

# Type Systems in Practice

suhorng

# Before We Start

- You may safely skip any judgments and inference rules in this slide (things like  $\Gamma \vdash e : t, \frac{(x:t) \in \Gamma}{\Gamma \vdash x:t}$  )
- The accompanying code does not correspond exactly to the (pseudo-) code in this slide. The `Infer` module in the accompanying code actually transforms its input, `LC` terms, into `STLC` terms or `SysF` terms.
- Outline
  - Language Syntax and Notations
  - Bidirectional Typing
  - Type Inference for Simply-Typed  $\lambda$ -Calculus
  - Hindley-Milner Type System

# Language Syntax (Incomplete)

In Slide	In Code	Analogy
$x$	VAR "x"	<code>x</code>
$\lambda x.e$ $\lambda(x:t).e$	LAM ("x", e) LAM ("x", t, e) ALAM ("x", t, e)	<code>\x -&gt; e</code> <code>function (x) { return e; }</code> <code>\(x :: t) -&gt; e</code> <code>(x : t) =&gt; e</code> <code>[???](t x) { return e; }</code>
$e_1 e_2$	AP (e1, e2)	<code>e1 e2</code> <code>e1(e2)</code>
<b>let</b> $x = e_1$ <b>in</b> $e_2$	LET ("x", e1, e2)	<code>let x = e1 in e2</code> <code>val x = e1; e2</code>
A	TVAR "A"	<code>a, T (Haskell)</code> <code>typename A, T</code> (roughly)
$t_1 \rightarrow t_2$	TARR (t1, t2)	<code>t1 -&gt; t2</code> <code>function&lt;t2&gt;(t1)&gt;</code> <code>t2(*) (t1)</code> (roughly)

# Examples

- Slide:  $\lambda x. \lambda f. f x$

- Code:

```
LAM ("x", LAM ("f", AP (VAR "f", VAR "x"))))
```

- Javascript

```
function (x) {  
  return function(f) { return f(x); }  
}
```

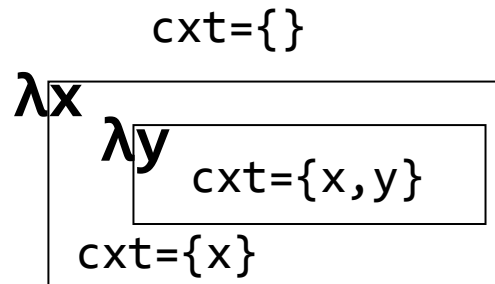
- C++ (roughly):

```
[](auto x) {  
  return [=](auto f) {  
    return f(x);  
  }  
}
```

# What Are Contexts

- Variables visible in current scope

```
function (x) {  
  //   ^ here: [x]  
  return function (y) { return x; }  
  //           ^ here: [x, y]  
  //   ^ here: [x]  
}
```



- A recurring theme

```
f (cxt, LAM (x, t, e)) = .... f ((x,t) :: cxt, e) ....
```

# How Type Systems are Defined

- $e : t$  means  $e$  has type  $t$
- When  $e_1 e_2$  occurs in some context `cxt`, for some  $\alpha, \beta$  we will have...
  - $e_1 : \alpha \rightarrow \beta$  in `cxt`
  - $e_2 : \alpha$  in `cxt`
  - $e_1 e_2 : \beta$  in `cxt`
- The definition is a sort of relation and it holds iff the “shape” is correct.
  - $e_1 e_2 : \mathbf{int}$  with  $e_1 : \mathbf{char} \rightarrow \mathbf{int}, e_2 : \mathbf{char}$ : OK
  - $e_1 e_2$  with  $e_1 : \mathbf{bool} \rightarrow \mathbf{string}, e_2 : \mathbf{int}$ : NO
  - $e_1 e_2$  with  $e_1 : \mathbf{int}$ : NO

# Other rules

- $x$  has type  $t$  precisely when  $(x:t) \in \mathbf{cxt}$   
(Lookup the type of  $x$  in the current context)
- When  $\lambda x.e$  occurs in the context  $\mathbf{cxt}$ , then for some  $\alpha, \beta \dots$ 
  - $x$  has type  $\alpha$
  - $e$  has type  $\beta$  in the context  $\{x:\alpha\} \cup \mathbf{cxt}$

```
auto f(int x) -> bool { return x == 0; }  
//                               ^ cxt={x:int}  
// 'x' has type 'int'  
// 'x == 0' has type 'bool'  
// 'f' (will) has type 'bool(*) (int)', or roughly 'int -> bool'
```

- The case for  $\lambda(x:t).e$  is similar. Just that  $x$  has type  $t$ .

# Problems with Typing Rules

- $e_1 e_2 : \exists \alpha, \beta \text{ s.t. } \dots$ 
  - $e_1 : \alpha \rightarrow \beta$
  - $e_2 : \alpha$
  - $e_1 e_2 : \beta$
  - There's simply no clue for  $\alpha$  even if we know  $\beta$ .
- $\lambda x. e : \exists \alpha, \beta \text{ s.t. } \dots$ 
  - $x : \alpha$
  - $e : \beta$
  - $(\lambda x. e) : \alpha \rightarrow \beta$
  - Gotta know  $\alpha \rightarrow \beta$  in order to proceed. Or do we?



