The λ-calculus: from simple types to non-idempotent intersection types

Day 3: The untyped λ-calculus

Giulio Guerrieri

Department of Informatics, University of Sussex (Brighton, UK)
✉ g.guerrieri@sussex.ac.uk https://pageperso.lis-lab.fr/~giulio.guerrieri/

37th Escuela de Ciencias Informáticas (ECI 2024)
Buenos Aires (Argentina), 31 July 2024

Outline

Outline

The λ-calculus beyond simple types

Term and β-reduction of the simply typed λ-calculus can be defined without types.
$\rightsquigarrow$ Let us explore the word of the λ-calculus without types.

1. What do we gain?
2. What do we lose?

We can freely apply s to t to get st, without requiring $s : A \Rightarrow B$ or $t : A$.

Consider the term $\lambda x.xx$. It not a term for the simply typed λ-calculus.

- Why is there no A such that $\vdash \lambda x.xx : A$ is derivable?
- $(\lambda x.xx)(\lambda x.xx) \to_\beta (xx)\{\lambda x.xx/x\} = (\lambda x.xx)(\lambda x.xx) \to_\beta \dots$ (normalization fails).

The λ-calculus beyond simple types

Term and β-reduction of the simply typed λ-calculus can be defined without types.
$\rightsquigarrow$ Let us explore the word of the λ-calculus without types.

1. What do we gain?
2. What do we lose?

We can freely apply s to t to get st, without requiring $s : A \Rightarrow B$ or $t : A$.

Consider the term $\lambda x.xx$. It not a term for the simply typed λ-calculus.

- Why is there no A such that $\vdash \lambda x.xx : A$ is derivable?
- $(\lambda x.xx)(\lambda x.xx) \to_\beta (xx)\{\lambda x.xx/x\} = (\lambda x.xx)(\lambda x.xx) \to_\beta \dots$ (normalization fails).

The λ-calculus beyond simple types

Term and β-reduction of the simply typed λ-calculus can be defined without types.
$\rightsquigarrow$ Let us explore the word of the λ-calculus without types.

1. What do we gain?
2. What do we lose?

We can freely apply s to t to get st, without requiring $s : A \Rightarrow B$ or $t : A$.

Consider the term $\lambda x.xx$. It not a term for the simply typed λ-calculus.

- Why is there no A such that $\vdash \lambda x.xx : A$ is derivable?
- $(\lambda x.xx)(\lambda x.xx) \to_\beta (xx)\{\lambda x.xx/x\} = (\lambda x.xx)(\lambda x.xx) \to_\beta \dots$ (normalization fails).

The λ-calculus beyond simple types

Term and β-reduction of the simply typed λ-calculus can be defined without types.
$\rightsquigarrow$ Let us explore the word of the λ-calculus without types.

1. What do we gain?
2. What do we lose?

We can freely apply s to t to get st, without requiring $s : A \Rightarrow B$ or $t : A$.

Consider the term $\lambda x.xx$. It not a term for the simply typed λ-calculus.

- Why is there no A such that $\vdash \lambda x.xx : A$ is derivable?
- $(\lambda x.xx)(\lambda x.xx) \to_\beta (xx)\{\lambda x.xx/x\} = (\lambda x.xx)(\lambda x.xx) \to_\beta \ldots$ (normalization fails).

The untyped λ-calculus: the philosophy

The functions can be treated anonymously, that is without giving them a name:

$$\mathrm{id}(x) = x \quad \rightsquigarrow \quad x \mapsto x \qquad\qquad \mathrm{sq\_sum}(x, y) = x^2 + y^2 \quad \rightsquigarrow \quad (x, y) \mapsto x^2 + y^2$$

Functions of several arguments can be transformed into function of a single argument:

$$(x, y) \mapsto x^2 + y^2 \qquad \rightsquigarrow \qquad x \mapsto (y \mapsto x^2 + y^2) \qquad\qquad (\text{currying})$$

Functions can be applied to functions and can return functions (higher-order):

$$(x \mapsto x)5 = 5 \qquad\qquad (x \mapsto x)(y \mapsto y^2) = y \mapsto y^2$$

The untyped λ-calculus performs higher-order computation:

