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 λ-Calculus
  - Hindley-Milner Type System

Language Syntax (Incomplete)

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)	\x -> e function (x) { return e; } \(x :: t) -> e (x : t) => e [???](t x) { return e; }
$e_1\ e_2$	AP (e1, e2)	e1 e2 e1(e2)
$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	LET ("x", e1, e2)	let x = e1 in e2 val x = e1; e2
A	TVAR "A"	a, T (Haskell) typename A, T (roughly)
$t_1 \to t_2$	TARR (t1, t2)	t1 -> t2 function<t2(t1)> t2(*)(t1) (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); }
}
```

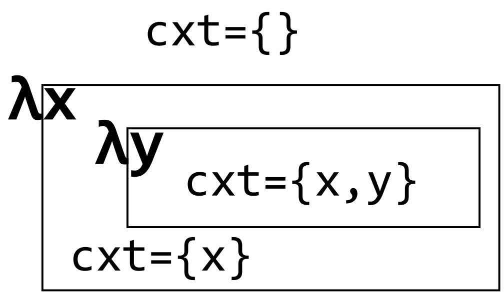
- C++ (roughly):

```
[](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 α, β we will have...
  - $e_1 : \alpha \to \beta$ 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 $e_1 : \mathbf{char} \to \mathbf{int}$, $e_2 : \mathbf{char}$: OK
  - $e_1\, e_2$ with $e_1 : \mathbf{bool} \to \mathbf{string}$, $e_2 : \mathbf{int}$: 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 `cxt`, then for some α, β ...
  - x has type α
  - e has type β 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

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

First Solution

- Sticking to $\lambda(x{:}t).\, e$, i.e. users are require to annotate α
    - 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 $t \to t'$

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 argumetns)?

Improvement via Bidirectional Typing

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

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

Improvement via Bidirectional Typing

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

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

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

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

- Separating type **checking** and **inference**:

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

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 α, regardless whether β is known or not.

- Must **infer** the type of e_1. Then e_2 can be check to have type α. 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. $(\lambda x.\, e) : \alpha \to \beta$, $(\lambda(x{:}\alpha).\, e) : \alpha \to \beta$

- 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

$$\vdash (\lambda(x{:}\,a \to a).\, x)(\lambda y.\, y) \Rightarrow$$

An Bidirectional Typing Example

$$\frac{\vdash \lambda(x{:}\,a \to a).\, x \Rightarrow \qquad \vdash \lambda y.\, y \Leftarrow}{\vdash (\lambda(x{:}\,a \to a).\, x)(\lambda y.\, y) \Rightarrow}$$

An Bidirectional Typing Example

$$\dfrac{\dfrac{x{:}\,a \to a \vdash x \Rightarrow}{\vdash \lambda(x{:}\,a \to a).\, x \Rightarrow} \qquad \vdash \lambda y.\, y \Leftarrow}{\vdash (\lambda(x{:}\,a \to a).\, x)(\lambda y.\, y) \Rightarrow}$$

An Bidirectional Typing Example

$$\dfrac{\dfrac{\overline{x{:}\,a \to a \vdash x \Rightarrow a \to a}}{\vdash \lambda(x{:}\,a \to a).\, x \Rightarrow} \qquad \vdash \lambda y.\, y \Leftarrow}{\vdash (\lambda(x{:}\,a \to a).\, x)(\lambda y.\, y) \Rightarrow}$$

An Bidirectional Typing Example

$$\dfrac{\dfrac{\overline{x{:}\,a \to a \vdash x \Rightarrow a \to a}}{\vdash \lambda(x{:}\,a \to a).\, x \Rightarrow (a \to a) \to a \to a} \qquad \vdash \lambda y.\, y \Leftarrow a \to a}{\vdash (\lambda(x{:}\,a \to a).\, x)(\lambda y.\, y) \Rightarrow a \to a}$$

