Last time: $\lambda^{\rightarrow}$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}\;\textsf{tvar}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda \mathsf{x}{:}A.M : A \to B}\;\to\textsf{-intro}
\qquad
\frac{\begin{array}{c}\Gamma \vdash M : A \to B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B}\;\to\textsf{-elim}$$

Last time: System F

$$\frac{\Gamma, \alpha{::}K \vdash M : A}{\Gamma \vdash \Lambda\alpha{::}K.M : \forall\alpha{::}K.A}\ \forall\text{-intro}$$

$$\frac{\Gamma \vdash M : \forall\alpha{::}K.A \qquad \Gamma \vdash B :: K}{\Gamma \vdash M\ [B] : A[\alpha{::}{=}B]}\ \forall\text{-elim}$$

This time

$$\Gamma \vdash M : \ ?$$

What is type inference?

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

What is type inference?

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

$$\lambda \mathsf{f}{:}A \to B.\lambda \mathsf{g}{:}C \to A.\lambda \mathsf{x}{:}C.\mathsf{f}\ (\mathsf{g}\ \mathsf{x})$$

What is type inference?

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

$$\lambda \mathsf{f}{:}A \to B.\lambda \mathsf{g}{:}C \to A.\lambda \mathsf{x}{:}C.\mathsf{f}\ (\mathsf{g}\ \mathsf{x})$$

$$\lambda \mathsf{f}.\lambda \mathsf{g}.\lambda \mathsf{x}.\mathsf{f}\ (\mathsf{g}\ \mathsf{x})$$

What is type inference?

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

$$\lambda \mathsf{f}{:}A \to B.\lambda \mathsf{g}{:}C \to A.\lambda \mathsf{x}{:}C.\mathsf{f}\;(\mathsf{g}\;\mathsf{x})$$

$$\lambda \mathsf{f}.\lambda \mathsf{g}.\lambda \mathsf{x}.\mathsf{f}\;(\mathsf{g}\;\mathsf{x})$$

$$:(A \to B) \to (C \to A) \to C \to B$$

Last time: $\lambda^{\to}$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}\ \textsf{tvar}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda \mathsf{x}{:}A.M : A \to B}\ \to\textsf{-intro} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \to B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B}\ \to\textsf{-elim}$$

This time: $\lambda^{\to}$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}\ \textsf{tvar}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda \mathsf{x}.M : A \to B}\ {\to}\textsf{-intro} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \to B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B}\ {\to}\textsf{-elim}$$

$\lambda^{\rightarrow}$ in Prolog

```
ty(G,v(X),A) :- in((X,A),G).
```

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}\ \text{tvar}$$

```
ty(G,l(X,M),to(A,B)) :- ty([(X,A)|G],M,B).
```

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \ \lambda \text{x}.M \ : A \rightarrow B}\ \rightarrow\text{-intro}$$

```
ty(G,a(M,N),Res) :- ty(G,M,Fun), ty(G,N,Arg),
  unify_with_occurs_check(Fun,to(Arg,Res)).
```

$$\frac{\begin{array}{c}\Gamma \vdash M : A \rightarrow B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B}\ \rightarrow\text{-elim}$$

$\lambda^{\rightarrow}$ in Prolog

```
in((X,A),G) :- member((X,A),G), !.

ty(G,v(X),A) :- in((X,A),G).
ty(G,l(X,M),to(A,B)) :- ty([(X,A)|G],M,B).
ty(G,a(M,N),Res) :- ty(G,M,Fun), ty(G,N,Arg),
  unify_with_occurs_check(Fun,to(Arg,Res)).

/* · ⊢ λf.λg.λx.f (g x) : (A → B) → (C → A) → C → B */
> ty([],l(f,l(g,l(x,a(v(f),a(v(g),v(x)))))),T).
T = to(to(A,B),to(to(C,A),to(C,B)))

/* · ⊢ λx.λx.x : _ → A → A */
> ty([],l(x,l(x,v(x))),T).
T = to(_,to(A,A)) /*

/* λx.(x x) -- self-application does not type check.*/
> ty([],l(x,a(v(x),v(x))),T).
no
```

What is type inference?

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

What is type inference?

