Type inference

Madhavan Mukund, **S P Suresh**

Programming Language Concepts

Lecture 24, 18 April 2024

Typed λ-calculus: Church-Rosser

- Extend $\longrightarrow_\beta$ to one-step reduction $\longrightarrow$, as usual

Typed λ-calculus: Church-Rosser

- Extend $\longrightarrow_\beta$ to one-step reduction $\longrightarrow$, as usual
- Extend to many-step $\stackrel{*}{\longrightarrow}_\beta$ as usual

Typed λ-calculus: Church-Rosser

- Extend $\longrightarrow_\beta$ to one-step reduction $\longrightarrow$, as usual
- Extend to many-step $\xrightarrow{*}_\beta$ as usual
- $\xrightarrow{*}_\beta$ is Church-Rosser

Typed λ-calculus: Church-Rosser

- Extend $\longrightarrow_\beta$ to one-step reduction $\longrightarrow$, as usual
- Extend to many-step $\xrightarrow{*}_\beta$ as usual
- $\xrightarrow{*}_\beta$ is Church-Rosser
  - Same proof as for untyped λ-calculus

Typed λ-calculus: Normalization

- A λ-expression is

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$
    - **Counterexample:** Ω

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$
    - **Counterexample:** Ω
  - **strongly normalizing** if every reduction sequence is terminating

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$
    - **Counterexample:** Ω
  - **strongly normalizing** if every reduction sequence is terminating
    - **Example:** $(\lambda x \cdot y)(\lambda x \cdot x)$

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$
    - **Counterexample:** Ω
  - **strongly normalizing** if every reduction sequence is terminating
    - **Example:** $(\lambda x \cdot y)(\lambda x \cdot x)$
    - **Counterexample:** $(\lambda x \cdot y)\Omega$

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$
    - **Counterexample:** Ω
  - **strongly normalizing** if every reduction sequence is terminating
    - **Example:** $(\lambda x \cdot y)(\lambda x \cdot x)$
    - **Counterexample:** $(\lambda x \cdot y)\Omega$
- A λ-calculus is **weakly normalizing** if every term in the calculus is weakly normalizing

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$
    - **Counterexample:** Ω
  - **strongly normalizing** if every reduction sequence is terminating
    - **Example:** $(\lambda x \cdot y)(\lambda x \cdot x)$
    - **Counterexample:** $(\lambda x \cdot y)\Omega$
- A λ-calculus is **weakly normalizing** if every term in the calculus is weakly normalizing
- A λ-calculus is **strongly normalizing** if every term in the calculus is strongly normalizing

Typed λ-calculus: Normalization

- A λ-expression is
  - **(weakly) normalizing** if it has a normal form
    - **Example:** $(\lambda x \cdot y)\Omega$
    - **Counterexample:** Ω
  - **strongly normalizing** if every reduction sequence is terminating
    - **Example:** $(\lambda x \cdot y)(\lambda x \cdot x)$
    - **Counterexample:** $(\lambda x \cdot y)\Omega$
- A λ-calculus is **weakly normalizing** if every term in the calculus is weakly normalizing
- A λ-calculus is **strongly normalizing** if every term in the calculus is strongly normalizing
- The typed λ-calculus is both strongly and weakly normalizing

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
  - For instance, we cannot give a valid type to $x\,x$

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
  - For instance, we cannot give a valid type to $x\,x$
  - If it were typable, x would have type $\sigma \longrightarrow \tau$ as well as σ

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
    - For instance, we cannot give a valid type to $x\,x$
    - If it were typable, x would have type $\sigma \longrightarrow \tau$ as well as σ
- A term may admit multiple types

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
  - For instance, we cannot give a valid type to $x\,x$
  - If it were typable, x would have type $\sigma \longrightarrow \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \longrightarrow p$, $r \longrightarrow r$, $(p \longrightarrow q) \longrightarrow (p \longrightarrow q), \ldots$

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
  - For instance, we cannot give a valid type to $x\,x$
  - If it were typable, x would have type $\sigma \longrightarrow \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \longrightarrow p$, $r \longrightarrow r$, $(p \longrightarrow q) \longrightarrow (p \longrightarrow q), \ldots$
- $p \longrightarrow p$ is the simplest (least constrained) type – modulo variable renaming