# First Solution

- Sticking to  $\lambda(x:t).e$ , i.e. users are required to annotate  $\alpha$ 
  - This is the only case where new variables will be added to the context, and we know the variables' types now.
  - $e$  can only refer to variables in the current context
  - Hence we can infer the type of  $e$ .
- $x$  : Lookup `cxt`
- $e_1 e_2$  : Infer the type of  $e_1$  and  $e_2$ , respectively. Check their shape.
- $\lambda(x:t).e$  : Infer the type (say  $t'$ ) of  $e$ . Then the type of  $\lambda(x:t).e$  is  $t \rightarrow t'$

# First Solution

```
let rec infer = function
  cxt, VAR x -> List.assoc x cxt
| cxt, LAM (x, t1, e) -> TARR (t1, infer ((x,t1)::cxt, e))
| cxt, (AP (e1, e2) as e) ->
  (match infer (cxt, e1), infer (cxt, e2) with
   TARR (t1, t2), t1' when t1 = t1' -> t2
  | _ -> ERROR!!!
```

- Problem: it's too verbose

```
\(z : A). let ididid = \(h : (A → A) → A → A). h in
  ididid (\(f : A → A). f) (\(x:A).x) z
```

- Why bother annotating  $f$  and  $x$  if the type of *ididid* is known (hence the type of its arguments)?

# Improvement via Bidirectional Typing

- Another motivation: We wish to turn judgments into functions, and make typing rules syntax-directed.

$$\frac{(x : t) \in \Gamma}{\Gamma \vdash x : t}$$

$$\frac{\Gamma, (x:t) \vdash e : t'}{\Gamma \vdash (\lambda x. e) : t \rightarrow t'}$$

$$\frac{\Gamma \vdash e_1 : t \rightarrow t' \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t'}$$

# Improvement via Bidirectional Typing

- Another motivation: We wish to turn judgments into functions, and make typing rules syntax-directed.

$$\frac{(x : t) \in \Gamma}{\Gamma \vdash x : t} \quad \frac{\Gamma, (x:t) \vdash e : t'}{\Gamma \vdash (\lambda x. e) : t \rightarrow t'} \quad \frac{\Gamma \vdash e_1 : t \rightarrow t' \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t'}$$

- Annotate judgments with I/O modes to get a function
  - $\Gamma_{\text{in}} \vdash x_{\text{in}} : t_{\text{out}}$  : Use following principles to examine and rewrite typing rules accordingly.

$$\text{recursion } \Uparrow \frac{(\Gamma, x:t)_{\text{shown\_in}} \vdash e_{\text{shown\_in}} : t'_{\text{assumed\_out, free}}}{\Gamma_{\text{assumed\_in}} \vdash (\lambda(x:t). e)_{\text{assumed\_in}} : (t \rightarrow t')_{\text{shown\_out}}} \Downarrow \text{return}$$

- Separating type checking and inference:

$$\Gamma_{\text{in}} \vdash e_{\text{in}} \Rightarrow t_{\text{out}} \quad (\text{infer}) \quad \Gamma_{\text{in}} \vdash e_{\text{in}} \Leftarrow t_{\text{in}} \quad (\text{check})$$

# Bidirectional Typing, Intuitively

- Separating type **checking** and **inference**. If we already know  $t$ , **check**  $e$  against  $t$ . Otherwise **infer**  $t$  from  $e$ .

```
val check : context * expr * type -> unit
val infer : context * expr -> type
```

- $x$ : Simple, look up the type of  $x$  in `cxt`
- If an expression  $e$  can be inferred to have type  $t$ , then it can be checked against  $t$ . That is,

```
function check(cxt, e, t):
  let t' = infer(cxt, e)
  if t ≠ t':
    error
```

†See appendix for typing rules.

# Bidirectional Typing, Intuitively

- The case  $e_1 e_2$ 
  1.  $e_1 : \alpha \rightarrow \beta$
  2.  $e_2 : \alpha$
  3.  $e_1 e_2 : \beta$
- There's no way to know  $\alpha$ , regardless whether  $\beta$  is known or not.
- Must **infer** the type of  $e_1$ . Then  $e_2$  can be check to have type  $\alpha$ . Luckily, most cases the funtion  $e_1$  are known from the context.

```
function infer(cxt, AP (e1, e2)):  
  -- Failed if e1 is not of function type  
  let TARR (t1, t2) = infer(cxt, e1)  
  check(cxt, e2, t1)  
  return t2
```

# Bidirectional Typing, Intuitively

- The case  $\lambda x.e$ ,  $\lambda(x:t).e$ 
  1.  $x : \alpha$
  2.  $e : \beta$
  3.  $(\lambda x.e) : \alpha \rightarrow \beta$ ,  $(\lambda(x:\alpha).e) : \alpha \rightarrow \beta$
- To check is ~~human~~ simple.

```
function check(cxt, LAM (x, e), TARR (t, t')):  
  check((x,t) :: cxt, e, t')
```