- everything is an anonymous function with a single argument (λ-calculus);
- functions can be applied to other functions without any restriction. (untyped)

The untyped λ-calculus: the philosophy

The functions can be treated anonymously, that is without giving them a name:

$$\mathrm{id}(x) = x \quad \rightsquigarrow \quad x \mapsto x \qquad\qquad \mathrm{sq\_sum}(x, y) = x^2 + y^2 \quad \rightsquigarrow \quad (x, y) \mapsto x^2 + y^2$$

Functions of several arguments can be transformed into function of a single argument:

$$(x, y) \mapsto x^2 + y^2 \quad \rightsquigarrow \quad x \mapsto (y \mapsto x^2 + y^2) \qquad\qquad (\text{currying})$$

Functions can be applied to functions and can return functions (higher-order):

$$(x \mapsto x)5 = 5 \qquad\qquad (x \mapsto x)(y \mapsto y^2) = y \mapsto y^2$$

The untyped λ-calculus performs higher-order computation:

- everything is an anonymous function with a single argument (λ-calculus);
- functions can be applied to other functions without any restriction. (untyped)

The untyped λ-calculus: the philosophy

The functions can be treated anonymously, that is without giving them a name:

$$\mathrm{id}(x) = x \quad \rightsquigarrow \quad x \mapsto x \qquad\qquad \mathrm{sq\_sum}(x, y) = x^2 + y^2 \quad \rightsquigarrow \quad (x, y) \mapsto x^2 + y^2$$

Functions of several arguments can be transformed into function of a single argument:

$$(x, y) \mapsto x^2 + y^2 \quad \rightsquigarrow \quad x \mapsto (y \mapsto x^2 + y^2) \qquad\qquad \text{(currying)}$$

Functions can be applied to functions and can return functions (higher-order):

$$(x \mapsto x)5 = 5 \qquad\qquad (x \mapsto x)(y \mapsto y^2) = y \mapsto y^2$$

The untyped λ-calculus performs higher-order computation:

- everything is an anonymous function with a single argument (λ-calculus);
- functions can be applied to other functions without any restriction. (untyped)

The untyped λ-calculus: the philosophy

The functions can be treated anonymously, that is without giving them a name:

$$\mathrm{id}(x) = x \quad \rightsquigarrow \quad x \mapsto x \qquad\qquad \mathrm{sq\_sum}(x, y) = x^2 + y^2 \quad \rightsquigarrow \quad (x, y) \mapsto x^2 + y^2$$

Functions of several arguments can be transformed into function of a single argument:

$$(x, y) \mapsto x^2 + y^2 \qquad \rightsquigarrow \qquad x \mapsto (y \mapsto x^2 + y^2) \qquad\qquad (\text{currying})$$

Functions can be applied to functions and can return functions (higher-order):

$$(x \mapsto x)5 = 5 \qquad\qquad (x \mapsto x)(y \mapsto y^2) = y \mapsto y^2$$

The untyped λ-calculus performs higher-order computation:

- everything is an anonymous function with a single argument (λ-calculus);
- functions can be applied to other functions without any restriction. (untyped)

The untyped λ-calculus, formally

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application) Rmk: stu stands for $(st)u$.

The free variables of a term t are the variables that are not bound to a λ. Formally,

$$\mathrm{fv}(x) = \{x\} \qquad \mathrm{fv}(st) = \mathrm{fv}(s) \cup \mathrm{fv}(t) \qquad \mathrm{fv}(\lambda x.t) = \mathrm{fv}(t) \setminus \{x\}$$

Terms are identified up to renaming of bound variables (α-equivalence), e.g. $\lambda x.x = \lambda y.y$

β-reduction ($t\{s/x\}$ is the capture-avoiding substitution of s for the free occurrences of x in t):

(the term on the left is a β-redex) $(\lambda x.t)s \to_\beta t\{s/x\}$

Substitution $t\{s/x\}$ should be defined carefully to avoid capture of variables.

$$(\lambda x.yx)\{x/y\} \neq \lambda x.xx \qquad \text{but} \qquad (\lambda x.yx)\{x/y\} = (\lambda z.yz)\{x/y\} = \lambda z.xz$$

To write $t\{s/x\}$, first take t such that its bound variables are not in $\mathrm{fv}(s)$ then substitute.