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Goal

succinctness of annotation-free code

+

safety and expressiveness of System F

What is type inference?

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Goal

succinctness of annotation-free code

$+$

safety and expressiveness of System F

Bad news

the goal is unachievable

The ML calculus

The ML calculus

$$\begin{tikzcd}
 & F\omega \\
F \arrow[ur] & \\
\mathsf{ML} \arrow[u] & \\
\lambda^{\rightarrow} \arrow[u] &
\end{tikzcd}$$

Prenex quantifification

$$\forall\alpha.\alpha \to \alpha$$

$$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$$

$$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$$

$$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$$

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

```
let id x = x
 in id id
```

```
let f id = id id
 in f (fun x -> x)
```

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

```
let id x = x
 in id id
```

```
let f id = id id
 in f (fun x -> x)
```

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$ ✓

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

```
let id x = x
 in id id
```

```
let f id = id id
 in f (fun x -> x)
```

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$ ✓

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$ ✗

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

```
let id x = x
 in id id
```

```
let f id = id id
 in f (fun x -> x)
```

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$ ✓

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$ ✗

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$ ✗

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

```
let id x = x
 in id id
```

```
let f id = id id
 in f (fun x -> x)
```

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$ ✓

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$ ✗

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$ ✗

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

✓

```
let id x = x
 in id id
```

```
let f id = id id
 in f (fun x -> x)
```

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$ ✓

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$ ✗

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$ ✗

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

✓

```
let id x = x
 in id id
```

✓

```
let f id = id id
 in f (fun x -> x)
```

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$ ✓

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$ ✗

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$ ✗

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

✓

```
let id x = x
 in id id
```

✓

```
let f id = id id
 in f (fun x -> x)
```

✗

```
(fun id -> id id)
  (fun x -> x)
```

Prenex quantifification

$\forall\alpha.\alpha \to \alpha$ ✓

$\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta)$ ✓

$\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha$ ✗

$\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta)$ ✗

Let-bound polymorphism

```
let id = fun x -> x
 in id id
```

✓

```
let id x = x
 in id id
```

✓

```
let f id = id id
 in f (fun x -> x)
```

✗

```
(fun id -> id id)
  (fun x -> x)
```

✗

Types and schemes

$$\frac{}{\Gamma \vdash \mathcal{B} \text{ is a type}}\ \mathcal{B}\text{-types}$$

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \text{ is a type}}\ \alpha\text{-types}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \text{ is a type} \\ \Gamma \vdash B \text{ is a type}\end{array}}{\Gamma \vdash A \to B \text{ is a type}}\ \to\text{-types}$$

$$\frac{\Gamma, \overline{\alpha} \vdash A \text{ is a type}}{\Gamma \vdash \forall \overline{\alpha}.A \text{ is a scheme}}\ \text{scheme}$$

Environments

$$\frac{}{\cdot \text{ is an environment}}\ \Gamma\text{-}\cdot$$

$$\frac{\begin{array}{c}\Gamma \text{ is an environment} \\ \Gamma \vdash S \text{ is a scheme}\end{array}}{\Gamma,\ x{:}S \text{ is an environment}}\ \Gamma\text{-}{:}$$

Typing rules for →

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \to\text{-intro}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \to\text{-elim}$$

Typing rules for schemes

$$\frac{\Gamma \vdash M : A \qquad \overline{\alpha} \cap ftv(\Gamma) = \emptyset \qquad \Gamma, x : \forall\overline{\alpha}.A \vdash N : B}{\Gamma \vdash \textsf{let x} = M \textsf{ in } N : B}\ \textsf{scheme-intro}$$

$$\frac{\begin{array}{c} x : \forall\overline{\alpha}.A \in \Gamma \\ \Gamma \vdash B \textsf{ is a type} \qquad (\textsf{for } B \in \overline{B}) \end{array}}{\Gamma \vdash x : A[\overline{\alpha} := \overline{B}]}\ \textsf{scheme-elim}$$

Milner’s algorithm

Substitutions

$$[\, a_1 \;\mapsto\; A_1\,,\;\; a_2 \;\mapsto\; A_2\,,\;\; \ldots\;\; a_n \;\mapsto\; A_n \,]$$

For example, let

σ be $[a \mapsto \mathcal{B}$, $b \mapsto (\mathcal{B} \to \mathcal{B})]$

A be $a \to b \to a$

Then

σA is $\mathcal{B} \to (\mathcal{B} \to \mathcal{B}) \to \mathcal{B}$.

