Typed lambda calculus

Madhavan Mukund, **S P Suresh**

Programming Language Concepts
Lecture 21, 3 April 2025

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?

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context
- **Haskell, ML, ...** the types of `m`, `n` to be fixed by the context. Types for `x`, `y` are flexible.

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context
- **Haskell, ML, ...** the types of `m`, `n` to be fixed by the context. Types for `x`, `y` are flexible.
    - **Polymorphic!**

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context
- **Haskell, ML, ...** the types of `m`, `n` to be fixed by the context. Types for `x`, `y` are flexible.
  - **Polymorphic!**
- **Pascal, C, most of Java, ...** specify all the types!

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context
- **Haskell, ML, ...** the types of `m`, `n` to be fixed by the context. Types for `x`, `y` are flexible.
  - **Polymorphic!**
- **Pascal, C, most of Java, ...** specify all the types!
- **Early versions of Fortran:** variables whose name begin with `I, J, K, L, M, N` are integers, other variables are floating-point numbers

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context
- **Haskell, ML, ...** the types of `m`, `n` to be fixed by the context. Types for `x`, `y` are flexible.
  - **Polymorphic!**
- **Pascal, C, most of Java, ...** specify all the types!
- **Early versions of Fortran:** variables whose name begin with `I, J, K, L, M, N` are integers, other variables are floating-point numbers
- **Church typing:** Pascal, C, Java, Fortran

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context
- **Haskell, ML, ...** the types of `m`, `n` to be fixed by the context. Types for `x`, `y` are flexible.
  - **Polymorphic!**
- **Pascal, C, most of Java, ...** specify all the types!
- **Early versions of Fortran:** variables whose name begin with `I, J, K, L, M, N` are integers, other variables are floating-point numbers
- **Church typing:** Pascal, C, Java, Fortran
- **Curry typing:** Haskell, ML

Styles of typing

- Consider a function with parameters `x`, `y`, and other variables `m`, `n` that are defined by the surrounding context
- **Haskell, ML, ...** the types of `m`, `n` to be fixed by the context. Types for `x`, `y` are flexible.
  - **Polymorphic!**
- **Pascal, C, most of Java, ...** specify all the types!
- **Early versions of Fortran:** variables whose name begin with `I, J, K, L, M, N` are integers, other variables are floating-point numbers
- **Church typing:** Pascal, C, Java, Fortran
- **Curry typing:** Haskell, ML
  - We will only learn Curry typing

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

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

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

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

  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
  - $p \rightarrow q$

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
  - $p \rightarrow q$
  - $p \rightarrow (q \rightarrow p)$
  - $(p \rightarrow r) \rightarrow r$

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 $p, q, r, p_1, q', \dots$
- No structured types (lists, tuples, ...) or user-defined types
- Function types arise naturally
  - $p \to q$
  - $p \to (q \to p)$
  - $(p \to r) \to r$
  - $(p \to p) \to (p \to q)$
- $\sigma, \tau, \dots$ 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: $\sigma \to \tau \to \vartheta$ is short for $\sigma \to (\tau \to \vartheta)$

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 $\Gamma = \{(x_1 : \sigma_1), \dots, (x_n : \sigma_n)\}$ 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
- The typing rules:

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

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 $\Gamma = \{(x_1 : \sigma_1), \dots, (x_n : \sigma_n)\}$ where the x_i are **distinct** variables, and the σ_i are types
- The typing rules:

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

- 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 $\Gamma = \{(x_1 : \sigma_1), \dots, (x_n : \sigma_n)\}$ where the x_i are **distinct** variables, and the σ_i are types
- The typing rules:

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

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

Curry typing: Examples

- $$\dfrac{x : p \vdash x : p}{\vdash \lambda x \cdot x : p \to p}\ \text{Abs}$$

Curry typing: Examples

- $$\dfrac{x : p \vdash x : p}{\vdash \lambda x \cdot x : p \rightarrow p}\ \text{Abs}$$

- $$\dfrac{\dfrac{x : p, y : q \vdash x : p}{x : p \vdash \lambda y \cdot x : q \rightarrow p}\ \text{Abs}}{\vdash \lambda xy \cdot x : p \rightarrow (q \rightarrow p)}\ \text{Abs}$$