The untyped λ-calculus, formally

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application) Rmk: stu stands for $(st)u$.

The free variables of a term t are the variables that are not bound to a λ. Formally,

$$\mathrm{fv}(x) = \{x\} \qquad \mathrm{fv}(st) = \mathrm{fv}(s) \cup \mathrm{fv}(t) \qquad \mathrm{fv}(\lambda x.t) = \mathrm{fv}(t) \setminus \{x\}$$

Terms are identified up to renaming of bound variables (α-equivalence), e.g. $\lambda x.x = \lambda y.y$

β-reduction ($t\{s/x\}$ is the capture-avoiding substitution of s for the free occurrences of x in t):

(the term on the left is a β-redex) $(\lambda x.t)s \to_\beta t\{s/x\}$

Substitution $t\{s/x\}$ should be defined carefully to avoid capture of variables.

$$(\lambda x.yx)\{x/y\} \neq \lambda x.xx \qquad \text{but} \qquad (\lambda x.yx)\{x/y\} = (\lambda z.yz)\{x/y\} = \lambda z.xz$$

To write $t\{s/x\}$, first take t such that its bound variables are not in $\mathrm{fv}(s)$ then substitute.

The untyped λ-calculus, formally

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application) Rmk: stu stands for $(st)u$.

The free variables of a term t are the variables that are not bound to a λ. Formally,

$$\mathsf{fv}(x) = \{x\} \qquad \mathsf{fv}(st) = \mathsf{fv}(s) \cup \mathsf{fv}(t) \qquad \mathsf{fv}(\lambda x.t) = \mathsf{fv}(t) \setminus \{x\}$$

Terms are identified up to renaming of bound variables (α-equivalence), e.g. $\lambda x.x = \lambda y.y$

β-reduction ($t\{s/x\}$ is the capture-avoiding substitution of s for the free occurrences of x in t):

(the term on the left is a β-redex) $(\lambda x.t)s \to_\beta t\{s/x\}$

Substitution $t\{s/x\}$ should be defined carefully to avoid capture of variables.

$$(\lambda x.yx)\{x/y\} \neq \lambda x.xx \qquad \text{but} \qquad (\lambda x.yx)\{x/y\} = (\lambda z.yz)\{x/y\} = \lambda z.xz$$

To write $t\{s/x\}$, first take t such that its bound variables are not in $\mathsf{fv}(s)$ then substitute.

The untyped λ-calculus, formally

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application)   Rmk: stu stands for $(st)u$.

The free variables of a term t are the variables that are not bound to a λ. Formally,

$$\mathsf{fv}(x) = \{x\} \qquad \mathsf{fv}(st) = \mathsf{fv}(s) \cup \mathsf{fv}(t) \qquad \mathsf{fv}(\lambda x.t) = \mathsf{fv}(t) \setminus \{x\}$$

Terms are identified up to renaming of bound variables (α-equivalence), e.g. $\lambda x.x = \lambda y.y$

β-reduction ($t\{s/x\}$ is the capture-avoiding substitution of s for the free occurrences of x in t):

(the term on the left is a β-redex) $(\lambda x.t)s \to_\beta t\{s/x\}$

Substitution $t\{s/x\}$ should be defined carefully to avoid capture of variables.

$$(\lambda x.yx)\{x/y\} \neq \lambda x.xx \qquad \text{but} \qquad (\lambda x.yx)\{x/y\} = (\lambda z.yz)\{x/y\} = \lambda z.xz$$

To write $t\{s/x\}$, first take t such that its bound variables are not in $\mathsf{fv}(s)$ then substitute.

The untyped λ-calculus, formally

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application)   Rmk: stu stands for $(st)u$.