If

σA = B (for some σ)

then we say

B is a *substitution instance* of A.

Constraints

$$a = b$$

$$a \to b = \mathcal{B} \to b$$

$$\mathcal{B} = \mathcal{B}$$

$$\mathcal{B} = \mathcal{B} \to \mathcal{B}$$

Unification

$$\text{unify} : \text{ConstraintSet} \rightarrow \text{Substitution}$$

$$\begin{aligned}
\text{unify}(\emptyset) &= [] \\
\text{unify}(\{A = A\} \cup C) &= \text{unify}(C) \\
\text{unify}(\{a = A\} \cup C) &= \text{unify}([a \mapsto A]C) \circ [a \mapsto A] \\
&\quad \text{when } a \notin ftv(A) \\
\text{unify}(\{A = a\} \cup C) &= \text{unify}([a \mapsto A]C) \circ [a \mapsto A] \\
&\quad \text{when } a \notin ftv(A) \\
\text{unify}(\{A \rightarrow B = A' \rightarrow B'\} \cup C) &= \text{unify}(\{A = A', B = B'\} \cup C) \\
\text{unify}(\{A = B\} \cup C) &= FAIL
\end{aligned}$$

Algorithm J

$$\mathsf{J} \ : \ \mathsf{Environment} \times \mathsf{Expression} \ \to \ \mathsf{Type}$$

$\mathsf{J}\ (\Gamma,\ \lambda\mathsf{x}.\mathsf{M}) = b \to A$
where $\mathsf{A} = \mathsf{J}\ (\Gamma, \mathsf{x}:b,\ \mathsf{M})$
and b *is fresh*

$\mathsf{J}\ (\Gamma,\ \mathsf{M}\ \mathsf{N}) = b$
where $A = \mathsf{J}\ (\Gamma,\ \mathsf{M})$
and $B = \mathsf{J}\ (\Gamma,\ \mathsf{N})$
and $\mathsf{unify'}\ (\{A = B \to b\})$
succeeds
and b *is fresh*

$\mathsf{J}\ (\Gamma,\ \mathsf{x}) = \mathsf{A}[\overline{\alpha}{:=}\bar{b}]$
where $\Gamma(\mathsf{x}) = \forall\overline{\alpha}.A$
and $\bar{b}$ *are fresh*

$\mathsf{J}\ (\Gamma,\ \mathsf{let}\ \mathsf{x} = \mathsf{M}\ \mathsf{in}\ \mathsf{N}) = B$
where $A = \mathsf{J}\ (\Gamma,\ \mathsf{M})$
and $B = \mathsf{J}\ (\Gamma, \mathsf{x}:\forall\overline{\alpha}.A,\ \mathsf{N})$
and $\overline{\alpha} = \mathsf{ftv}(A) \setminus \mathsf{ftv}(\Gamma)$

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
      let id = λy.y in
       apply id) =
```

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
       let id = λy.y in
         apply id) =
    J(·, λf.λx.f x) =
```

Algorithm J in action

$\mathtt{J}(\cdot,\ \mathbf{let}\ \mathtt{apply} = \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ \mathtt{x}\ \mathbf{in}$
$\quad\quad \mathbf{let}\ \mathtt{id} = \lambda\mathtt{y}.\mathtt{y}\ \mathbf{in}$
$\quad\quad\quad \mathtt{apply}\ \mathtt{id}) =$
$\quad \mathtt{J}(\cdot,\ \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ \mathtt{x}) =$
$\quad\quad \mathtt{J}(\cdot,\mathtt{f}{:}b_1,\ \lambda\mathtt{x}.\mathtt{f}\ \mathtt{x}) =$

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
      let id = λy.y in
       apply id) =
   J(·, λf.λx.f x) = $b_1 \to b_2 \to b_3$
     J(·,f:$b_1$, λx.f x) = $b_2 \to b_3$
       J(·,f:$b_1$,x:$b_2$, f x) = $b_3$
```

Algorithm J in action