- To infer, ~~divine~~  $x$  needs an annotation.

```
function infer(cxt, ALAM (x, t, e)):  
  return (TARR (t, infer ((x,t) :: cxt, e)))
```

# Bidirectional Typing, Intuitively

- Even better: we allow the user annotate expressions with types
- An expression “ $e : t$ ” can be **inferred** to have type  $t$  while we check  $e$  against  $t$ .

```
function infer(cxt, ANNO (e, t)):  
  check(cxt, e, t)  
  return t
```

- Examples

```
(\x. \f. f x) : (A → (A → B) → B)  
\(z : A). let ididid = \ (h : (A → A) → A → A). h in  
  ididid (\f.f) (\x.x) z
```



# An Bidirectional Typing Example

$$\vdash (\lambda(x:a \rightarrow a).x)(\lambda y.y) \Rightarrow$$

# An Bidirectional Typing Example

$$\frac{\vdash \lambda(x:a \rightarrow a).x \Rightarrow \quad \vdash \lambda y.y \Leftarrow}{\vdash (\lambda(x:a \rightarrow a).x)(\lambda y.y) \Rightarrow}$$

# An Bidirectional Typing Example

$$\frac{\frac{x:a \rightarrow a \vdash x \Rightarrow}{\vdash \lambda(x:a \rightarrow a).x \Rightarrow} \quad \vdash \lambda y.y \Leftarrow}{\vdash (\lambda(x:a \rightarrow a).x)(\lambda y.y) \Rightarrow}$$

# An Bidirectional Typing Example

$$\frac{\frac{\overline{x:a \rightarrow a \vdash x \Rightarrow a \rightarrow a}}{\vdash \lambda(x:a \rightarrow a).x \Rightarrow} \quad \vdash \lambda y.y \Leftarrow}{\vdash (\lambda(x:a \rightarrow a).x)(\lambda y.y) \Rightarrow}$$

# An Bidirectional Typing Example

$$\frac{\frac{\overline{x:a \rightarrow a \vdash x \Rightarrow a \rightarrow a}}{\vdash \lambda(x:a \rightarrow a).x \Rightarrow (a \rightarrow a) \rightarrow a \rightarrow a} \quad \vdash \lambda y.y \Leftarrow a \rightarrow a}{\vdash (\lambda(x:a \rightarrow a).x)(\lambda y.y) \Rightarrow a \rightarrow a}$$

# An Bidirectional Typing Example

$$\frac{\frac{\overline{x:a \rightarrow a \vdash x \Rightarrow a \rightarrow a}}{\vdash \lambda(x:a \rightarrow a).x \Rightarrow (a \rightarrow a) \rightarrow a \rightarrow a} \quad \frac{y:a \vdash y \Leftarrow a}{\vdash \lambda y.y \Leftarrow a \rightarrow a}}{\vdash (\lambda(x:a \rightarrow a).x)(\lambda y.y) \Rightarrow a \rightarrow a}$$

# An Bidirectional Typing Example

$$\frac{\frac{\overline{x:a \rightarrow a \vdash x \Rightarrow a \rightarrow a}}{\vdash \lambda(x:a \rightarrow a).x \Rightarrow (a \rightarrow a) \rightarrow a \rightarrow a} \quad \frac{\frac{\overline{y:a \vdash y \Rightarrow a}}{y:a \vdash y \Leftarrow a}}{\vdash \lambda y.y \Leftarrow a \rightarrow a}}{\vdash (\lambda(x:a \rightarrow a).x)(\lambda y.y) \Rightarrow a \rightarrow a}$$

# An Equational Approach

- The case  $e_1 e_2$ 
  1.  $e_1 : \alpha \rightarrow \beta$
  2.  $e_2 : \alpha$
  3.  $e_1 e_2 : \beta$
- Does not knowing  $\alpha$  really matter?
  - Set up a constraint and solve it latter!
  - Let  $e_1 : t$  and  $e_2 : t'$
  - $t = t' \rightarrow t''$  for some unknown  $t''$



# An Equational Approach

$$\vdash \lambda f. \lambda z. f\ z\ z : \_$$

# An Equational Approach

$$\frac{f:t_1 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_1, z:t_2$ .

$$\frac{\Gamma \vdash f z z : \_}{\frac{f:t_1 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}}$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_1, z:t_2$ .

$$\frac{\frac{\frac{\Gamma \vdash f z : \_}{\Gamma \vdash f z z : \_}}{f:t_1 \vdash \lambda z. f z z : \_}}{\vdash \lambda f. \lambda z. f z z : \_}$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_1, z:t_2$ .

$$\frac{\frac{\frac{\Gamma \vdash f : t_1}{\Gamma \vdash f z : \_}}{\Gamma \vdash f z z : \_}}{f:t_1 \vdash \lambda z. f z z : \_} \vdash \lambda f. \lambda z. f z z : \_$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_1, z:t_2$ .

$$\frac{\frac{\frac{\Gamma \vdash f : t_1 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : \_}}{\Gamma \vdash f z z : \_}}{\frac{f:t_1 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}}$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_1, z:t_2$ .