The free variables of a term t are the variables that are not bound to a λ. Formally,

$$\mathsf{fv}(x) = \{x\} \qquad \mathsf{fv}(st) = \mathsf{fv}(s) \cup \mathsf{fv}(t) \qquad \mathsf{fv}(\lambda x.t) = \mathsf{fv}(t) \setminus \{x\}$$

Terms are identified up to renaming of bound variables (α-equivalence), e.g. $\lambda x.x = \lambda y.y$

β-reduction ($t\{s/x\}$ is the capture-avoiding substitution of s for the free occurrences of x in t):

(the term on the left is a β-redex) $(\lambda x.t)s \to_\beta t\{s/x\}$

Substitution $t\{s/x\}$ should be defined carefully to avoid capture of variables.

$$(\lambda x.yx)\{x/y\} \neq \lambda x.xx \qquad \text{but} \qquad (\lambda x.yx)\{x/y\} = (\lambda z.yz)\{x/y\} = \lambda z.xz$$

To write $t\{s/x\}$, first take t such that its bound variables are not in $\mathsf{fv}(s)$ then substitute.

The untyped λ-calculus, formally

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application) Rmk: stu stands for $(st)u$.

The free variables of a term t are the variables that are not bound to a λ. Formally,

$$\mathrm{fv}(x) = \{x\} \qquad \mathrm{fv}(st) = \mathrm{fv}(s) \cup \mathrm{fv}(t) \qquad \mathrm{fv}(\lambda x.t) = \mathrm{fv}(t) \setminus \{x\}$$

Terms are identified up to renaming of bound variables (α-equivalence), e.g. $\lambda x.x = \lambda y.y$

β-reduction ($t\{s/x\}$ is the capture-avoiding substitution of s for the free occurrences of x in t):

(the term on the left is a β-redex) $(\lambda x.t)s \to_\beta t\{s/x\}$

Substitution $t\{s/x\}$ should be defined carefully to avoid capture of variables.

$$(\lambda x.yx)\{x/y\} \neq \lambda x.xx \qquad \text{but} \qquad (\lambda x.yx)\{x/y\} = (\lambda z.yz)\{x/y\} = \lambda z.xz$$

To write $t\{s/x\}$, first take t such that its bound variables are not in $\mathrm{fv}(s)$ then substitute.

The structure of a term

Rmk. Every term s can be written in a unique way as

$$s = \lambda x_1 \dots \lambda x_n . h t_1 \dots t_m \qquad \text{with } m, n \in \mathbb{N}$$

where h (the head of s) is either a variable (head variable) or a β-redex (head β-redex).

In a tree representation:

Compare this tree with a derivation in natural deduction. Similarities? Differences?

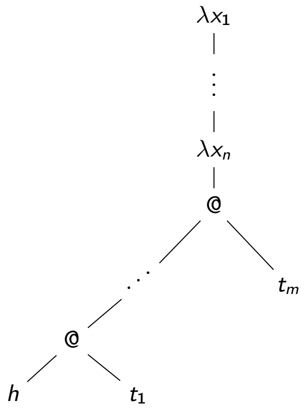
The structure of a term

Rmk. Every term s can be written in a unique way as

$$s = \lambda x_1 \dots \lambda x_n . h t_1 \dots t_m \qquad \text{with } m, n \in \mathbb{N}$$

where h (the head of s) is either a variable (head variable) or a β-redex (head β-redex).

In a tree representation:



Compare this tree with a derivation in natural deduction. Similarities? Differences?

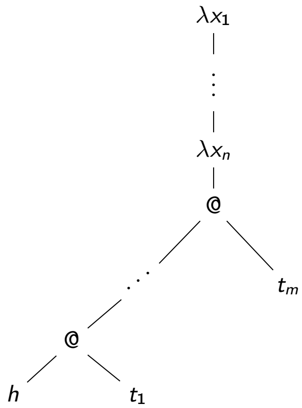
The structure of a term

Rmk. Every term s can be written in a unique way as

$$s = \lambda x_1 \dots \lambda x_n . h t_1 \dots t_m \qquad \text{with } m, n \in \mathbb{N}$$

where h (the head of s) is either a variable (head variable) or a β-redex (head β-redex).

In a tree representation:



Compare this tree with a derivation in natural deduction. Similarities? Differences?