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
  - For instance, we cannot give a valid type to $x\,x$
  - If it were typable, x would have type $\sigma \longrightarrow \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \longrightarrow p$, $r \longrightarrow r$, $(p \longrightarrow q) \longrightarrow (p \longrightarrow q), \ldots$
- $p \longrightarrow p$ is the simplest (least constrained) type – modulo variable renaming
- **Principal type**

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
  - For instance, we cannot give a valid type to $x\,x$
  - If it were typable, x would have type $\sigma \longrightarrow \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \longrightarrow p$, $r \longrightarrow r$, $(p \longrightarrow q) \longrightarrow (p \longrightarrow q)$, $\ldots$
- $p \longrightarrow p$ is the simplest (least constrained) type – modulo variable renaming
- **Principal type**
  - a type for a term M such that every other type for M is got by uniformly replacing each variable by a type

Curry typing: typability

- Given a term of the (untyped) λ-calculus, can it be given a type (assuming some types for the free variables)?
  - For instance, we cannot give a valid type to $x\,x$
  - If it were typable, x would have type $\sigma \longrightarrow \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \longrightarrow p$, $r \longrightarrow r$, $(p \longrightarrow q) \longrightarrow (p \longrightarrow q), \ldots$
- $p \longrightarrow p$ is the simplest (least constrained) type – modulo variable renaming
- **Principal type**
  - a type for a term M such that every other type for M is got by uniformly replacing each variable by a type
  - unique for each typable term – modulo renaming of variables!

Curry typing: typability

Definition (Typability problem)

Given a term M of the untyped λ-calculus, check whether it can be given a time (assuming some types for free variables)

Curry typing: typability

Definition (Typability problem)

Given a term M of the untyped λ-calculus, check whether it can be given a time (assuming some types for free variables)

Definition (Type inference)

Given a typable term M, compute its principal type

Curry typing: typability

Definition (Typability problem)

Given a term M of the untyped λ-calculus, check whether it can be given a time (assuming some types for free variables)

Definition (Type inference)

Given a typable term M, compute its principal type

Theorem

Typability and type inference for simply typed λ-calculus is solvable in polynomial time

Type inference

- For every λ-expression M, build

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)
  - τ_M, a type

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)
  - τ_M, a type
- E_M has a solution iff M is typable

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)
  - τ_M, a type
- E_M has a solution iff M is typable
- **Solution for** E_M – a substitution S mapping type variables to types that makes all equations true (both sides identical under S)

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)
  - τ_M, a type
- E_M has a solution iff M is typable
- **Solution for** E_M – a substitution S mapping type variables to types that makes all equations true (both sides identical under S)
- If S is the least constrained solution for E_M, $S(\tau_M)$ is a principal type for M

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)
  - τ_M, a type
- E_M has a solution iff M is typable
- **Solution for** E_M – a substitution S mapping type variables to types that makes all equations true (both sides identical under S)
- If S is the least constrained solution for E_M, $S(\tau_M)$ is a principal type for M
- Type variables in E_M and τ_M

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)
  - τ_M, a type
- E_M has a solution iff M is typable
- **Solution for** E_M – a substitution S mapping type variables to types that makes all equations true (both sides identical under S)
- If S is the least constrained solution for E_M, $S(\tau_M)$ is a principal type for M
- Type variables in E_M and τ_M
  - **main** – p_x, for $x \in \mathbf{fv}(M)$ **(if** $x \neq y$**,** $p_x \neq p_y$**)**

Type inference

- For every λ-expression M, build
  - E_M, a system of equations (over types)
  - τ_M, a type
- E_M has a solution iff M is typable
- **Solution for** E_M – a substitution S mapping type variables to types that makes all equations true (both sides identical under S)
- If S is the least constrained solution for E_M, $S(\tau_M)$ is a principal type for M
- Type variables in E_M and τ_M
  - **main** – p_x, for $x \in \mathbf{fv}(M)$ **(if** $x \neq y, p_x \neq p_y$**)**
  - **auxiliary** – not of the form p_x

Type inference ...