$$\frac{\frac{\frac{\Gamma \vdash f : t_1 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : t_3}}{\Gamma \vdash f z z : \_}}{\frac{f:t_1 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}}$$

$$t_1 = t_2 \rightarrow t_3$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_2 \rightarrow t_3, z:t_2$ .

$$\frac{\frac{\frac{\Gamma \vdash f : t_2 \rightarrow t_3 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : t_3}}{\Gamma \vdash f z z : \_}}{\frac{f:t_2 \rightarrow t_3 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}}$$

$$t_1 = t_2 \rightarrow t_3$$



# An Equational Approach

- Write  $\Gamma \equiv f:t_2 \rightarrow t_3, z:t_2$ .

$$\frac{\frac{\frac{\Gamma \vdash f : t_2 \rightarrow t_3 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : t_3} \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z z : \_}}{\frac{f:t_2 \rightarrow t_3 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}}}$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_2 \rightarrow t_3, z:t_2$ .

$$\begin{array}{c}
 \frac{\Gamma \vdash f : t_2 \rightarrow t_3 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : t_3} \quad \Gamma \vdash z : t_2 \\
 \hline
 \Gamma \vdash f z z : t_4 \\
 \hline
 \frac{f:t_2 \rightarrow t_3 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}
 \end{array}$$

$$t_3 = t_2 \rightarrow t_4$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_2 \rightarrow t_2 \rightarrow t_4, z:t_2$  .

$$\begin{array}{c}
 \frac{\Gamma \vdash f : t_2 \rightarrow t_2 \rightarrow t_4 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : t_2 \rightarrow t_4} \quad \Gamma \vdash z : t_2 \\
 \hline
 \Gamma \vdash f z z : t_4 \\
 \hline
 \frac{f:t_2 \rightarrow t_2 \rightarrow t_4 \vdash \lambda z. f z z : \_}{\vdash \lambda f. \lambda z. f z z : \_}
 \end{array}$$

$$t_3 = t_2 \rightarrow t_4$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_2 \rightarrow t_2 \rightarrow t_4, z:t_2$  .

$$\frac{\frac{\frac{\Gamma \vdash f : t_2 \rightarrow t_2 \rightarrow t_4 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : t_2 \rightarrow t_4} \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z z : t_4}}{\frac{f:t_2 \rightarrow t_2 \rightarrow t_4 \vdash \lambda z. f z z : t_2 \rightarrow t_4}{\vdash \lambda f. \lambda z. f z z : \_}}$$

# An Equational Approach

- Write  $\Gamma \equiv f:t_2 \rightarrow t_2 \rightarrow t_4, z:t_2$  .

$$\begin{array}{c}
 \frac{\Gamma \vdash f : t_2 \rightarrow t_2 \rightarrow t_4 \quad \Gamma \vdash z : t_2}{\Gamma \vdash f z : t_2 \rightarrow t_4} \quad \Gamma \vdash z : t_2 \\
 \hline
 \Gamma \vdash f z z : t_4 \\
 \hline
 f:t_2 \rightarrow t_2 \rightarrow t_4 \vdash \lambda z. f z z : t_2 \rightarrow t_4 \\
 \hline
 \vdash \lambda f. \lambda z. f z z : (t_2 \rightarrow t_2 \rightarrow t_4) \rightarrow t_2 \rightarrow t_4
 \end{array}$$

# An Equational Approach

$$\lambda f. \lambda z. fzz$$

- Allocate fresh unknown  $t_1$
- Case  $\lambda x. e$  with  $x \equiv f$ ,  
 $e \equiv \lambda z. fzz$ 
  - $x : \alpha$
  - $e : \beta$
  - $(\lambda x. e) : \alpha \rightarrow \beta$
- Context:  $\square$

Now  $f : t_1$

# An Equational Approach

$\lambda f. \lambda z. fzz$

- Allocate fresh unknown  $t_2$
- Case  $\lambda x. e$  with  $x \equiv z$ ,  
 $e \equiv fzz$ 
  - $x : \alpha$
  - $e : \beta$
  - $(\lambda x. e) : \alpha \rightarrow \beta$
- Context:  $[(f:t_1)]$

Now  $f : t_1$        $z : t_2$

# An Equational Approach

$\lambda f. \lambda z. fzz$

- Recursion.
- Case  $e_1 e_2$  with  $e_1 \equiv fz$ ,  
 $e_2 \equiv z$ 
  - $e_1 : \alpha \rightarrow \beta$
  - $e_2 : \alpha$
  - $e_1 e_2 : \beta$
- Context:  $[(f:t_1), (z:t_2)]$

Now  $f : t_1 \quad z : t_2$



# An Equational Approach

$$\lambda f. \lambda z. \boxed{fz} z$$

- Recursion.
- Case  $e_1 e_2$  with  $e_1 \equiv f$ ,  
 $e_2 \equiv z$ 
  - $e_1 : \alpha \rightarrow \beta$
  - $e_2 : \alpha$
  - $e_1 e_2 : \beta$
- Context:  $[(f:t_1), (z:t_2)]$