Curry typing: Examples

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

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

Curry typing: Examples

- Let $\Gamma = \{f : q, x : p\}$

$$\dfrac{\dfrac{\Gamma \vdash x : p}{f : q \vdash \lambda x \cdot x : p \to p}\ \text{Abs}}{\vdash \lambda f\, x \cdot x : q \to (p \to p)}\ \text{Abs}$$

Curry typing: Examples

- Let $\Gamma = \{f : q, x : p\}$

$$\dfrac{\dfrac{\Gamma \vdash x : p}{f : q \vdash \lambda x \cdot x : p \to p}\ \text{Abs}}{\vdash \lambda f\, x \cdot x : q \to (p \to p)}\ \text{Abs}$$

- Let $\Delta = \{f : p \to p, x : p\}$

$$\dfrac{\dfrac{\Delta \vdash x : p}{f : p \to p \vdash \lambda x \cdot x : p \to p}\ \text{Abs}}{\vdash \lambda f\, x \cdot x : (p \to p) \to (p \to p)}\ \text{Abs}$$

Curry typing: Examples

- Let $\Gamma = \{f : p \rightarrow q, x : p\}$.

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash f : p \rightarrow q \qquad \Gamma \vdash x : p}{\Gamma \vdash f\,x : q}\ \text{App}}{f : p \rightarrow q \vdash \lambda x \cdot f\,x : p \rightarrow q}\ \text{Abs}}{\vdash \lambda f\,x \cdot f\,x : (p \rightarrow q) \rightarrow (p \rightarrow q)}\ \text{Abs}$$

Curry typing: Examples

- Let $\Gamma = \{f : p \rightarrow q, x : p\}$.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : p \rightarrow q \qquad \Gamma \vdash x : p}{\Gamma \vdash f\,x : q}\text{App}}{f : p \rightarrow q \vdash \lambda x \cdot f\,x : p \rightarrow q}\text{Abs}}{\vdash \lambda f\,x \cdot f\,x : (p \rightarrow q) \rightarrow (p \rightarrow q)}\text{Abs}$$

- Let $\Delta = \{f : p \rightarrow p, x : p\}$.

$$\dfrac{\dfrac{\dfrac{\Delta \vdash f : p \rightarrow q \qquad \Delta \vdash x : p}{\Gamma \vdash f\,x : q}\text{App}}{f : p \rightarrow p \vdash \lambda x \cdot f\,x : p \rightarrow p}\text{Abs}}{\vdash \lambda f\,x \cdot f\,x : (p \rightarrow p) \rightarrow (p \rightarrow p)}\text{Abs}$$

Curry typing: Examples

- Let $\Delta = \{f : p \to p, x : p\}$.

$$\dfrac{\dfrac{\dfrac{\Delta \vdash f : p \to p \qquad \dfrac{\Delta \vdash f : p \to p \qquad \Delta \vdash x : p}{\Delta \vdash f\,x : p}\;\text{App}}{\Delta \vdash f\,(f\,x) : p}\;\text{App}}{f : p \to p \vdash \lambda x \cdot f\,(f\,x) : p \to p}\;\text{Abs}}{\vdash \lambda f\,x \cdot f\,(f\,x) : (p \to p) \to (p \to p)}\;\text{Abs}$$

Curry typing: Examples

- Let $\Delta = \{f : p \to p, x : p\}$.

$$
\dfrac{\dfrac{\Delta \vdash f : p \to p \qquad \dfrac{\Delta \vdash f : p \to p \qquad \Delta \vdash x : p}{\Delta \vdash f\,x : p}\,\text{App}}{\Delta \vdash f\,(f\,x) : p}\,\text{App}}{\dfrac{f : p \to p \vdash \lambda x \cdot f\,(f\,x) : p \to p}{\vdash \lambda f\,x \cdot f\,(f\,x) : (p \to p) \to (p \to p)}\,\text{Abs}}\,\text{Abs}
$$

- Define **int** $:= (p \to p) \to (p \to p)$

Curry typing: Examples

- Let $\Delta = \{f : p \to p, x : p\}$.