An Bidirectional Typing Example

$$\dfrac{\dfrac{\dfrac{}{x{:}\,a \to a \vdash x \Rightarrow a \to a}}{\vdash \lambda(x{:}\,a \to a).\, x \Rightarrow (a \to a) \to a \to a} \qquad \dfrac{y{:}\,a \vdash y \Leftarrow a}{\vdash \lambda y.\, y \Leftarrow a \to a}}{\vdash (\lambda(x{:}\,a \to a).\, x)(\lambda y.\, y) \Rightarrow a \to a}$$

An Bidirectional Typing Example

$$\dfrac{\dfrac{\dfrac{}{x{:}\,a \to a \vdash x \Rightarrow a \to a}}{\vdash \lambda(x{:}\,a \to a).\,x \Rightarrow (a \to a) \to a \to a} \qquad \dfrac{\dfrac{\dfrac{}{y{:}\,a \vdash y \Rightarrow a}}{y{:}\,a \vdash y \Leftarrow a}}{\vdash \lambda y.\,y \Leftarrow a \to a}}{\vdash (\lambda(x{:}\,a \to a).\,x)(\lambda y.\,y) \Rightarrow a \to a}$$

An Equational Approach

- The case $e_1\, e_2$

  1. $e_1 : \alpha \to \beta$
  2. $e_2 : \alpha$
  3. $e_1\, e_2 : \beta$

- Does not knowing α really matter?

  - Set up a constraint and solve it latter!
  - Let $e_1 : t$ and $e_2 : t'$
  - $t = t' \to t''$ 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 $\Gamma :\equiv f{:}t_1 ,\ z{:}t_2$.

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

An Equational Approach

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

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

An Equational Approach

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

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

An Equational Approach

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

$$
\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \vdash f : t_1 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\,z : \_}}{\Gamma \vdash f\,z\,z : \_}}{f{:}t_1 \vdash \lambda z.\, f\,z\,z : \_}}{\vdash \lambda f.\, \lambda z.\, f\,z\,z : \_}
$$

An Equational Approach

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : t_1 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\,z : t_3}}{\Gamma \vdash f\,z\,z : \_}}{f{:}t_1 \vdash \lambda z.\, f\,z\,z : \_}}{\vdash \lambda f.\, \lambda z.\, f\,z\,z : \_}$$

$t_1 = t_2 \to t_3$

An Equational Approach

- Write $\Gamma :\equiv f{:}\,t_2 \to t_3\,,\ z{:}\,t_2\,.$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : t_2 \to t_3 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\,z : t_3}}{\Gamma \vdash f\,z\,z : \_}}{f{:}\,t_2 \to t_3 \vdash \lambda z.\, f\,z\,z : \_}}{\vdash \lambda f.\, \lambda z.\, f\,z\,z : \_}$$

$t_1 = t_2 \to t_3$

An Equational Approach

- Write $\Gamma :\equiv f{:}\,t_2 \to t_3\,,\ z{:}\,t_2\,.$

$$\dfrac{\dfrac{\Gamma \vdash f : t_2 \to t_3 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\,z : t_3} \qquad \Gamma \vdash z : t_2}{\dfrac{\Gamma \vdash f\,z\,z : \_}{\dfrac{f{:}\,t_2 \to t_3 \vdash \lambda z.\,f\,z\,z : \_}{\vdash \lambda f.\,\lambda z.\,f\,z\,z : \_}}}$$

An Equational Approach

- Write $\Gamma :\equiv f{:}\,t_2 \to t_3\,,\ z{:}\,t_2\,.$

$$
\dfrac{\dfrac{\Gamma \vdash f : t_2 \to t_3 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\,z : t_3} \qquad \Gamma \vdash z : t_2}{\dfrac{\Gamma \vdash f\,z\,z : t_4}{\dfrac{f{:}\,t_2 \to t_3 \vdash \lambda z.\,f\,z\,z : \_}{\vdash \lambda f.\,\lambda z.\,f\,z\,z : \_}}}
$$

$t_3 = t_2 \to t_4$

An Equational Approach

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

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : t_2 \to t_2 \to t_4 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\,z : t_2 \to t_4} \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\,z\,z : t_4}}{\dfrac{f\colon t_2 \to t_2 \to t_4 \vdash \lambda z.\, f\,z\,z : \_}{\vdash \lambda f.\, \lambda z.\, f\,z\,z : \_}}$$

$t_3 = t_2 \to t_4$

An Equational Approach

- Write $\Gamma :\equiv f{:}\, t_2 \to t_2 \to t_4,\ z{:}\, t_2$.