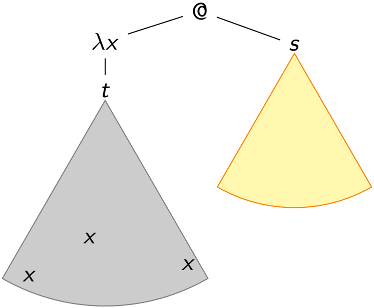
β-reduction from a graphical point of view

$$(\lambda x.t)s \to_\beta t\{s/x\}$$

Compare this figure with the cut-elimination step for natural deduction (see Day 1).

β-reduction from a graphical point of view

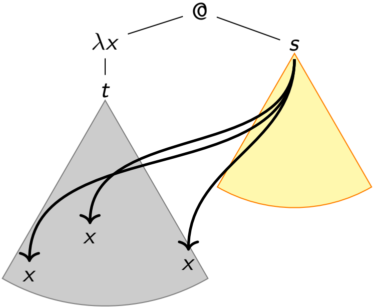
$$(\lambda x.t)s \to_\beta t\{s/x\}$$



Compare this figure with the cut-elimination step for natural deduction (see Day 1).

β-reduction from a graphical point of view

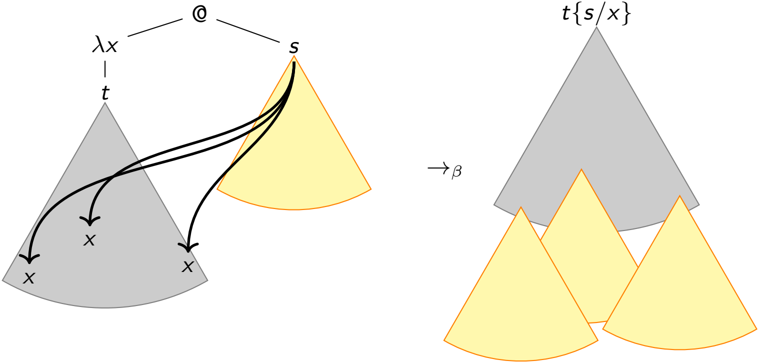
$$(\lambda x.t)s \to_\beta t\{s/x\}$$



Compare this figure with the cut-elimination step for natural deduction (see Day 1).

β-reduction from a graphical point of view

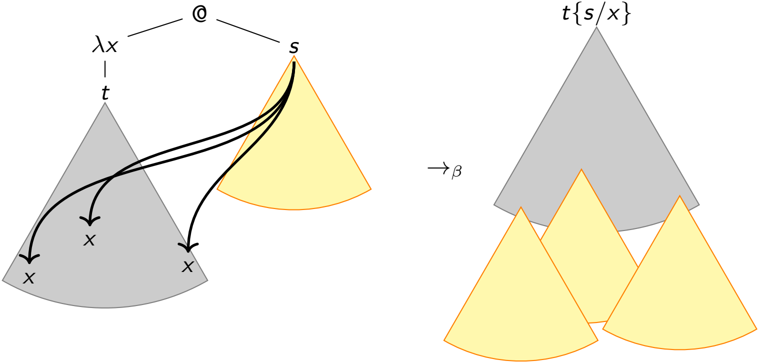
$$(\lambda x.t)s \to_\beta t\{s/x\}$$



Compare this figure with the cut-elimination step for natural deduction (see Day 1).

β-reduction from a graphical point of view

$$(\lambda x.t)s \to_\beta t\{s/x\}$$



Compare this figure with the cut-elimination step for natural deduction (see Day 1).

Different notions of reduction

(Full) β-reduction $\to_\beta$ fires a β-redex anywhere in a term. Formally,

$$\frac{}{(\lambda x.t)s \to_\beta t\{s/x\}} \qquad \frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'} \qquad \frac{t \to_\beta t'}{ts \to_\beta t's} \qquad \frac{t \to_\beta t'}{st \to_\beta st'}$$

Head β-reduction $\to_{h\beta}$ fires a β-redex only in the "head" of a term. Formally,

$$\frac{}{(\lambda x.t)s \to_{h\beta} t\{s/x\}} \qquad \frac{t \to_{h\beta} t'}{\lambda x.t \to_{h\beta} \lambda x.t'} \qquad \frac{t \to_{h\beta} t' \quad t \neq \lambda x.r}{ts \to_{h\beta} t's}$$

