get to this*

(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!

Fresh type variables

-- I Now has to keep track of current substitution!

`infer :: Subst -> TypeEnv -> Expr -> (Subst, Type)`

-- Lambda case: simply generate fresh type variable!

`infer tEnv (ELam x e) = 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 :(

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

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