```
J($\cdot$, let apply = $\lambda$f.$\lambda$x.f x in
     let id = $\lambda$y.y in
      apply id) =
  J($\cdot$, $\lambda$f.$\lambda$x.f x) = $b_1 \to b_2 \to b_3$
    J($\cdot$,f:$b_1$, $\lambda$x.f x) = $b_2 \to b_3$
      J($\cdot$,f:$b_1$,x:$b_2$, f x) = $b_3$
        J($\cdot$,f:$b_1$,x:$b_2$, f) =
```

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
     let id = λy.y in
       apply id) =
  J(·, λf.λx.f x) = $b_1 \to b_2 \to b_3$
    J(·,f:$b_1$, λx.f x) = $b_2 \to b_3$
      J(·,f:$b_1$,x:$b_2$, f x) = $b_3$
        J(·,f:$b_1$,x:$b_2$, f) = $b_1$
```

Algorithm J in action

```
J(·, let apply = $\lambda$f.$\lambda$x.f x in
      let id = $\lambda$y.y in
       apply id) =
  J(·, $\lambda$f.$\lambda$x.f x) = $b_1 \to b_2 \to b_3$
    J(·,f:$b_1$, $\lambda$x.f x) = $b_2 \to b_3$
      J(·,f:$b_1$,x:$b_2$, f x) = $b_3$
        J(·,f:$b_1$,x:$b_2$, f) = $b_1$
        J(·,f:$b_1$,x:$b_2$, x) =
```

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
     let id = λy.y in
       apply id) =
  J(·, λf.λx.f x) = $b_1 \to b_2 \to b_3$
    J(·,f:$b_1$, λx.f x) = $b_2 \to b_3$
      J(·,f:$b_1$,x:$b_2$, f x) = $b_3$
        J(·,f:$b_1$,x:$b_2$, f) = $b_1$
        J(·,f:$b_1$,x:$b_2$, x) = $b_2$
```

Algorithm J in action

```
J(·, let apply = $\lambda$f.$\lambda$x.f x in
      let id = $\lambda$y.y in
        apply id) =
   J(·, $\lambda$f.$\lambda$x.f x) = $b_1 \to b_2 \to b_3$
     J(·,f:$b_1$, $\lambda$x.f x) = $b_2 \to b_3$
       J(·,f:$b_1$,x:$b_2$, f x) = $b_3$
         J(·,f:$b_1$,x:$b_2$, f) = $b_1$
         J(·,f:$b_1$,x:$b_2$, x) = $b_2$
         unify({$b_1$ = $b_2 \to b_3$}) =
```

Algorithm J in action

$$
\begin{array}{l}
\texttt{J}(\cdot,\ \textbf{let}\ \texttt{apply} = \lambda\texttt{f}.\lambda\texttt{x}.\texttt{f x}\ \textbf{in} \\
\quad\quad \textbf{let}\ \texttt{id} = \lambda\texttt{y}.\texttt{y}\ \textbf{in} \\
\quad\quad\quad \texttt{apply id}) = \\
\quad \texttt{J}(\cdot,\ \lambda\texttt{f}.\lambda\texttt{x}.\texttt{f x}) = b_1 \to b_2 \to b_3 \\
\quad\quad \texttt{J}(\cdot,\texttt{f}{:}b_1,\ \lambda\texttt{x}.\texttt{f x}) = b_2 \to b_3 \\
\quad\quad\quad \texttt{J}(\cdot,\texttt{f}{:}b_1,\texttt{x}{:}b_2,\ \texttt{f x}) = b_3 \\
\quad\quad\quad\quad \texttt{J}(\cdot,\texttt{f}{:}b_1,\texttt{x}{:}b_2,\ \texttt{f}) = b_1 \\
\quad\quad\quad\quad \texttt{J}(\cdot,\texttt{f}{:}b_1,\texttt{x}{:}b_2,\ \texttt{x}) = b_2 \\
\quad\quad\quad\quad \texttt{unify}(\{b_1 = b_2 \to b_3\}) = \{b_1 \mapsto b_2 \to b_3\}
\end{array}
$$

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
     let id = λy.y in
      apply id) =
  J(·, λf.λx.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
    J(·,f:$b_2 \to b_3$, λx.f x) = $b_2 \to b_3$
      J(·,f:$b_2 \to b_3$,x:$b_2$, f x) = $b_3$
        J(·,f:$b_2 \to b_3$,x:$b_2$, f) = $b_2 \to b_3$
        J(·,f:$b_2 \to b_3$,x:$b_2$, x) = $b_2$
        unify({$b_1$ = $b_2 \to b_3$}) = {$b_1 \mapsto b_2 \to b_3$}
```