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash f : t_2 \to t_2 \to t_4 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\, z : t_2 \to t_4} \qquad \Gamma \vdash z : t_2}{\cfrac{\Gamma \vdash f\, z\, z : t_4}{f{:}\, t_2 \to t_2 \to t_4 \vdash \lambda z.\, f\, z\, z : t_2 \to t_4}}}{\vdash \lambda f.\, \lambda z.\, f\, z\, z : \_}$$

An Equational Approach

- Write $\Gamma :\equiv f{:}\, t_2 \to t_2 \to t_4 ,\ z{:}\, t_2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : t_2 \to t_2 \to t_4 \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\, z : t_2 \to t_4} \qquad \Gamma \vdash z : t_2}{\Gamma \vdash f\, z\, z : t_4}}{f{:}\, t_2 \to t_2 \to t_4 \vdash \lambda z.\, f\, z\, z : t_2 \to t_4}}{\vdash \lambda f.\, \lambda z.\, f\, z\, z : (t_2 \to t_2 \to t_4) \to t_2 \to t_4}$$

An Equational Approach

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

- Allocate fresh unknown t_1

- Case $\lambda x.\, e$ with $x :\equiv f$, $e :\equiv \lambda z.\, f z z$
  - $x : \alpha$
  - $e : \beta$
  - $(\lambda x.\, e) : \alpha \to \beta$
- Context: $[]$

Now $f : t_1$

An Equational Approach

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

- Allocate fresh unknown t_2

- Case $\lambda x.\, e$ with $x :\equiv z$, $e :\equiv f z z$
  - $x : \alpha$
  - $e : \beta$
  - $(\lambda x.\, e) : \alpha \to \beta$
- Context: $[(f\!:\!t_1)]$

Now $f : t_1$ $z : t_2$

An Equational Approach

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

- Recursion.

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

Now $f : t_1$ $\quad z : t_2$

An Equational Approach

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

- Recursion.

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

Now $f : t_1 \qquad z : t_2$

An Equational Approach

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

- Lookup context and return t_1.

- Case x with $x :\equiv f$
  - $x : \alpha$
  - $(x\!:\alpha) \in \mathtt{cxt}$

- 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 $x :\equiv z$
  - $x : \alpha$
  - $(x{:}\,\alpha) \in \texttt{cxt}$

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

Now $f : t_1$ $\quad z : t_2$

An Equational Approach

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

- Allocate fresh unknown t_3. Return t_3.

$t_1 = t_2 \to t_3$

- Case $e_1\, e_2$ with $e_1 :\equiv f$, $e_2 :\equiv z$
  - $e_1 : \alpha \to \beta$
  - $e_2 : \alpha$
  - $e_1\, e_2 : \beta$
- Context: $[(f{:}\, t_1), (z{:}\, t_2)]$

Now $f : t_1$ $\quad z : t_2$

An Equational Approach

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

- Lookup context and return t_2.

$t_1 = t_2 \to t_3$

- Case x with $x :\equiv z$
  - $x : \alpha$
  - $(x{:}\alpha) \in \texttt{cxt}$

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

Now $f : t_1$ $\quad z : t_2$

An Equational Approach

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

- Allocate fresh unknown t_4. Return t_4.

$$t_1 = t_2 \to t_3$$
$$t_3 = t_2 \to t_4$$

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

Now $f : t_1$ $\qquad z : t_2$ $\qquad f z : t_3$

An Equational Approach

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

- Return $t_2 \to t_4$

$$t_1 = t_2 \to t_3$$
$$t_3 = t_2 \to t_4$$

- Case $\lambda x.\, e$ with $x :\equiv z$, $e :\equiv f z z$
  - $x : \alpha$
  - $e : \beta$
  - $(\lambda x.\, e) : \alpha \to \beta$
- Context: $[(f : t_1)]$

