Programming Language Concepts: Lecture 22

S P Suresh

April 7, 2021

Adding types to λ-calculus

- The basic λ-calculus is untyped

Adding types to λ-calculus

- The basic λ-calculus is untyped
- The first functional programming language, LISP, was also untyped

Adding types to λ-calculus

- The basic λ-calculus is untyped
- The first functional programming language, LISP, was also untyped
- Modern languages such as Haskell, ML, …are typed

Adding types to λ-calculus

- The basic λ-calculus is untyped
- The first functional programming language, **LISP**, was also untyped
- Modern languages such as **Haskell**, **ML**, …are typed
- What is the theoretical foundation for such languages?

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

  Tuples If `a1`, `a2`, …, `ak` are types, so is `(a1, a2, ..., ak)`

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

  Tuples If `a1`, `a2`, …, `ak` are types, so is `(a1, a2, ..., ak)`
- Function types

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

  Tuples If `a1`, `a2`, …, `ak` are types, so is `(a1, a2, ..., ak)`
- Function types
  - If `a`, `b` are types, so is `a -> b`

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

  Tuples If `a1`, `a2`, ..., `ak` are types, so is `(a1, a2, ..., ak)`
- Function types
  - If `a`, `b` are types, so is `a -> b`
  - Function with input of type `a` and output of type `b`

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

  Tuples If `a1`, `a2`, ..., `ak` are types, so is `(a1, a2, ..., ak)`
- Function types
  - If `a`, `b` are types, so is `a -> b`
  - Function with input of type `a` and output of type `b`
- User defined types

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

  Tuples If `a1`, `a2`, …, `ak` are types, so is `(a1, a2, ..., ak)`

- Function types
  - If `a`, `b` are types, so is `a -> b`
  - Function with input of type `a` and output of type `b`
- User defined types
  - `data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat`

Types in functional programming

The structure of types in Haskell

- Basic types—`Int`, `Bool`, `Float`, `Char`
- Structured types

  Lists If `a` is a type, so is `[a]`

  Tuples If `a1`, `a2`, …, `ak` are types, so is `(a1, a2, ..., ak)`
- Function types
  - If `a`, `b` are types, so is `a -> b`
  - Function with input of type `a` and output of type `b`
- User defined types
  - `data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat`
  - `data BTree a = Nil | Node (BTree a) a (BTree a)`

Adding types to λ-calculus

- Set Λ of untyped lambda expressions given by the syntax

$$\Lambda = x \mid \lambda x.M \mid M\,N$$

where $x \in Var$, $M, N \in \Lambda$

Adding types to λ-calculus

- Set Λ of untyped lambda expressions given by the syntax

$$\Lambda = x \mid \lambda x.M \mid MN$$

  where $x \in Var$, $M, N \in \Lambda$

- Add a syntax for types

Adding types to λ-calculus

- Set Λ of untyped lambda expressions given by the syntax

$$\Lambda = x \mid \lambda x.M \mid M N$$

  where $x \in Var$, $M, N \in \Lambda$

- Add a syntax for types
- When constructing expressions, build up the type from the types of the parts

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, …) or user-defined types

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, …) or user-defined types
- Function types arise naturally

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, …) or user-defined types
- Function types arise naturally
  - $p \to q$

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, …) or user-defined types
- Function types arise naturally
  - $p \rightarrow q$
  - $p \rightarrow (q \rightarrow p)$

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, …) or user-defined types
- Function types arise naturally
  - $p \to q$
  - $p \to (q \to p)$
  - $(p \to r) \to r$

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, ...) or user-defined types
- Function types arise naturally
  - $p \to q$
  - $p \to (q \to p)$
  - $(p \to r) \to r$
  - $(p \to p) \to (p \to q)$

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, …) or user-defined types
- Function types arise naturally
  - $p \to q$
  - $p \to (q \to p)$
  - $(p \to r) \to r$
  - $(p \to p) \to (p \to q)$
- $\sigma, \tau, \ldots$ stand for arbitrary types

Adding types to λ-calculus

- Assume an infinite set of type variables $p, q, r, p_1, q', \ldots$
- No structured types (lists, tuples, …) or user-defined types
- Function types arise naturally
  - $p \to q$
  - $p \to (q \to p)$
  - $(p \to r) \to r$
  - $(p \to p) \to (p \to q)$
- $\sigma, \tau, \ldots$ stand for arbitrary types
- $\to$ is right associative: $\sigma \to \tau \to \theta$ is short for $\sigma \to (\tau \to \theta)$

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:
  - $x \in Var_\sigma \implies x \in \Lambda_\sigma$

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:
  - $x \in Var_\sigma \implies x \in \Lambda_\sigma$
  - $M \in \Lambda_{\sigma\to\tau}, N \in \Lambda_\sigma \implies MN \in \Lambda_\tau$