Now  $f : t_1$        $z : t_2$

# An Equational Approach

$$\lambda f. \lambda z. \boxed{f} \, z z$$

- Lookup context and return  $t_1$ .
- Case  $x$  with  $x :\equiv f$ 
  - $x : \alpha$
  - $(x:\alpha) \in \mathbf{cxt}$
- Context:  $[(f:t_1), (z:t_2)]$

Now  $f : t_1 \quad z : t_2$

# An Equational Approach

$$\lambda f. \lambda z. f \boxed{z} z$$

- Lookup context and return  $t_2$ .
- Case  $x$  with  $x :\equiv z$ 
  - $x : \alpha$
  - $(x:\alpha) \in \mathbf{cxt}$
- Context:  $[(f:t_1), (z:t_2)]$

Now  $f : t_1 \quad z : t_2$

# An Equational Approach

$\lambda f. \lambda z. fz z$

- Allocate fresh unknown  $t_3$ . Return  $t_3$ .

$t_1 = t_2 \rightarrow t_3$

- Case  $e_1 e_2$  with  $e_1 \equiv f$ ,  
 $e_2 \equiv z$

- $e_1 : \alpha \rightarrow \beta$
- $e_2 : \alpha$
- $e_1 e_2 : \beta$

- Context:  $[(f:t_1), (z:t_2)]$

Now  $f : t_1 \quad z : t_2$

# An Equational Approach

$\lambda f. \lambda z. fz$   $z$

- Lookup context and return  $t_2$ .

$t_1 = t_2 \rightarrow t_3$

- Case  $x$  with  $x :\equiv z$

- $x : \alpha$
- $(x:\alpha) \in \mathbf{cxt}$

- Context:  $[(f:t_1), (z:t_2)]$

Now  $f : t_1$        $z : t_2$

# An Equational Approach

$\lambda f. \lambda z. \boxed{fzz}$

- Allocate fresh unknown  $t_4$ . Return  $t_4$ .

$$t_1 = t_2 \rightarrow t_3$$

$$t_3 = t_2 \rightarrow t_4$$

- Case  $e_1 e_2$  with  $e_1 \equiv fz$ ,  
 $e_2 \equiv z$

- $e_1 : \alpha \rightarrow \beta$

- $e_2 : \alpha$

- $e_1 e_2 : \beta$

- Context:  $[(f:t_1), (z:t_2)]$

Now  $f : t_1$        $z : t_2$        $fz : t_3$

# An Equational Approach

$\lambda f. \lambda z. fzz$

- Return  $t_2 \rightarrow t_4$

$$t_1 = t_2 \rightarrow t_3$$

$$t_3 = t_2 \rightarrow t_4$$

- Case  $\lambda x. e$  with  $x \equiv z$ ,  
 $e \equiv fzz$

- $x : \alpha$

- $e : \beta$

- $(\lambda x. e) : \alpha \rightarrow \beta$

- Context:  $[(f:t_1)]$

Now  $f : t_1$        $z : t_2$        $fz : t_3$        $fzz : t_4$

# An Equational Approach

$\lambda f. \lambda z. fzz$

- Return  $t_1 \rightarrow (t_2 \rightarrow t_4)$

$$t_1 = t_2 \rightarrow t_3$$

$$t_3 = t_2 \rightarrow t_4$$

- Case  $\lambda x. e$  with  $x \equiv f$ ,  
 $e \equiv \lambda z. fzz$

- $x : \alpha$

- $e : \beta$

- $(\lambda x. e) : \alpha \rightarrow \beta$

- Context:  $\square$

Now  $f : t_1$        $z : t_2$        $fz : t_3$        $fzz : t_4$        $(\lambda z. fzz) : t_2 \rightarrow t_4$



# Solving Equations

- At top level:  $(\lambda f. \lambda z. fzz) : t_1 \rightarrow t_2 \rightarrow t_4$
- $(f : t_1), (z : t_2), (fz : t_3), (fzz : t_4), (\lambda z. fzz) : t_2 \rightarrow t_4$

$$t_1 = t_2 \rightarrow t_3$$

$$t_3 = t_2 \rightarrow t_4$$

# Solving Equations

- At top level:  $(\lambda f. \lambda z. f z z) : (t_2 \rightarrow t_3) \rightarrow t_2 \rightarrow t_4$
- $(f : t_2 \rightarrow t_3), (z : t_2), (f z : t_3), (f z z : t_4), (\lambda z. f z z) : t_2 \rightarrow t_4$

$$t_1 = t_2 \rightarrow t_3$$

$$t_3 = t_2 \rightarrow t_4$$

- Substitute  $t_2 \rightarrow t_3$  for  $t_1$ !

# Solving Equations

- At top level:  $(\lambda f. \lambda z. f z z) : (t_2 \rightarrow t_2 \rightarrow t_4) \rightarrow t_2 \rightarrow t_4$
- $(f : t_2 \rightarrow t_2 \rightarrow t_4), (z : t_2), (f z : t_2 \rightarrow t_4), (f z z : t_4),$   
 $(\lambda z. f z z) : t_2 \rightarrow t_4$

$$t_1 = t_2 \rightarrow t_3$$

$$t_3 = t_2 \rightarrow t_4$$

- Substitute  $t_2 \rightarrow t_4$  for  $t_3$ !