- M is the variable x

Type inference …

- M is the variable x
  - $E_M = \varnothing$

Type inference . . .

- M is the variable x
  - $E_M = \emptyset$
  - Define $\tau_M = p_x$

Type inference ...

- M is the variable x
  - $E_M = \emptyset$
  - Define $\tau_M = p_x$
- M is PQ

Type inference ...

- M is the variable x
  - $E_M = \varnothing$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P

Type inference …

- M is the variable x
  - $E_M = \emptyset$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P
  - Choose a fresh auxilliary type variable p

Type inference ...

- M is the variable x
  - $E_M = \emptyset$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P
  - Choose a fresh auxilliary type variable p
  - Define $\tau_M = p$

Type inference …

- M is the variable x
  - $E_M = \emptyset$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P
  - Choose a fresh auxilliary type variable p
  - Define $\tau_M = p$
  - $E_M = E_P \cup E_Q \cup \{\tau_P = \tau_Q \to p\}$

Type inference …

- M is the variable x
  - $E_M = \varnothing$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P
  - Choose a fresh auxilliary type variable p
  - Define $\tau_M = p$
  - $E_M = E_P \cup E_Q \cup \{\tau_P = \tau_Q \rightarrow p\}$
- M is $\lambda x \cdot P$

Type inference …

- M is the variable x
  - $E_M = \emptyset$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P
  - Choose a fresh auxilliary type variable p
  - Define $\tau_M = p$
  - $E_M = E_P \cup E_Q \cup \{\tau_P = \tau_Q \rightarrow p\}$
- M is $\lambda x \cdot P$
  - Choose a fresh auxilliary type variable p

Type inference ...

- M is the variable x
  - $E_M = \varnothing$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P
  - Choose a fresh auxilliary type variable p
  - Define $\tau_M = p$
  - $E_M = E_P \cup E_Q \cup \{\tau_P = \tau_Q \rightarrow p\}$
- M is $\lambda x \cdot P$
  - Choose a fresh auxilliary type variable p
  - $E_M = E_P[p_x := p]$

Type inference …

- M is the variable x
  - $E_M = \emptyset$
  - Define $\tau_M = p_x$
- M is PQ
  - Rename auxilliary variables in E_Q and τ_Q, to keep them distinct from auxilliary variables in E_P and τ_P
  - Choose a fresh auxilliary type variable p
  - Define $\tau_M = p$
  - $E_M = E_P \cup E_Q \cup \{\tau_P = \tau_Q \to p\}$
- M is $\lambda x \cdot P$
  - Choose a fresh auxilliary type variable p
  - $E_M = E_P[p_x := p]$
  - Define $\tau_M = p \to \tau_P[p_x := p]$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
  - $E_x = E_y = E_z = \emptyset$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
    - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
    - $E_x = E_y = E_z = \emptyset$
    - $\tau_{yz} = p, E_{yz} = \{p_y = p_z \rightarrow p\}$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
  - $E_x = E_y = E_z = \varnothing$
  - $\tau_{yz} = p, E_{yz} = \{p_y = p_z \rightarrow p\}$
  - $\tau_N = q, E_N = \{p_y = p_z \rightarrow p, p_x = p \rightarrow q\}$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
  - $E_x = E_y = E_z = \emptyset$
  - $\tau_{yz} = p, E_{yz} = \{p_y = p_z \rightarrow p\}$
  - $\tau_N = q, E_N = \{p_y = p_z \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda z \cdot N} = r \rightarrow q, E_{\lambda z \cdot N} = \{p_y = r \rightarrow p, p_x = p \rightarrow q\}$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
  - $E_x = E_y = E_z = \emptyset$
  - $\tau_{yz} = p, E_{yz} = \{p_y = p_z \rightarrow p\}$
  - $\tau_N = q, E_N = \{p_y = p_z \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda z \cdot N} = r \rightarrow q, E_{\lambda z \cdot N} = \{p_y = r \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda yz \cdot N} = s \rightarrow r \rightarrow q, E_{\lambda yz \cdot N} = \{s = r \rightarrow p, p_x = p \rightarrow q\}$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
  - $E_x = E_y = E_z = \emptyset$
  - $\tau_{yz} = p, E_{yz} = \{p_y = p_z \rightarrow p\}$
  - $\tau_N = q, E_N = \{p_y = p_z \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda z \cdot N} = r \rightarrow q, E_{\lambda z \cdot N} = \{p_y = r \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda yz \cdot N} = s \rightarrow r \rightarrow q, E_{\lambda yz \cdot N} = \{s = r \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_M = t \rightarrow s \rightarrow r \rightarrow q, E_M = \{s = r \rightarrow p, t = p \rightarrow q\}$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
  - $E_x = E_y = E_z = \emptyset$
  - $\tau_{yz} = p, E_{yz} = \{p_y = p_z \rightarrow p\}$
  - $\tau_N = q, E_N = \{p_y = p_z \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda z \cdot N} = r \rightarrow q, E_{\lambda z \cdot N} = \{p_y = r \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda yz \cdot N} = s \rightarrow r \rightarrow q, E_{\lambda yz \cdot N} = \{s = r \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_M = t \rightarrow s \rightarrow r \rightarrow q, E_M = \{s = r \rightarrow p, t = p \rightarrow q\}$
- A minimal solution for E_M is $S = \{s := r \rightarrow p, t := p \rightarrow q\}$