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:
  - $x \in Var_\sigma \implies x \in \Lambda_\sigma$
  - $M \in \Lambda_{\sigma\to\tau}, N \in \Lambda_\sigma \implies M\,N \in \Lambda_\tau$
  - $x \in Var_\sigma, M \in \Lambda_\tau \implies \lambda x \cdot M \in \Lambda_{\sigma\to\tau}$

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:
  - $x \in Var_\sigma \implies x \in \Lambda_\sigma$
  - $M \in \Lambda_{\sigma \to \tau}, N \in \Lambda_\sigma \implies M N \in \Lambda_\tau$
  - $x \in Var_\sigma, M \in \Lambda_\tau \implies \lambda x \cdot M \in \Lambda_{\sigma \to \tau}$
- β-reduction is as usual: $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:
  - $x \in Var_\sigma \implies x \in \Lambda_\sigma$
  - $M \in \Lambda_{\sigma\to\tau}, N \in \Lambda_\sigma \implies MN \in \Lambda_\tau$
  - $x \in Var_\sigma, M \in \Lambda_\tau \implies \lambda x \cdot M \in \Lambda_{\sigma\to\tau}$
- β-reduction is as usual: $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$
  - $\lambda x \cdot M$ has type $\sigma \to \tau$ and N has type σ, for some σ and τ

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:
  - $x \in Var_\sigma \implies x \in \Lambda_\sigma$
  - $M \in \Lambda_{\sigma\to\tau}, N \in \Lambda_\sigma \implies MN \in \Lambda_\tau$
  - $x \in Var_\sigma, M \in \Lambda_\tau \implies \lambda x \cdot M \in \Lambda_{\sigma\to\tau}$
- β-reduction is as usual: $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$
  - $\lambda x \cdot M$ has type $\sigma \to \tau$ and N has type σ, for some σ and τ
  - x has type σ, so matches N

Adding types to λ-calculus: Church typing

- For every type σ, an infinite set Var_σ of (term) variables of type σ
- Define Λ_σ for all σ, by simultaneous induction:
  - $x \in Var_\sigma \implies x \in \Lambda_\sigma$
  - $M \in \Lambda_{\sigma\to\tau}, N \in \Lambda_\sigma \implies M\,N \in \Lambda_\tau$
  - $x \in Var_\sigma, M \in \Lambda_\tau \implies \lambda x \cdot M \in \Lambda_{\sigma\to\tau}$
- β-reduction is as usual: $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$
  - $\lambda x \cdot M$ has type $\sigma \to \tau$ and N has type σ, for some σ and τ
  - x has type σ, so matches N
  - Both sides have type τ

Church typing: alternate presentation

- **Environment** Γ – a finite set of pairs $\{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where each $x_i \in Var_{\sigma_i}$

Church typing: alternate presentation

- **Environment** Γ – a finite set of pairs $\{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where each $x_i \in Var_{\sigma_i}$
- We write $\Gamma, y : \tau$ for $\Gamma \cup \{(y : \tau)\}$

Church typing: alternate presentation

- **Environment** Γ – a finite set of pairs $\{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where each $x_i \in Var_{\sigma_i}$
- We write $\Gamma, y : \tau$ for $\Gamma \cup \{(y : \tau)\}$
- The typing rules:

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x \cdot M) : \sigma \rightarrow \tau} \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M N) : \tau}$$

Church typing: examples

- If $x \in Var_p$, $\lambda x \cdot x : p \rightarrow p$

Church typing: examples

- If $x \in Var_p$, $\lambda x \cdot x : p \rightarrow p$
- If $x \in Var_p, y \in Var_q$, $\lambda x\, y \cdot x : p \rightarrow q \rightarrow p$

Church typing: examples

- If $x \in Var_p$, $\lambda x \cdot x : p \rightarrow p$
- If $x \in Var_p$, $y \in Var_q$, $\lambda x y \cdot x : p \rightarrow q \rightarrow p$
- If $x \in Var_{p \rightarrow q \rightarrow r}$, $y \in Var_{p \rightarrow q}$, $z \in Var_p$,

$$\lambda x y z \cdot x z (y z) : (p \rightarrow q \rightarrow r) \rightarrow (p \rightarrow q) \rightarrow p \rightarrow r$$

Church typing: Church-Rosser

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

Church typing: Church-Rosser

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

Church typing: Church-Rosser

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

Church typing: Church-Rosser

- Extend $\longrightarrow_\beta$ to one-step reduction $\longrightarrow$, as usual
- Extend to many-step $\stackrel{*}{\longrightarrow}_\beta$ as usual
- $\stackrel{*}{\longrightarrow}_\beta$ is Church-Rosser
  - Cannot easily adapt the proof for untyped λ-calculus