Now $f : t_1$ $\quad z : t_2$ $\quad f z : t_3$ $\quad f z z : t_4$

An Equational Approach

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

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

$$t_1 = t_2 \to t_3$$
$$t_3 = t_2 \to t_4$$

- Case $\lambda x.\, e$ with $x :\equiv f$, $e :\equiv \lambda z.\, f z z$
  - $x : \alpha$
  - $e : \beta$
  - $(\lambda x.\, e) : \alpha \to \beta$
- Context: $[]$

Now $f : t_1$ $\quad z : t_2$ $\quad f z : t_3$ $\quad f z z : t_4$ $\quad (\lambda z.\, f z z) : t_2 \to t_4$

Solving Equations

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

$$t_1 = t_2 \to t_3$$
$$t_3 = t_2 \to t_4$$

Solving Equations

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

- $(f : t_2 \to t_3)$, $(z : t_2)$, $(fz : t_3)$, $(fzz : t_4)$, $(\lambda z.\, fzz) : t_2 \to t_4$

$$
\begin{aligned}
t_1 &= t_2 \to t_3 \\
t_3 &= t_2 \to t_4
\end{aligned}
$$

- Substitute $t_2 \to t_3$ for t_1!

Solving Equations

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

- $(f : t_2 \to t_2 \to t_4)$, $(z : t_2)$, $(fz : t_2 \to t_4)$, $(fzz : t_4)$, $(\lambda z.\, f z z) : t_2 \to t_4$

  $$t_1 = t_2 \to t_3$$
  $$t_3 = t_2 \to t_4$$

- Substitute $t_2 \to t_4$ for t_3!

Solving Equations

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

- $(f : t_2 \to t_2 \to t_4)$, $(z : t_2)$, $(fz : t_2 \to t_4)$, $(fzz : t_4)$, $(\lambda z.\, fzz) : t_2 \to t_4$

  $$t_1 = t_2 \to t_3$$
  $$t_3 = t_2 \to t_4$$

- No more t_1 and t_3. Done!

  $$\boxed{(\lambda f.\, \lambda z.\, fzz) : (A \to A \to B) \to A \to B}$$

- There might be equations of form $T_1 \to T_2 = T_3 \to T_4$. 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 integraed 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 $= (\_ \to \_), \ldots$) $\Rightarrow$ substitute RHS for t_i (*)
- $T_1 \to T_2 = T_3 \to T_4$ $\Rightarrow$ 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 $e_1\, e_2$

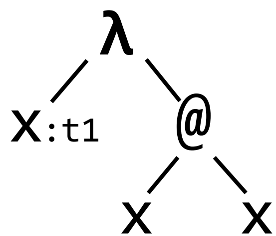
```
function typeinfer(cxt, AP (e1, e2)):
  let t1 = typeinfer(cxt, e1)
  let t2 = typeinfer(cxt, e2)
  let t = fresh_var()
  unify(t1, TARR (t2, t))
  return t
```

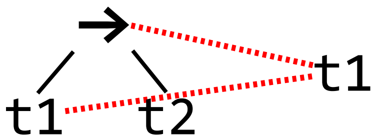
- Case $\lambda x.\, e$

```
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 $t_1 = t_1 \to t_2$
  - Cannot eliminate t_1 by substituting $t_1 \to t_2$ 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:

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

Polymorphism: Girard's System F

In Slide	In Code	Analogy

$\Lambda\alpha.e$	`TLAM ("a", e)`	`def foo[a](...) = ...`
		`fn bar<a>(...) { ... }`
		`template<typename a> e`

$e\ [t]$	`TAP (e, t)`	`e<t>()`
A	`TVAR "A"`	`a, T (Haskell)` `typename A, T     (roughly)`

$t_1 \to t_2$	`TARR (t1, t2)`	`t1 -> t2`
		`function<t2(t1)>`
		`t2(*)(t1)         (roughly)`

$\forall a. t$	`TALL (a, t)`	`forall a. t`
		`{a : Set l} → t`
		`∀{a} → t`

Examples

- Slide & Code: $\Lambda a.\, \Lambda b.\, \lambda(x{:}\,a).\, \lambda(f{:}\,a \to b).\, f\,x$