Type inference: example

- $M = \lambda xyz \cdot N$ where $N = x(yz)$
  - $\tau_x = p_x, \tau_y = p_y, \tau_z = p_z$
  - $E_x = E_y = E_z = \emptyset$
  - $\tau_{yz} = p, E_{yz} = \{p_y = p_z \rightarrow p\}$
  - $\tau_N = q, E_N = \{p_y = p_z \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda z \cdot N} = r \rightarrow q, E_{\lambda z \cdot N} = \{p_y = r \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_{\lambda yz \cdot N} = s \rightarrow r \rightarrow q, E_{\lambda yz \cdot N} = \{s = r \rightarrow p, p_x = p \rightarrow q\}$
  - $\tau_M = t \rightarrow s \rightarrow r \rightarrow q, E_M = \{s = r \rightarrow p, t = p \rightarrow q\}$
- A minimal solution for E_M is $S = \{s := r \rightarrow p, t := p \rightarrow q\}$
- The principal type of M: $S(\tau_M) = (p \rightarrow q) \rightarrow (r \rightarrow p) \rightarrow (r \rightarrow q)$

Type inference: example

- $M = PQ$ where $P = Q = \lambda x \cdot x$

Type inference: example

- $M = PQ$ where $P = Q = \lambda x \cdot x$
  - $\tau_x = p_x, E_x = \varnothing$

Type inference: example

- $M = PQ$ where $P = Q = \lambda x \cdot x$
  - $\tau_x = p_x, E_x = \varnothing$
  - $\tau_P = p \rightarrow p, E_P = \varnothing$

Type inference: example

- $M = PQ$ where $P = Q = \lambda x \cdot x$
  - $\tau_x = p_x, E_x = \varnothing$
  - $\tau_P = p \rightarrow p, E_P = \varnothing$
  - $\tau_Q = q \rightarrow q, E_Q = \varnothing$

Type inference: example

- $M = PQ$ where $P = Q = \lambda x \cdot x$
  - $\tau_x = p_x, E_x = \varnothing$
  - $\tau_P = p \rightarrow p, E_P = \varnothing$
  - $\tau_Q = q \rightarrow q, E_Q = \varnothing$
  - $\tau_{PQ} = r, E_{PQ} = \{p \rightarrow p = (q \rightarrow q) \rightarrow r\}$

Type inference: example

- $M = PQ$ where $P = Q = \lambda x \cdot x$
  - $\tau_x = p_x, E_x = \varnothing$
  - $\tau_P = p \to p, E_P = \varnothing$
  - $\tau_Q = q \to q, E_Q = \varnothing$
  - $\tau_{PQ} = r, E_{PQ} = \{p \to p = (q \to q) \to r\}$
- A minimal solution for E_M is $S = \{p := q \to q, r := q \to q\}$

Type inference: example

- $M = PQ$ where $P = Q = \lambda x \cdot x$
  - $\tau_x = p_x, E_x = \varnothing$
  - $\tau_P = p \to p, E_P = \varnothing$
  - $\tau_Q = q \to q, E_Q = \varnothing$
  - $\tau_{PQ} = r, E_{PQ} = \{p \to p = (q \to q) \to r\}$
- A minimal solution for E_M is $S = \{p := q \to q, r := q \to q\}$
- The principal type of M: $S(\tau_M) = q \to q$

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, ...