Church typing: Church-Rosser

- Extend $\longrightarrow_\beta$ to one-step reduction $\longrightarrow$, as usual
- Extend to many-step $\stackrel{*}{\longrightarrow}_\beta$ as usual
- $\stackrel{*}{\longrightarrow}_\beta$ is Church-Rosser
  - Cannot easily adapt the proof for untyped λ-calculus
  - Use weak Church-Rosser for Church typing and **strong normalization** instead

Church typing: Normalization

- A λ-expression is

Church typing: Normalization

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

Church typing: Normalization

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

Church typing: Normalization

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

Church typing: 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

Church typing: 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:** $(\lambda x \cdot y)(\lambda x \cdot x)$

Church typing: 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: $(\lambda x \cdot y)\Omega$

Church typing: 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

Church typing: 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

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Proof.

□

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Proof.

- **Terminating reduction strategy**

□

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Proof.

- **Terminating reduction strategy**
- **complexity of a redex**: $\delta((\lambda x \cdot M)N) = \mathit{size}(\sigma \to \tau)$, where $\sigma \to \tau$ is the type of $\lambda x \cdot M$

□

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Proof.

- **Terminating reduction strategy**
- **complexity of a redex**: $\delta((\lambda x \cdot M)N) = \mathit{size}(\sigma \to \tau)$, where $\sigma \to \tau$ is the type of $\lambda x \cdot M$
- Pick an innermost redex t with maximum δ value (among all redexes inside the original expression M)

□

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Proof.

- **Terminating reduction strategy**
- **complexity of a redex**: $\delta((\lambda x \cdot M)N) = \mathit{size}(\sigma \to \tau)$, where $\sigma \to \tau$ is the type of $\lambda x \cdot M$
- Pick an innermost redex t with maximum δ value (among all redexes inside the original expression M)
  - If a subterm t' of t is also a redex, then $\delta(t') < \delta(t)$

□

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Proof.

- **Terminating reduction strategy**
- **complexity of a redex**: $\delta((\lambda x \cdot M)N) = size(\sigma \to \tau)$, where $\sigma \to \tau$ is the type of $\lambda x \cdot M$
- Pick an innermost redex t with maximum δ value (among all redexes inside the original expression M)
  - If a subterm t' of t is also a redex, then $\delta(t') < \delta(t)$
- Replace t by u, where u is got by contracting t

□

Church typing: Weak normalization

Theorem

The λ-calculus with Church typing is weakly normalizing

Proof.

- **Terminating reduction strategy**
- **complexity of a redex**: $\delta((\lambda x \cdot M)N) = size(\sigma \to \tau)$, where $\sigma \to \tau$ is the type of $\lambda x \cdot M$
- Pick an innermost redex t with maximum δ value (among all redexes inside the original expression M)
  - If a subterm t' of t is also a redex, then $\delta(t') < \delta(t)$
- Replace t by u, where u is got by contracting t
- This strategy is guaranteed to rerminate!

□

Church typing: Strong normalization

Theorem

The λ-calculus with Church typing is strongly normalizing

Church typing: Strong normalization

Theorem

The λ-calculus with Church typing is strongly normalizing

Church typing: Strong normalization

Theorem

The λ-calculus with Church typing is strongly normalizing

Proof.

□

Church typing: Strong normalization

Theorem

The λ-calculus with Church typing is strongly normalizing

Proof.

- Define $\mathbf{Red}_\sigma \subseteq \Lambda_\sigma$ **(Logically complex!)**

$$t \in \mathbf{Red}_p \iff t \text{ is strongly normalizing}$$

$$t \in \mathbf{Red}_{\sigma \to \tau} \iff \forall u \left[u \in \mathbf{Red}_\sigma \implies t\, u \in \mathbf{Red}_\tau \right]$$

□

Church typing: Strong normalization

Theorem

The λ-calculus with Church typing is strongly normalizing

Proof.

- Define $\mathbf{Red}_\sigma \subseteq \Lambda_\sigma$ **(Logically complex!)**

$$t \in \mathbf{Red}_p \iff t \text{ is strongly normalizing}$$

$$t \in \mathbf{Red}_{\sigma\to\tau} \iff \forall u \left[u \in \mathbf{Red}_\sigma \implies t\,u \in \mathbf{Red}_\tau \right]$$

- For all σ, if $t \in \mathbf{Red}_\sigma$ then t is strongly normalizing **(Induction on types)**

□

Church typing: Strong normalization

Theorem

The λ-calculus with Church typing is strongly normalizing

Proof.