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
      let id = λy.y in
        apply id) =
   J(·, λf.λx.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
     J(·,f:$b_2 \to b_3$, λx.f x) = $b_2 \to b_3$
       J(·,f:$b_2 \to b_3$,x:$b_2$, f x) = $b_3$
         J(·,f:$b_2 \to b_3$,x:$b_2$, f) = $b_2 \to b_3$
         J(·,f:$b_2 \to b_3$,x:$b_2$, x) = $b_2$
   ftv(($b_2 \to b_3$)$\to b_2 \to b_3$) = {$b_2$, $b_3$}
   ftv(·) = {}
   {$b_2$, $b_3$} \ {} = {$b_2$, $b_3$}
```

Algorithm J in action

J($\cdot$, **let** apply = λf.λx.f x **in**
let id = λy.y **in**
apply id) =
J($\cdot$, λf.λx.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
let id = λy.y **in** apply id) =

Algorithm J in action

```
J($\cdot$, let apply = $\lambda$f.$\lambda$x.f x in
     let id = $\lambda$y.y in
       apply id) =
  J($\cdot$, $\lambda$f.$\lambda$x.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
  J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
    let id = $\lambda$y.y in apply id) =
    J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
      $\lambda$y.y) =
```

Algorithm J in action

```
J($\cdot$, let apply = $\lambda$f.$\lambda$x.f x in
     let id = $\lambda$y.y in
       apply id) =
  J($\cdot$, $\lambda$f.$\lambda$x.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
  J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
     let id = $\lambda$y.y in apply id) =
     J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
       $\lambda$y.y) = $b_4 \to b_4$
       J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,y:$b_4$, y)
         = $b_4$
```

Algorithm J in action

```
J($\cdot$, let apply = $\lambda$f.$\lambda$x.f x in
      let id = $\lambda$y.y in
        apply id) =
   J($\cdot$, $\lambda$f.$\lambda$x.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
   J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
     let id = $\lambda$y.y in apply id) =
     J($\cdot$,apply:$\forall\alpha_2\alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
       $\lambda$y.y) = $b_4 \to b_4$
     ftv($b_4 \to b_4$) = {$b_4$}
     ftv($\cdot$,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$) = {}
     {$b_4$} \ {} = {$b_4$}
```

Algorithm J in action

```
J($\cdot$, let apply = $\lambda$f.$\lambda$x.f x in
      let id = $\lambda$y.y in
       apply id) =
   J($\cdot$, $\lambda$f.$\lambda$x.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
   J($\cdot$,apply:$\forall \alpha_2 \alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
     let id = $\lambda$y.y in apply id) =
     J($\cdot$,apply:$\forall \alpha_2 \alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,
       $\lambda$y.y) = $b_4 \to b_4$
     J($\cdot$, apply:$\forall \alpha_2 \alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,id:$\forall \alpha_4.\alpha_4 \to \alpha_4$,
       apply id) = $b_5$
```

Algorithm J in action