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, …
- Type constructors too – *[a]*, *(a,b)*, …

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, …
- Type constructors too – *[a]*, *(a,b)*, …
- constant terms and constant functions

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, ...
- Type constructors too – *[a]*, *(a,b)*, ...
- constant terms and constant functions
  - 0 : *Int*, *True* : *Bool*

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, ...
- Type constructors too – *[a]*, *(a,b)*, ...
- constant terms and constant functions
  - $0 : Int$, $True : Bool$
  - $cons : a \longrightarrow [a] \longrightarrow [a]$

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, ...
- Type constructors too – *[a]*, *(a,b)*, ...
- constant terms and constant functions
  - $0 : Int$, $True : Bool$
  - $cons : a \longrightarrow [a] \longrightarrow [a]$
  - $if : Bool \longrightarrow a \longrightarrow a \longrightarrow a$

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, ...
- Type constructors too – *[a]*, *(a,b)*, ...
- constant terms and constant functions
  - $0 : Int$, $True : Bool$
  - $cons : a \rightarrow [a] \rightarrow [a]$
  - $if : Bool \rightarrow a \rightarrow a \rightarrow a$
- **Polymorphic** – each occurrence of *if*, *cons*, etc. is given a fresh instance of the types

Type inference: constant types

- Can introduce constant types – *Int*, *Bool*, ...
- Type constructors too – *[a]*, *(a,b)*, ...
- constant terms and constant functions
  - $0 : Int$, $True : Bool$
  - $cons : a \longrightarrow [a] \longrightarrow [a]$
  - $if : Bool \longrightarrow a \longrightarrow a \longrightarrow a$
- **Polymorphic** – each occurrence of *if*, *cons*, etc. is given a fresh instance of the types
- The type inference algorithm is more or less unchanged!

Type inference: richer typing

- $M = (\lambda x \cdot x)(\lambda x \cdot x)$ has principal type $q \to q$

Type inference: richer typing

- $M = (\lambda x \cdot x)(\lambda x \cdot x)$ has principal type $q \to q$
- Let M_1 be **let** $y = \lambda x \cdot x$ **in** yy

Type inference: richer typing

- $M = (\lambda x \cdot x)(\lambda x \cdot x)$ has principal type $q \rightarrow q$
- Let M_1 be **let** $y = \lambda x \cdot x$ **in** yy
- Let M_2 be $(\lambda y \cdot yy)(\lambda x \cdot x)$

Type inference: richer typing

- $M = (\lambda x \cdot x)(\lambda x \cdot x)$ has principal type $q \longrightarrow q$
- Let M_1 be $\textbf{let}\ y = \lambda x \cdot x\ \textbf{in}\ yy$
- Let M_2 be $(\lambda y \cdot yy)(\lambda x \cdot x)$
- M_1 is equivalent to M and has the same principal type

Type inference: richer typing

- $M = (\lambda x \cdot x)(\lambda x \cdot x)$ has principal type $q \rightarrow q$
- Let M_1 be $\textbf{let } y = \lambda x \cdot x \textbf{ in } yy$
- Let M_2 be $(\lambda y \cdot yy)(\lambda x \cdot x)$
- M_1 is equivalent to M and has the same principal type
- M_2 is not typable, because $\lambda y \cdot yy$ is not typable

Type inference: richer typing

- $M = (\lambda x \cdot x)(\lambda x \cdot x)$ has principal type $q \rightarrow q$
- Let M_1 be **let** $y = \lambda x \cdot x$ **in** yy
- Let M_2 be $(\lambda y \cdot yy)(\lambda x \cdot x)$
- M_1 is equivalent to M and has the same principal type
- M_2 is not typable, because $\lambda y \cdot yy$ is not typable
- M_1 is typable despite the occurrence of yy