```
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

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

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

$$\rightsquigarrow \Lambda T.\, \Lambda S.$$
$$(\lambda(f{:}\, T \to \boxed{S} \to \boxed{S}).\, \lambda(z{:}\, T).\, f\, z)(\lambda(x{:}\, T).\, \lambda(y{:}\, S).\, y)$$

$$\rightsquigarrow \Lambda T.\, \Lambda S.$$
$$\lambda(z{:}\, T)\ \boxed{(\lambda(x{:}\, T).\, \lambda(y{:}\, S).\, y)}\ z$$

$$\rightsquigarrow \Lambda T.\, \Lambda S.$$
$$\lambda(z{:}\, T) \lambda(y{:}\, S).\, y$$

Damas-Hindley-Milner System

- Generalize at `let`. $\forall$-quantifiers are only allowed at "top" level
  - $\forall a. \forall b. a \to b \to b$: OK
  - $\forall a. \forall b. b \to (a \to b) \to b$: OK
  - $(\forall a. a \to a) \to \texttt{Int}$: 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!)

$\textbf{let}\ id = \lambda x. x\ \textbf{in}$ $\Leftarrow$ Generalize at this point
$id\ id\ id$

Damas-Hindley-Milner System

$\lambda u.$
$(\textbf{let}\ id = \boxed{\lambda x.x}\ \textbf{in}$
$id\ id\ id)$

- Inferred to be $t_1 \to t_1$ 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 $\lambda(u : t_3).\ \ldots$ should not be generalized

Damas-Hindley-Milner System

$\lambda u.$
$(\textbf{let} \boxed{id} = \lambda x.\, x\ \textbf{in}$
$id\ id\ id)$

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

Damas-Hindley-Milner System

$$\lambda u.$$
$$(\textbf{let } id = \Lambda t_1 . \lambda(x{:}t_1).\, x \textbf{ in}$$
$$\boxed{id[(t_2 \to t_2) \to t_2 \to t_2]\; id[t_2 \to t_2]\; id[t_2]}\;)$$

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

- Case $\mathbf{let}$

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

  - Quantify: calculate free variables `a`s of `t1` which are unbound in `cxt`, i.e. $\overline{a} :\equiv FV(t_1) \setminus \left(\bigcup_{t \in \texttt{cxt}} FV(t)\right)$.

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 $\forall a\, b.\, a \to b \to a \prec \forall c.\, c \to c \to c$. 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?

  $$(\lambda(f{:}\forall\alpha.\,\alpha\to\alpha).\; f)\,(\Lambda\beta.\,\lambda(x{:}\beta).\; x)$$

  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 Löh, 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)`
		`\x -> e`
		`function (x) { return e; }`
		`\(x :: t) -> e`
		`(x : t) => e`
		`[???](t x) { return e; }`

$e_1\ e_2$	`AP (e1, e2)`	`e1 e2` `e1(e2)`
let $x = e_1$ **in** e_2	`LET ("x", e1, e2)`	`let x = e1 in e2` `val x = e1; e2`
$e{:}t$	`ANNO (e,t)`	`e :: t`

$\Lambda\alpha.e$	`TLAM ("a", e)`	`def foo[a](...) = ...`
		`fn bar<a>(...) { ... }`
		`template<typename a> e`

$e\ [t]$	`TAP (e, t)`	`e<t>()`

Bidirectional Typing

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

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

$$\frac{\Gamma \vdash e_1 \Rightarrow \alpha \to \beta \qquad \Gamma \vdash e_2 \Leftarrow \alpha}{\Gamma \vdash e_1\, e_2 \Rightarrow \beta}$$

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