Leftmost-outermost β-reduction $\to_{\ell\beta}$ fires the leftmost-outermost β-redex in a term.

$$\frac{}{(\lambda x.t)s \to_{\ell\beta} t\{s/x\}} \qquad \frac{t \to_{\ell\beta} t'}{\lambda x.t \to_{\ell\beta} \lambda x.t'} \qquad \frac{t \to_{\ell\beta} t' \quad t \neq \lambda x.r}{ts \to_{\ell\beta} t's} \qquad \frac{t \to_{\ell\beta} t' \quad s \text{ neutral}}{st \to_{\ell\beta} st'}$$

where neutral means $s = xs_1 \dots x_n$ and $s_1, \dots, s_n$ normal, for some $n \in \mathbb{N}$.

Rmk. $\to_{h\beta} \subsetneq \to_{l\beta} \subsetneq \to_\beta$. For strictness, consider $I = \lambda x.x$ and $t = (Ix)(Iy)(Iz)$. Then,

- $t \to_{h\beta} x(Iy)(Iz)$ but $t \not\to_{h\beta} (Ix)y(Iz)$ and $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{\ell\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{\ell\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ and $x(Iy)(Iz) \to_\beta x(Iy)z$.

Different notions of reduction

(Full) β-reduction $\to_\beta$ fires a β-redex anywhere in a term. Formally,

$$\frac{}{(\lambda x.t)s \to_\beta t\{s/x\}} \qquad \frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'} \qquad \frac{t \to_\beta t'}{ts \to_\beta t's} \qquad \frac{t \to_\beta t'}{st \to_\beta st'}$$

Head β-reduction $\to_{h\beta}$ fires a β-redex only in the "head" of a term. Formally,

$$\frac{}{(\lambda x.t)s \to_{h\beta} t\{s/x\}} \qquad \frac{t \to_{h\beta} t'}{\lambda x.t \to_{h\beta} \lambda x.t'} \qquad \frac{t \to_{h\beta} t' \quad t \neq \lambda x.r}{ts \to_{h\beta} t's}$$

Leftmost-outermost β-reduction $\to_{\ell\beta}$ fires the leftmost-outermost β-redex in a term.

$$\frac{}{(\lambda x.t)s \to_{\ell\beta} t\{s/x\}} \qquad \frac{t \to_{\ell\beta} t'}{\lambda x.t \to_{\ell\beta} \lambda x.t'} \qquad \frac{t \to_{\ell\beta} t' \quad t \neq \lambda x.r}{ts \to_{\ell\beta} t's} \qquad \frac{t \to_{\ell\beta} t' \quad s \text{ neutral}}{st \to_{\ell\beta} st'}$$

where neutral means $s = xs_1 \dots x_n$ and $s_1, \dots, s_n$ normal, for some $n \in \mathbb{N}$.

Rmk. $\to_{h\beta} \subsetneq \to_{l\beta} \subsetneq \to_\beta$. For strictness, consider $I = \lambda x.x$ and $t = (Ix)(Iy)(Iz)$. Then,

- $t \to_{h\beta} x(Iy)(Iz)$ but $t \not\to_{h\beta} (Ix)y(Iz)$ and $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{\ell\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{\ell\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ and $x(Iy)(Iz) \to_\beta x(Iy)z$.

Different notions of reduction

(Full) β-reduction $\to_\beta$ fires a β-redex anywhere in a term. Formally,

$$\frac{}{(\lambda x.t)s \to_\beta t\{s/x\}} \qquad \frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'} \qquad \frac{t \to_\beta t'}{ts \to_\beta t's} \qquad \frac{t \to_\beta t'}{st \to_\beta st'}$$

Head β-reduction $\to_{h\beta}$ fires a β-redex only in the "head" of a term. Formally,

$$\frac{}{(\lambda x.t)s \to_{h\beta} t\{s/x\}} \qquad \frac{t \to_{h\beta} t'}{\lambda x.t \to_{h\beta} \lambda x.t'} \qquad \frac{t \to_{h\beta} t' \qquad t \neq \lambda x.r}{ts \to_{h\beta} t's}$$

Leftmost-outermost β-reduction $\to_{\ell\beta}$ fires the leftmost-outermost β-redex in a term.