Type inference: richer typing

- $M = (\lambda x \cdot x)(\lambda x \cdot x)$ has principal type $q \longrightarrow q$
- Let M_1 be **let** $y = \lambda x \cdot x$ **in** yy
- Let M_2 be $(\lambda y \cdot yy)(\lambda x \cdot x)$
- M_1 is equivalent to M and has the same principal type
- M_2 is not typable, because $\lambda y \cdot yy$ is not typable
- M_1 is typable despite the occurrence of yy
    - **variable defined by local definition** – treated differently

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
  - Find principal types of $M_1, \ldots, M_n$

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
  - Find principal types of $M_1, \ldots, M_n$
  - Set τ_{x_i} to be τ_{M_i}, and $E_{x_i} = \varnothing$

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
  - Find principal types of $M_1, \ldots, M_n$
  - Set τ_{x_i} to be τ_{M_i}, and $E_{x_i} = \emptyset$
  - Find the type of N as usual, using the above definition for τ_{x_i}'s

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 ; \cdots ; x_n = M_n\}$ **in** N
    - Find principal types of $M_1, \ldots, M_n$
    - Set τ_{x_i} to be τ_{M_i}, and $E_{x_i} = \emptyset$
    - Find the type of N as usual, using the above definition for τ_{x_i}'s
    - Each occurrence of x_i in N will get a different instance of τ_{x_i} as its type

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 \;;\; \cdots \;;\; x_n = M_n\}$ **in** N
  - Find principal types of $M_1, \ldots, M_n$
  - Set τ_{x_i} to be τ_{M_i}, and $E_{x_i} = \emptyset$
  - Find the type of N as usual, using the above definition for τ_{x_i}'s
  - Each occurrence of x_i in N will get a different instance of τ_{x_i} as its type
  - All auxilliary type variables in τ_{x_i} will be renamed to fresh variables

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
    - Find principal types of $M_1, \ldots, M_n$
    - Set τ_{x_i} to be τ_{M_i}, and $E_{x_i} = \emptyset$
    - Find the type of N as usual, using the above definition for τ_{x_i}'s
    - Each occurrence of x_i in N will get a different instance of τ_{x_i} as its type
    - All auxilliary type variables in τ_{x_i} will be renamed to fresh variables
    - Main type variables of the form p_x will not be renamed

Type inference: non-recursive local definitions

- M is **let** $\{x_1 = M_1 \,;\, \cdots \,;\, x_n = M_n\}$ **in** N
  - Find principal types of $M_1, \ldots, M_n$
  - Set τ_{x_i} to be τ_{M_i}, and $E_{x_i} = \varnothing$
  - Find the type of N as usual, using the above definition for τ_{x_i}'s
  - Each occurrence of x_i in N will get a different instance of τ_{x_i} as its type
  - All auxilliary type variables in τ_{x_i} will be renamed to fresh variables
  - Main type variables of the form p_x will not be renamed
- x_i's are used in N as **polymorphic expressions**

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \rightarrow p$, for some auxilliary type variable p

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \to p$, for some auxilliary type variable p
  - yy is of the form PQ

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \rightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \longrightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \longrightarrow p$

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \longrightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \longrightarrow p$
  - Type of second y is $q \longrightarrow q$

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \longrightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \longrightarrow p$
  - Type of second y is $q \longrightarrow q$
  - Now solve as usual!

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \rightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \rightarrow p$
  - Type of second y is $q \rightarrow q$
  - Now solve as usual!
- Let M be **let** $\{f = \lambda y \cdot x\ ;\ g = \lambda x \cdot x\}$ **in** gf

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \longrightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \longrightarrow p$
  - Type of second y is $q \longrightarrow q$
  - Now solve as usual!
- Let M be **let** $\{f = \lambda y \cdot x \,;\; g = \lambda x \cdot x\}$ **in** gf
  - $\tau_f = p \longrightarrow p_x$