# Solving Equations

- At top level:  $(\lambda f. \lambda z. f z z) : (t_2 \rightarrow t_2 \rightarrow t_4) \rightarrow t_2 \rightarrow t_4$
- $(f : t_2 \rightarrow t_2 \rightarrow t_4), (z : t_2), (f z : t_2 \rightarrow t_4), (f z z : t_4),$   
 $(\lambda z. f z z) : t_2 \rightarrow t_4$

$$t_1 = t_2 \rightarrow t_3$$

$$t_3 = t_2 \rightarrow t_4$$

- No more  $t_1$  and  $t_3$ . Done!

$$(\lambda f. \lambda z. f z z) : (A \rightarrow A \rightarrow B) \rightarrow A \rightarrow B$$

- There might be equations of form  $T_1 \rightarrow T_2 = T_3 \rightarrow T_4$ .  
We can simply solve  $T_1 = T_3$  and  $T_2 = T_4$  recursively,  
arriving at equations of form  $t_i = T_j$ .

# Disjoint Sets to the Rescue

- Union and Find!
- Our data type for meta-types:

```
type typ = TVAR of (metavar ref)
          | TARR of typ * typ
and metavar = UNLINK of string
              | LINK of typ
```

- Disjoint set (metavar ref) is integrated into (meta-) types typ
- A 'a ref is a mutable variable of type 'a

```
let tvar = ref (UNLINK "x")
let t = TARR (TVAR tvar, TVAR var)
print t      -- => x -> x

let t' = TVAR (ref (UNLINK "A"))
tvar := BOUND (TARR (t', t'))
print t      -- => (t -> t) -> t -> t
```

# Unification: Solve an Equation

- $t_i = T$  (or  $= (_ \rightarrow _), \dots$ )  $\Rightarrow$  substitute RHS for  $t_i$  (\*)
- $T_1 \rightarrow T_2 = T_3 \rightarrow T_4 \Rightarrow$  Solve  $T_1 = T_3, T_2 = T_4$  recursively.

```
function unify(TVAR (r = LINK t), t'):  -- ← Find
    unify(t, t')

function unify(TVAR (r = UNLINK x), TVAR (s = UNLINK y)), x == y:
    no-op

function unify(TVAR (r = UNLINK x), t):  -- ← Union
    if occurs(x, t):  -- ← See ◇◇page
        ERROR!!
    r := LINK t

function unify(TARR (t1, t2), TARR (t3, t4)):
    unify(t1, t3)
    unify(t2, t4)
```

- The case for `t`, `TVAR _` the same

# Final Step

- Case  $x$

```
function typeinfer(cxt, VAR x):  
  return (lookup x in cxt)
```

- Case  $e_1 e_2$

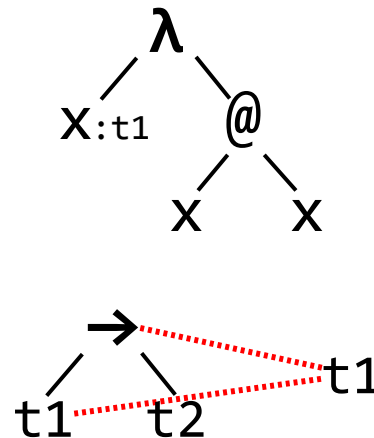
```
function typeinfer(cxt, AP (e1, e2)):  
  let t1 = typeinfer(cxt, e1)  
  let t2 = typeinfer(cxt, e2)  
  let t = fresh_var()  
  unify(t1, TARR (t2, t))  
  return t
```

- Case  $\lambda x.e$

```
function typeinfer(cxt, LAM (x, e)):  
  let t = fresh_var()  
  return (TARR (t, typeinfer((x,t) :: cxt, e)))
```

## (\*) Occurs Check

- Infer  $\lambda x.xx$ : Get equation  $x : t_1, xx : t_2$  and  $t_1 = t_1 \rightarrow t_2$ 
  - Cannot eliminate  $t_1$  by substituting  $t_1 \rightarrow t_2$  for  $t_1$ !



- In equation  $t_i = T$ , it's an error if  $t_i$  occurs in  $T$ .



# Pseudo(?) Code...

- The accompanying code further handles exception and translate LC terms into STLC terms.

```
type typ = TVAR of metavar ref | TARR of typ * typ
and metavar = UNLINK of string | LINK of typ

let rec unify = function
  TVAR {contents = LINK t}, t' |
  t', TVAR {contents = LINK t} ->
    unify (t, t')
  | TVAR ({contents = UNLINK x}), TVAR ({contents = UNLINK x'})
    when x = x' -> ()
  | TVAR ({contents = UNLINK x} as r), t |
  t, TVAR ({contents = UNLINK x} as r) ->
    if occurs (x, t) then raise Occurs_check else r := LINK t
  | TARR (t1, t2), TARR (t1', t2') ->
    (unify (t1, t1'); unify (t2, t2'))

let rec infer = function
  cxt, VAR x -> List.assoc x cxt
  | cxt, LAM (x, e) ->
    let t = fresh_var () in
    TARR (t, infer ((x, t)::cxt, e))
  | cxt, AP (e1, e2) ->
    let t = fresh_var () in
    (unify (infer (cxt, e1), TARR (infer (cxt, e2), t));
     t)
```