$$\dfrac{\Delta \vdash f : p \to p \qquad \dfrac{\Delta \vdash f : p \to p \qquad \Delta \vdash x : p}{\Delta \vdash f\,x : p}\,\text{App}}{\dfrac{\Delta \vdash f\,(f\,x) : p}{\dfrac{f : p \to p \vdash \lambda x \cdot f\,(f\,x) : p \to p}{\vdash \lambda f\,x \cdot f\,(f\,x) : (p \to p) \to (p \to p)}\,\text{Abs}}\,\text{Abs}}\,\text{App}$$

- Define **int** $:= (p \to p) \to (p \to p)$
- For all $n \in \mathbb{N}$, $\vdash$ «n» : **int**

Curry typing: Examples

- Recall that **succ** $:= \lambda m\, f\, x \cdot f\ (m\, f\, x)$

Curry typing: Examples

- Recall that **succ** $:= \lambda m\, f\, x \cdot f\ (m\, f\, x)$
- **succ** can be given the type **int** $\rightarrow$ **int**

Curry typing: Examples

- Recall that **succ** $:= \lambda m\,f\,x \cdot f\,(m\,f\,x)$
- **succ** can be given the type **int** $\to$ **int**
- Let $\Gamma = \{m : \mathbf{int}, f : p \to p, x : p\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : p \to p \qquad \dfrac{\dfrac{\Gamma \vdash m : (p \to p) \to (p \to p) \qquad \Gamma \vdash f : p \to p}{\Gamma \vdash m\,f : p \to p}\,\text{App} \qquad \Gamma \vdash x : p}{\Gamma \vdash m\,f\,x : p}\,\text{App}}{\Gamma \vdash f\,(m\,f\,x) : p}\,\text{App}}{m : \mathbf{int}, f : p \to p \vdash \lambda x \cdot f\,(m\,f\,x) : p \to p}\,\text{Abs}}{\dfrac{m : \mathbf{int} \vdash \lambda f\,x \cdot f\,(m\,f\,x) : \mathbf{int}}{\vdash \lambda m\,f\,x \cdot f\,(m\,f\,x) : \mathbf{int} \to \mathbf{int}}\,\text{Abs}}\,\text{Abs}$$

Curry typing: Examples

- Similarly $\textbf{plus} : \textbf{int} \to \textbf{int} \to \textbf{int}$ and $\textbf{mult} : \textbf{int} \to \textbf{int} \to \textbf{int}$

Curry typing: Examples

- Similarly $\mathbf{plus} : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ and $\mathbf{mult} : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$
- But one cannot assign type $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ to $\mathbf{exp} := \lambda m\, n \cdot m\, n$

Curry typing: Examples

- Similarly $\textbf{plus} : \textbf{int} \rightarrow \textbf{int} \rightarrow \textbf{int}$ and $\textbf{mult} : \textbf{int} \rightarrow \textbf{int} \rightarrow \textbf{int}$
- But one cannot assign type $\textbf{int} \rightarrow \textbf{int} \rightarrow \textbf{int}$ to $\textbf{exp} := \lambda m\, n \cdot m\, n$
- For the above typing to be possible, we must have $m : \textbf{int}, n : \textbf{int} \vdash m\, n : \textbf{int}$

Curry typing: Examples

- Similarly $\textbf{plus} : \textbf{int} \to \textbf{int} \to \textbf{int}$ and $\textbf{mult} : \textbf{int} \to \textbf{int} \to \textbf{int}$
- But one cannot assign type $\textbf{int} \to \textbf{int} \to \textbf{int}$ to $\textbf{exp} := \lambda m\, n \cdot m\, n$
- For the above typing to be possible, we must have $m : \textbf{int}, n : \textbf{int} \vdash m\, n : \textbf{int}$
- But this is possible only if $m : \textbf{int}, n : \textbf{int} \vdash m : \textbf{int} \to \textbf{int}$ is derivable

Curry typing: Examples

- Similarly $\mathbf{plus} : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ and $\mathbf{mult} : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$
- But one cannot assign type $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ to $\mathbf{exp} := \lambda m\, n \cdot m\, n$
- For the above typing to be possible, we must have $m : \mathbf{int}, n : \mathbf{int} \vdash m\, n : \mathbf{int}$
- But this is possible only if $m : \mathbf{int}, n : \mathbf{int} \vdash m : \mathbf{int} \rightarrow \mathbf{int}$ is derivable
- Not possible!