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \longrightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \longrightarrow p$
  - Type of second y is $q \longrightarrow q$
  - Now solve as usual!
- Let M be **let** $\{f = \lambda y \cdot x\, ;\; g = \lambda x \cdot x\}$ **in** gf
  - $\tau_f = p \longrightarrow p_x$
  - $\tau_g = q \longrightarrow q$

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \rightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \rightarrow p$
  - Type of second y is $q \rightarrow q$
  - Now solve as usual!
- Let M be **let** $\{f = \lambda y \cdot x \,;\; g = \lambda x \cdot x\}$ **in** gf
  - $\tau_f = p \rightarrow p_x$
  - $\tau_g = q \rightarrow q$
  - $\tau_{gf} = r, E_{gf} = \{q \rightarrow q = (p \rightarrow p_x) \rightarrow r\}$

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \rightarrow p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \rightarrow p$
  - Type of second y is $q \rightarrow q$
  - Now solve as usual!
- Let M be **let** $\{f = \lambda y \cdot x \,;\; g = \lambda x \cdot x\}$ **in** gf
  - $\tau_f = p \rightarrow p_x$
  - $\tau_g = q \rightarrow q$
  - $\tau_{gf} = r, E_{gf} = \{q \rightarrow q = (p \rightarrow p_x) \rightarrow r\}$
  - $S = q := p \rightarrow p_x, r := p \rightarrow p_x$ is a solution

Type inference: non-recursive local definitions

- Consider **let** $y = \lambda x \cdot x$ **in** yy
  - $\tau_y = p \to p$, for some auxilliary type variable p
  - yy is of the form PQ
  - We rename auxilliary type variables in τ_Q
  - Type of the first y is $p \to p$
  - Type of second y is $q \to q$
  - Now solve as usual!
- Let M be **let** $\{f = \lambda y \cdot x \; ; \; g = \lambda x \cdot x\}$ **in** gf
  - $\tau_f = p \to p_x$
  - $\tau_g = q \to q$
  - $\tau_{gf} = r, E_{gf} = \{q \to q = (p \to p_x) \to r\}$
  - $S = q := p \to p_x, r := p \to p_x$ is a solution
  - Principal type of M is $p \to p_x$

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \,;\, \cdots \,;\, x_n = M_n\}$ **in** N

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1\ ;\ \cdots\ ;\ x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 ; \cdots ; x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$
  - Obtain $\sigma_1, \ldots, \sigma_n$ and $E_1, \ldots, E_n$

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \,;\, \cdots \,;\, x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$
  - Obtain $\sigma_1, \ldots, \sigma_n$ and $E_1, \ldots, E_n$
    - $\sigma_i := \tau_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \,;\, \cdots \,;\, x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$
  - Obtain $\sigma_1, \ldots, \sigma_n$ and $E_1, \ldots, E_n$
    - $\sigma_i := \tau_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
    - $E_i := E_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1\ ;\ \cdots\ ;\ x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$
  - Obtain $\sigma_1, \ldots, \sigma_n$ and $E_1, \ldots, E_n$
    - $\sigma_i := \tau_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
    - $E_i := E_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
  - Solve $E := E_1 \cup \cdots \cup E_n \cup \{q_1 = \sigma_1, \ldots, q_n = \sigma_n\}$

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$
  - Obtain $\sigma_1, \ldots, \sigma_n$ and $E_1, \ldots, E_n$
    - $\sigma_i := \tau_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
    - $E_i := E_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
  - Solve $E := E_1 \cup \cdots \cup E_n \cup \{q_1 = \sigma_1, \ldots, q_n = \sigma_n\}$
  - Let S be the most general solution to E

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \; ; \; \cdots \; ; \; x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$
  - Obtain $\sigma_1, \ldots, \sigma_n$ and $E_1, \ldots, E_n$
    - $\sigma_i := \tau_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
    - $E_i := E_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
  - Solve $E := E_1 \cup \cdots \cup E_n \cup \{q_1 = \sigma_1, \ldots, q_n = \sigma_n\}$
  - Let S be the most general solution to E
  - $\tau_{x_i} := S(q_i)$