# Polymorphism: Generic the Easy Way

- C++ Style:

```
template<typename a>    a id(a x) { return x; }  
id<int>(...)  
id<shared_ptr<expr_t>>(...)
```

Different code (semantics) when instantiated with different types

- Functional Style:
  - $(\lambda x.x) : A \rightarrow A$  : OK.
  - $(\lambda x.x) : B \rightarrow B$  : OK.
  - The same term  $(\lambda x.x) : T \rightarrow T$  is OK for any type  $T$ !
  - Let give it the type  $\forall T. T \rightarrow T$

# Polymorphism: Girard's System F

In Slide	In Code	Analogy
$\Lambda\alpha.e$	TLAM ("a", e)	<pre>def foo[a](...) = ... fn bar&lt;a&gt;(...) { ... } template&lt;typename a&gt; e</pre>
$e[t]$	TAP (e, t)	<pre>e&lt;t&gt;()</pre>
A	TVAR "A"	<pre>a, T (Haskell) typename A, T    (roughly)</pre>
$t_1 \rightarrow t_2$	TARR (t1, t2)	<pre>t1 -&gt; t2 function&lt;t2(t1)&gt; t2(*) (t1)    (roughly)</pre>
$\forall a.t$	TALL (a, t)	<pre>forall a. t {a : Set 1} -&gt; t ∀{a} -&gt; t</pre>

# Examples

- Slide & Code:  $\Lambda a. \Lambda b. \lambda(x:a). \lambda(f:a \rightarrow b). fx$

```
TALL("a", TALL ("b",  
  LAM ("x", TVAR "A",  
    LAM ("f", TARR (TVAR "A", TVAR "B"),  
      AP (VAR "f", VAR "x")))))
```

- Rust: (not sure)

```
fn app<A,B>(x : A, f : fn(A) -> B) -> B {  
  f(x)  
}
```

- C++ (roughly):

```
template<typename A,typename B>  
B app(A x, function<B(A)> f) {  
  return f(x);  
}
```

# Examples

$\Lambda T. \Lambda S.$

$$\underline{(\Lambda \alpha. \Lambda \beta. \lambda(f: \alpha \rightarrow \beta). \lambda(z: \alpha). f z) [T] [S \rightarrow S] (\lambda(x: T). \lambda(y: S). y)}$$

$\rightsquigarrow \Lambda T. \Lambda S.$

$$\underline{(\Lambda \beta. \lambda(f: \boxed{T} \rightarrow \beta). \lambda(z: \boxed{T}). f z) [S \rightarrow S] (\lambda(x: T). \lambda(y: S). y)}$$

$\rightsquigarrow \Lambda T. \Lambda S.$

$$\underline{(\lambda(f: T \rightarrow \boxed{S \rightarrow S}). \lambda(z: T). f z)(\lambda(x: T). \lambda(y: S). y)}$$

$\rightsquigarrow \Lambda T. \Lambda S.$

$$\lambda(z: T) \boxed{(\lambda(x: T). \lambda(y: S). y)} \underline{z}$$

$\rightsquigarrow \Lambda T. \Lambda S.$

$$\lambda(z: T) \lambda(y: S). y$$

# Damas-Hindley-Milner System

- Generalize at `let`.  $\forall$ -quantifiers are only allowed at “top” level
  - $\forall a. \forall b. a \rightarrow b \rightarrow b$  : OK
  - $\forall a. \forall b. b \rightarrow (a \rightarrow b) \rightarrow b$  : OK
  - $(\forall a. a \rightarrow a) \rightarrow \text{Int}$  : Not OK
- A type together with a list of  $\forall$ -quantified variables

```
type typescheme = POLY of string list * typ
```

- Generalization: (Not typeable in previous systems!)

`let id =  $\lambda x. x$  in`  $\Leftarrow$  Generalize at this point  
*id id id*

# Damas-Hindley-Milner System

$\lambda u.$

(**let**  $id = \lambda x.x$  **in**  
 $id\ id\ id$ )

- Inferred to be  $t_1 \rightarrow t_1$  by the preceding algorithm.
- $t_1$  is free: UNLINK "t1" and is not bound in cxt
  - Assume that  $u : t_3$ .
  - cxt-bound type variables like UNLINK "t3" in  $\lambda(u : t_3). \dots$  should not be generalized

# Damas-Hindley-Milner System

$\lambda u.$

(**let**  $id$   $= \lambda x. x$  **in**  
 $id\ id\ id$ )

- Generalize unbound free variables ( $t_1$  in this case)
- $id : \forall t_1. t_1 \rightarrow t_1$



# Damas-Hindley-Milner System

$\lambda u.$

(**let**  $id = \Lambda t_1. \lambda(x:t_1). x$  **in**  
 $id[(t_2 \rightarrow t_2) \rightarrow t_2 \rightarrow t_2] id[t_2 \rightarrow t_2] id[t_2]$  )

- Being explicit...
- Each use of  $id$  is of different type.  $t_2$  is another unbound type variable.