```
J(·, let apply = $\lambda$f.$\lambda$x.f x in
      let id = $\lambda$y.y in
        apply id) =
    J(·, $\lambda$f.$\lambda$x.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
    J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
      let id = $\lambda$y.y in apply id) =
      J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
        $\lambda$y.y) = $b_4 \to b_4$
      J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,id:$\forall\alpha_4.\alpha_4 \to \alpha_4$,
        apply id) = $b_5$
        J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
             id:$\forall\alpha_4.\alpha_4 \to \alpha_4$, apply)
          = $(b_6 \to b_7) \to b_6 \to b_7$
```

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
      let id = λy.y in
        apply id) =
  J(·, λf.λx.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
  J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
    let id = λy.y in apply id) =
    J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
      λy.y) = $b_4 \to b_4$
    J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,id:$\forall\alpha_4.\alpha_4 \to \alpha_4$,
      apply id) = $b_5$
      J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
           id:$\forall\alpha_4.\alpha_4 \to \alpha_4$, apply)
        = $(b_6 \to b_7) \to b_6 \to b_7$
      J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
           id:$\forall\alpha_4.\alpha_4 \to \alpha_4$, id)
        = $b_8 \to b_8$
```

Algorithm J in action

`unify` $(\{(b_6 \to b_7) \to b_6 \to b_7 = (b_8 \to b_8) \to b_5\})$

Algorithm J in action

```
  unify ({(b_6 → b_7) → b_6 → b_7 = (b_8 → b_8) → b_5})
= unify ({b_6 → b_7 = b_8 → b_8,
          b_6 → b_7 = b_5})
```

$$
\begin{aligned}
&\texttt{unify}\ (\{(b_6 \to b_7) \to b_6 \to b_7 = (b_8 \to b_8) \to b_5\}) \\
= {}&\texttt{unify}\ (\{b_6 \to b_7 = b_8 \to b_8, \\
&\qquad\qquad b_6 \to b_7 = b_5\})
\end{aligned}
$$

Algorithm J in action

$$
\begin{array}{ll}
 & \texttt{unify}\ (\{(b_6 \to b_7) \to b_6 \to b_7\ \texttt{=}\ (b_8 \to b_8)\ \to\ b_5\}) \\
\texttt{=} & \texttt{unify}\ (\{b_6 \to b_7\ \texttt{=}\ b_8 \to b_8, \\
 & \qquad\quad b_6 \to b_7\ \texttt{=}\ b_5\}) \\
\texttt{=} & \texttt{unify}\ (\{b_6\ \texttt{=}\ b_8, \\
 & \qquad\quad b_7\ \texttt{=}\ b_8, \\
 & \qquad\quad b_6 \to b_7\ \texttt{=}\ b_5\})
\end{array}
$$

Algorithm J in action

$$\begin{aligned}
& \texttt{unify}\ (\{(b_6 \to b_7) \to b_6 \to b_7\ \texttt{=}\ (b_8 \to b_8)\ \to\ b_5\}) \\
\texttt{=}\ & \texttt{unify}\ (\{b_6 \to b_7\ \texttt{=}\ b_8 \to b_8\texttt{,} \\
& \qquad\qquad\ \ b_6 \to b_7\ \texttt{=}\ b_5\}) \\
\texttt{=}\ & \texttt{unify}\ (\{b_6\ \texttt{=}\ b_8\texttt{,} \\
& \qquad\qquad\ \ b_7\ \texttt{=}\ b_8\texttt{,} \\
& \qquad\qquad\ \ b_6 \to b_7\ \texttt{=}\ b_5\}) \\
\texttt{=}\ & \{b_6\ \mapsto\ b_8\texttt{,}\ b_7\ \mapsto\ b_8\texttt{,}\ b_5\ \mapsto\ b_6 \to b_7\}
\end{aligned}$$

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
      let id = λy.y in
        apply id) =
  J(·, λf.λx.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
  J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
     let id = λy.y in apply id) =
     J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
       λy.y) = $b_4 \to b_4$
     J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,id:$\forall\alpha_4.\alpha_4 \to \alpha_4$,
       apply id) = $b_5$
       J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
            id:$\forall\alpha_4.\alpha_4 \to \alpha_4$, apply)
         = $(b_6 \to b_7) \to b_6 \to b_7$
       J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
            id:$\forall\alpha_4.\alpha_4 \to \alpha_4$, id)
         = $b_8 \to b_8$
```

Algorithm J in action