Curry typing: Examples

- Similarly $\textbf{plus} : \textbf{int} \rightarrow \textbf{int} \rightarrow \textbf{int}$ and $\textbf{mult} : \textbf{int} \rightarrow \textbf{int} \rightarrow \textbf{int}$
- But one cannot assign type $\textbf{int} \rightarrow \textbf{int} \rightarrow \textbf{int}$ to $\textbf{exp} := \lambda m\, n \cdot m\, n$
- For the above typing to be possible, we must have $m : \textbf{int}, n : \textbf{int} \vdash m\, n : \textbf{int}$
- But this is possible only if $m : \textbf{int}, n : \textbf{int} \vdash m : \textbf{int} \rightarrow \textbf{int}$ is derivable
- Not possible!
- But we can derive the judgement «m» «n» $: \textbf{int}$

Curry typing: Examples

- Similarly **plus** $: \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ and **mult** $: \mathbf{int} \to \mathbf{int} \to \mathbf{int}$
- But one cannot assign type $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$ to **exp** $:= \lambda m\, n \cdot m\, n$
- For the above typing to be possible, we must have $m : \mathbf{int}, n : \mathbf{int} \vdash m\, n : \mathbf{int}$
- But this is possible only if $m : \mathbf{int}, n : \mathbf{int} \vdash m : \mathbf{int} \to \mathbf{int}$ is derivable
- Not possible!
- But we can derive the judgement $«m» «n» : \mathbf{int}$
- For example, letting $\tau := p \to p$,

$$\frac{\vdash «2» : (\tau \to \tau) \to (\tau \to \tau) \qquad \vdash «2» : (p \to p) \to (p \to p)}{\vdash «2» «2» : \mathbf{int}}\ \text{App}$$

Defining arithmetic functions in typed λ-calculus

- A function $f : \mathbb{N}^k \to \mathbb{N}$ is **defined** in the typed λ-calculus if there is a term F such that:

Defining arithmetic functions in typed λ-calculus

- A function $f : \mathbb{N}^k \to \mathbb{N}$ is **defined** in the typed λ-calculus if there is a term F such that:
  - $\vdash F : \textbf{int} \to \textbf{int} \to \cdots \to \textbf{int}$ ($\textbf{int}$ occurring $k + 1$ times)

Defining arithmetic functions in typed λ-calculus

- A function $f : \mathbb{N}^k \to \mathbb{N}$ is **defined** in the typed λ-calculus if there is a term F such that:
  - $\vdash F : \textbf{int} \to \textbf{int} \to \cdots \to \textbf{int}$ (**int** occurring $k+1$ times)
  - for all $m_1, \ldots, m_k, n \in \mathbb{N}$: $f(m_1, \ldots, m_k) = n$ iff $F \ll m_1 \gg \cdots \ll m_k \gg \xrightarrow{*} \ll n \gg$

Defining arithmetic functions in typed λ-calculus

- A function $f : \mathbb{N}^k \to \mathbb{N}$ is **defined** in the typed λ-calculus if there is a term F such that:
  - $\vdash F : \textbf{int} \to \textbf{int} \to \cdots \to \textbf{int}$ (**int** occurring $k + 1$ times)
  - for all $m_1, \ldots, m_k, n \in \mathbb{N}$: $f(m_1, \ldots, m_k) = n$ iff $F \text{«} m_1 \text{»} \cdots \text{«} m_k \text{»} \xrightarrow{*} \text{«} n \text{»}$
- f is definable in typed λ-calculus iff it is essentially a polynomial function!

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 $\xrightarrow{*}_\beta$ as usual

Typed λ-calculus: Church-Rosser

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

Typed λ-calculus: 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
  - 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 $\sigma \rightarrow \tau$ 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 $\sigma \rightarrow \tau$ as well as σ
- A term may admit multiple types
  - $\lambda x \cdot x$ can be given types $p \rightarrow p$, $r \rightarrow r$, $(p \rightarrow q) \rightarrow (p \rightarrow q)$, ...

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 $p \longrightarrow p$, $r \longrightarrow r$, $(p \longrightarrow q) \longrightarrow (p \longrightarrow q)$, ...
- $p \longrightarrow p$ 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 $\sigma \to \tau$ 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
- **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 $\sigma \rightarrow \tau$ 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
  - unique for each typable term – modulo renaming of variables!