- Define $\mathbf{Red}_\sigma \subseteq \Lambda_\sigma$ **(Logically complex!)**

$$t \in \mathbf{Red}_p \iff t \text{ is strongly normalizing}$$
$$t \in \mathbf{Red}_{\sigma \to \tau} \iff \forall u \left[u \in \mathbf{Red}_\sigma \implies t\,u \in \mathbf{Red}_\tau \right]$$

- For all σ, if $t \in \mathbf{Red}_\sigma$ then t is strongly normalizing **(Induction on types)**
- For all terms t, if $t \in \Lambda_\sigma$ then $t \in \mathbf{Red}_\sigma$ **(Induction on term size)**

□

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify **typable** terms

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify **typable** terms
- Each typable term has a **judgement** asserting its type

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify **typable** terms
- Each typable term has a **judgement** asserting its type
- Types of variables are given by an **environment**

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify **typable** terms
- Each typable term has a **judgement** asserting its type
- Types of variables are given by an **environment**
  - A finite set of pairs $\Gamma = \{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where the x_i are **distinct** variables, and the σ_i are types

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify **typable** terms
- Each typable term has a **judgement** asserting its type
- Types of variables are given by an **environment**
  - A finite set of pairs $\Gamma = \{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where the x_i are **distinct** variables, and the σ_i are types
  - no requirement that $x_i \in Var_{\sigma_i}$ – there is no Var_{σ_i}!

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify **typable** terms
- Each typable term has a **judgement** asserting its type
- Types of variables are given by an **environment**
  - A finite set of pairs $\Gamma = \{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where the x_i are **distinct** variables, and the σ_i are types
  - no requirement that $x_i \in Var_{\sigma_i}$ – there is no Var_{σ_i}!
- The typing rules:

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x \cdot M) : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M N) : \tau}$$

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify **typable** terms
- Each typable term has a **judgement** asserting its type
- Types of variables are given by an **environment**
  - A finite set of pairs $\Gamma = \{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where the x_i are **distinct** variables, and the σ_i are types
  - no requirement that $x_i \in Var_{\sigma_i}$ – there is no Var_{σ_i}!
- The typing rules:

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x \cdot M) : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

- β-reduction is as usual: $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$

Adding types to λ-calculus: Curry typing

- Terms of the untyped lambda calculus – identify typable terms
- Each typable term has a judgement asserting its type
- Types of variables are given by an environment
  - A finite set of pairs $\Gamma = \{(x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\}$ where the x_i are distinct variables, and the σ_i are types
  - no requirement that $x_i \in Var_{\sigma_i}$ – there is no Var_{σ_i}!
- The typing rules:

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x \cdot M) : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M N) : \tau}$$

- β-reduction is as usual: $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$
  - Types match

Curry typing: Examples

- $$\frac{x : p \vdash x : p}{\vdash \lambda x \cdot x : p \to p}$$

Curry typing: Examples

- $$\frac{x : p \vdash x : p}{\vdash \lambda x \cdot x : p \to p}$$

- $$\frac{\dfrac{x : p, y : q \vdash x : p}{x : p \vdash \lambda y \cdot x : q \to p}}{\vdash \lambda x y \cdot x : p \to (q \to p)}$$

Curry typing: Examples

- Let $\Gamma = \{x : p \to q \to r, y : p \to q, z : p\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x : p \to q \to r \quad \Gamma \vdash z : p}{\Gamma \vdash xz : q \to r} \quad \dfrac{\Gamma \vdash y : p \to q \quad \Gamma \vdash z : p}{\Gamma \vdash yz : q}}{\Gamma \vdash xz(yz) : r}}{x : p \to q \to r, y : p \to q \vdash \lambda z \cdot xz(yz) : p \to r}}{x : p \to q \to r \vdash \lambda yz \cdot xz(yz) : (p \to q) \to (p \to r)}}{\vdash \lambda xyz \cdot xz(yz) : (p \to q \to r) \to (p \to q) \to (p \to r)}}{}$$

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 $\sigma \to \tau$ 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 $\sigma \to \tau$ 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 $\sigma \to \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \to p$, $r \to r$, $(p \to q) \to (p \to q)$, $\ldots$

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 $\sigma \to \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \to p$, $r \to r$, $(p \to q) \to (p \to q)$, …
- $p \to p$ 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 $\sigma \to \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \to p$, $r \to r$, $(p \to q) \to (p \to q)$, $\ldots$
- $p \to p$ 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 $\sigma \rightarrow \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \rightarrow p$, $r \rightarrow r$, $(p \rightarrow q) \rightarrow (p \rightarrow q)$, …
- $p \rightarrow p$ 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 $\sigma \to \tau$ as well as σ
- A term may admit multiple types
    - $\lambda x \cdot x$ can be given types $p \to p$, $r \to r$, $(p \to q) \to (p \to q)$, $\ldots$
- $p \to p$ 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!