# Upgrading our Algorithm

- Case  $x$ : Instantiate the term. Replace  $\forall$ -quantified type variables with fresh type variables.

```
function typeinfer(cxt, VAR x):  
  return (instantiate(lookup x in cxt))
```

- Instantiate:  $\forall a. \forall b. a \rightarrow (a \rightarrow b) \rightarrow b$  becomes  $t_i \rightarrow (t_i \rightarrow t_j) \rightarrow t_j$  for fresh type variables  $t_i, t_j$

- Case **let**

```
function typeinfer(cxt, LET (x, e1, e2)):  
  let t1 = typeinfer(cxt, e1)  
  return typeinfer((x,  $\forall a. t1$ ) :: cxt, e2)
```

- Quantify: calculate free variables  $\bar{a}$  of  $t_1$  which are unbound in  $\text{cxt}$ , i.e.  $\bar{a} \equiv FV(t_1) \setminus (\bigcup_{t \in \text{cxt}} FV(t))$ .

# Remarks

- Bidirectional often allows an algorithm to be read off from the typing rules. It is also comparatively easy to adapt bidirectional approach to more sophisticated typing systems like RankNTypes [4] and dependently typed languages [5] where full type inference can be undecidable.
- There are still much work to do when turning a type system into a bidirectional system. For example, our type system may have **subtyping** like  $\forall a b. a \rightarrow b \rightarrow a \prec \forall c. c \rightarrow c \rightarrow c$ . The user could have annotated less general type than expected.

# Remarks

- Type checking System F (The one with  $\Lambda a. e, e[t]$ ) might involve checking equivalence between types. For example, should the following term be accepted?

$$(\lambda(f:\forall\alpha. \alpha \rightarrow \alpha). f) (\Lambda\beta. \lambda(x:\beta). x)$$

But the issue comes back when we have type constructors anyway.

- Algorithm W (Type inference algorithm for HM System) is kind of global algorithm whereas bidirectional approach propagates type information locally.
- We can extend HM system with data types. This naturally leads to a kind system on types that classifies type constructors.

# References

1. Frank Pfenning. CMU CS15-312 Foundations of Programming Languages  
<http://www.cs.cmu.edu/~fp/courses/15312-f04/index.html>
2. D. Christiansen. Bidirectional Typing Rules: A Tutorial  
<http://www.itu.dk/people/drc/tutorials/bidirectional.pdf>
3. Oleg Kiselyov. How OCaml type checker works -- or what polymorphism and garbage collection have in common. <http://okmij.org/ftp/ML/generalization.html>
4. Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. <http://research.microsoft.com/en-us/um/people/simonpj/papers/higher-rank/>
5. Andres Löb, Conor McBride and Wouter Swierstra. A Tutorial Implementation of a Dependently Typed Lambda Calculus. <http://www.andres-loeh.de/LambdaPi/>

# Language Syntax Quick Guide

In Slide	In Code	Analogy
$x$	VAR "x"	x
$\lambda x. e$ $\lambda(x:t). e$	LAM ("x", e) LAM ("x", t, e) ALAM ("x", t, e)	<pre>\x -&gt; e function (x) { return e; } \x (x :: t) -&gt; e (x : t) =&gt; e [???](t x) { return e; }</pre>
$e_1 e_2$	AP (e1, e2)	<pre>e1 e2 e1(e2)</pre>
<b>let</b> $x = e_1$ <b>in</b> $e_2$	LET ("x", e1, e2)	<pre>let x = e1 in e2 val x = e1; e2</pre>
$e:t$	ANNO (e,t)	$e :: t$
$\Lambda \alpha. e$	TLAM ("a", e)	<pre>def foo[a](...) = ... fn bar&lt;a&gt;(...) { ... } template&lt;typename a&gt; e</pre>
$e [t]$	TAP (e, t)	$e<t>()$

# Bidirectional Typing

- $\Gamma_{\text{in}} \vdash e_{\text{in}} \Rightarrow t_{\text{out}} : e$  can be inferred to have type  $t$
- $\Gamma_{\text{in}} \vdash e_{\text{in}} \Leftarrow t_{\text{in}} : e$  can be checked against type  $t$

$$\frac{(x:t) \in \Gamma}{\Gamma \vdash x \Rightarrow t} \qquad \frac{\Gamma, x:t \vdash e \Rightarrow t'}{\Gamma \vdash (\lambda(x:t).e) \Rightarrow t \rightarrow t'} \qquad \frac{\Gamma, x:t \vdash e \Leftarrow t'}{\Gamma \vdash (\lambda x.e) \Leftarrow t \rightarrow t'}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \alpha \rightarrow \beta \quad \Gamma \vdash e_2 \Leftarrow \alpha}{\Gamma \vdash e_1 e_2 \Rightarrow \beta}$$

$$\frac{\Gamma \vdash e \Leftarrow t}{\Gamma \vdash (e:t) \Rightarrow t} \qquad \frac{\Gamma \vdash e \Rightarrow t' \quad \Gamma \vdash t = t'}{\Gamma \vdash e \Leftarrow t}$$