Type inference: recursive local definitions

- M is **letrec** $\{x_1 = M_1 \,;\, \cdots \,;\, x_n = M_n\}$ **in** N
  - Build each τ_{M_i} and E_{M_i}, treating each x_j as a free variable with type p_{x_j}
  - Ensure that the auxilliary variables in the E_{M_i}'s and τ_{M_i}'s are all distinct
  - Choose n fresh type variables $q_1, \ldots, q_n$
  - Obtain $\sigma_1, \ldots, \sigma_n$ and $E_1, \ldots, E_n$
    - $\sigma_i := \tau_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
    - $E_i := E_{M_i}[p_{x_1} := q_1, \ldots, p_{x_n} := q_n]$
  - Solve $E := E_1 \cup \cdots \cup E_n \cup \{q_1 = \sigma_1, \ldots, q_n = \sigma_n\}$
  - Let S be the most general solution to E
  - $\tau_{x_i} := S(q_i)$
  - Find the type of N as usual, using the above τ_{x_i}'s

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x
  - Let M_1 be $\lambda f \cdot f(xf)$

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x
  - Let M_1 be $\lambda f \cdot f(xf)$
  - $\tau_{M_1} = p \rightarrow r$ and $E_{M_1} = \{p_x = p \rightarrow q, p = q \rightarrow r\}$

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x
  - Let M_1 be $\lambda f \cdot f(xf)$
  - $\tau_{M_1} = p \rightarrow r$ and $E_{M_1} = \{p_x = p \rightarrow q, p = q \rightarrow r\}$
  - Now $\sigma_1 = p \rightarrow r$ and $E_1 = \{q_1 = p \rightarrow q, p = q \rightarrow r\}$

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x
  - Let M_1 be $\lambda f \cdot f(xf)$
  - $\tau_{M_1} = p \rightarrow r$ and $E_{M_1} = \{p_x = p \rightarrow q, p = q \rightarrow r\}$
  - Now $\sigma_1 = p \rightarrow r$ and $E_1 = \{q_1 = p \rightarrow q, p = q \rightarrow r\}$
  - $E = \{q_1 = p \rightarrow q, p = q \rightarrow r, q_1 = p \rightarrow r\}$

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x
  - Let M_1 be $\lambda f \cdot f(xf)$
  - $\tau_{M_1} = p \rightarrow r$ and $E_{M_1} = \{p_x = p \rightarrow q, p = q \rightarrow r\}$
  - Now $\sigma_1 = p \rightarrow r$ and $E_1 = \{q_1 = p \rightarrow q, p = q \rightarrow r\}$
  - $E = \{q_1 = p \rightarrow q, p = q \rightarrow r, q_1 = p \rightarrow r\}$
  - Solution for E is $\{q := r, p := r \rightarrow r, q_1 := (r \rightarrow r) \rightarrow r\}$

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x
  - Let M_1 be $\lambda f \cdot f(xf)$
  - $\tau_{M_1} = p \rightarrow r$ and $E_{M_1} = \{p_x = p \rightarrow q, p = q \rightarrow r\}$
  - Now $\sigma_1 = p \rightarrow r$ and $E_1 = \{q_1 = p \rightarrow q, p = q \rightarrow r\}$
  - $E = \{q_1 = p \rightarrow q, p = q \rightarrow r, q_1 = p \rightarrow r\}$
  - Solution for E is $\{q := r, p := r \rightarrow r, q_1 := (r \rightarrow r) \rightarrow r\}$
  - $\tau_x := (r \rightarrow r) \rightarrow r$

Type inference: recursive local definitions

- Consider **letrec** $x = \lambda f \cdot f(xf)$ **in** x
  - Let M_1 be $\lambda f \cdot f(xf)$
  - $\tau_{M_1} = p \to r$ and $E_{M_1} = \{p_x = p \to q, p = q \to r\}$
  - Now $\sigma_1 = p \to r$ and $E_1 = \{q_1 = p \to q, p = q \to r\}$
  - $E = \{q_1 = p \to q, p = q \to r, q_1 = p \to r\}$
  - Solution for E is $\{q := r, p := r \to r, q_1 := (r \to r) \to r\}$
  - $\tau_x := (r \to r) \to r$
- The type of **letrec** $x = \lambda f \cdot f(xf)$ **in** x is thus $(r \to r) \to r$