```
J($\cdot$, let apply = $\lambda$f.$\lambda$x.f x in
     let id = $\lambda$y.y in
      apply id) =
  J($\cdot$, $\lambda$f.$\lambda$x.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
  J($\cdot$,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
    let id = $\lambda$y.y in apply id) =
    J($\cdot$,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
      $\lambda$y.y) = $b_4 \to b_4$
    J($\cdot$, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,id:$\forall\alpha_4.\alpha_4 \to \alpha_4$,
      apply id) = $b_8 \to b_8$
      J($\cdot$, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
           id:$\forall\alpha_4.\alpha_4 \to \alpha_4$, apply)
        = $(b_8 \to b_8) \to b_8 \to b_8$
      J($\cdot$, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
           id:$\forall\alpha_4.\alpha_4 \to \alpha_4$, id)
        = $b_8 \to b_8$
```

Algorithm J in action

```
J(·, let apply = λf.λx.f x in
      let id = λy.y in
       apply id) =
  J(·, λf.λx.f x) = $(b_2 \to b_3) \to b_2 \to b_3$
  J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
    let id = λy.y in apply id) =
    J(·,apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,
      λy.y) = $b_4 \to b_4$
    J(·, apply:$\forall\alpha_2\alpha_3.(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,id:$\forall\alpha_4.\alpha_4 \to \alpha_4$,
      apply id) = $b_8 \to b_8$
```

Algorithm J in action

J($\cdot$, **let** apply = λf.λx.f x **in**

let id = λy.y **in**

apply id) =

J($\cdot$, λf.λx.f x) = $(b_2 \to b_3) \to b_2 \to b_3$

J($\cdot$,apply:$\forall \alpha_2 \alpha_3$.($\alpha_2 \to \alpha_3$)$\to \alpha_2 \to \alpha_3$,

let id = λy.y **in** apply id) = $b_8 \to b_8$

J($\cdot$, apply:$\forall \alpha_2 \alpha_3 .(\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_3$,id:$\forall \alpha_4 . \alpha_4 \to \alpha_4$,

apply id) = $b_8 \to b_8$

Algorithm J in action

J($\cdot$, **let** `apply = `λ`f.`λ`x.f x` **in**
let `id = `λ`y.y` **in**
`apply id) = `$b_8 \to b_8$

Type inference in practice

Type inference and recursion

$$\frac{\Gamma, x{:}A \vdash M : A \qquad \overline{\alpha} \notin ftv(\Gamma) \qquad \Gamma, x : \forall\overline{\alpha}.A \vdash N : B}{\Gamma \vdash \text{let rec x = M in N} : B}\ \text{let-rec}$$

Supporting imperative programming: the value restriction

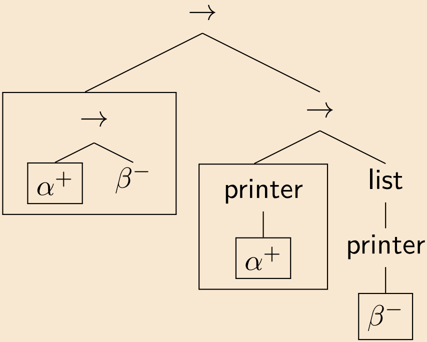
```
type 'a ref = { mutable contents : 'a }
val ref : 'a -> 'a ref
val ( ! ) : 'a ref -> 'a
val (:=) : 'a ref -> 'a -> unit

let r = ref None in
   r := Some "boom";
   match !r with
     None -> ()
   | Some f -> f ()
```

Relaxing the value restriction: variance

```
type 'a printer = 'a -> string
```

```
('a -> 'b) -> 'a printer -> 'b printer list
```



Relaxing the value restriction: the rules

Should we generalize?

- covariant type variables
- invariant type variables
- contravariant type variables
- bivariant type variables

Relaxing the value restriction: the rules

Should we generalize?

- covariant type variables ✓
- invariant type variables
- contravariant type variables
- bivariant type variables

Relaxing the value restriction: the rules

Should we generalize?

- covariant type variables ✓
- invariant type variables ✗
- contravariant type variables
- bivariant type variables

Relaxing the value restriction: the rules

Should we generalize?

- covariant type variables ✓
- invariant type variables ✗
- contravariant type variables ✗
- bivariant type variables

Relaxing the value restriction: the rules

Should we generalize?

- covariant type variables ✓
- invariant type variables ✗
- contravariant type variables ✗
- bivariant type variables ✓

Next time

Abstraction