$$\frac{}{(\lambda x.t)s \to_{\ell\beta} t\{s/x\}} \qquad \frac{t \to_{\ell\beta} t'}{\lambda x.t \to_{\ell\beta} \lambda x.t'} \qquad \frac{t \to_{\ell\beta} t' \qquad t \neq \lambda x.r}{ts \to_{\ell\beta} t's} \qquad \frac{t \to_{\ell\beta} t' \quad s \text{ neutral}}{st \to_{\ell\beta} st'}$$

where neutral means $s = xs_1 \dots x_n$ and $s_1, \dots, s_n$ normal, for some $n \in \mathbb{N}$.

Rmk. $\to_{h\beta} \subsetneq \to_{l\beta} \subsetneq \to_\beta$. For strictness, consider $I = \lambda x.x$ and $t = (Ix)(Iy)(Iz)$. Then,

- $t \to_{h\beta} x(Iy)(Iz)$ but $t \not\to_{h\beta} (Ix)y(Iz)$ and $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{\ell\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{\ell\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ and $x(Iy)(Iz) \to_\beta x(Iy)z$.

Different notions of reduction

(Full) β-reduction $\to_\beta$ fires a β-redex anywhere in a term. Formally,

$$\frac{}{(\lambda x.t)s \to_\beta t\{s/x\}} \qquad \frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'} \qquad \frac{t \to_\beta t'}{ts \to_\beta t's} \qquad \frac{t \to_\beta t'}{st \to_\beta st'}$$

Head β-reduction $\to_{h\beta}$ fires a β-redex only in the "head" of a term. Formally,

$$\frac{}{(\lambda x.t)s \to_{h\beta} t\{s/x\}} \qquad \frac{t \to_{h\beta} t'}{\lambda x.t \to_{h\beta} \lambda x.t'} \qquad \frac{t \to_{h\beta} t' \quad t \neq \lambda x.r}{ts \to_{h\beta} t's}$$

Leftmost-outermost β-reduction $\to_{\ell\beta}$ fires the leftmost-outermost β-redex in a term.

$$\frac{}{(\lambda x.t)s \to_{\ell\beta} t\{s/x\}} \qquad \frac{t \to_{\ell\beta} t'}{\lambda x.t \to_{\ell\beta} \lambda x.t'} \qquad \frac{t \to_{\ell\beta} t' \quad t \neq \lambda x.r}{ts \to_{\ell\beta} t's} \qquad \frac{t \to_{\ell\beta} t' \quad s \text{ neutral}}{st \to_{\ell\beta} st'}$$

where neutral means $s = xs_1 \dots x_n$ and $s_1, \dots, s_n$ normal, for some $n \in \mathbb{N}$.

Rmk. $\to_{h\beta} \subsetneq \to_{l\beta} \subsetneq \to_\beta$. For strictness, consider $I = \lambda x.x$ and $t = (Ix)(Iy)(Iz)$. Then,

- $t \to_{h\beta} x(Iy)(Iz)$ but $t \not\to_{h\beta} (Ix)y(Iz)$ and $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{\ell\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{\ell\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ and $x(Iy)(Iz) \to_\beta x(Iy)z$.

Different notions of reduction

(Full) β-reduction $\to_\beta$ fires a β-redex anywhere in a term. Formally,

$$\frac{}{(\lambda x.t)s \to_\beta t\{s/x\}} \qquad \frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'} \qquad \frac{t \to_\beta t'}{ts \to_\beta t's} \qquad \frac{t \to_\beta t'}{st \to_\beta st'}$$

Head β-reduction $\to_{h\beta}$ fires a β-redex only in the "head" of a term. Formally,

$$\frac{}{(\lambda x.t)s \to_{h\beta} t\{s/x\}} \qquad \frac{t \to_{h\beta} t'}{\lambda x.t \to_{h\beta} \lambda x.t'} \qquad \frac{t \to_{h\beta} t' \quad t \neq \lambda x.r}{ts \to_{h\beta} t's}$$

Leftmost-outermost β-reduction $\to_{\ell\beta}$ fires the leftmost-outermost β-redex in a term.

$$\frac{}{(\lambda x.t)s \to_{\ell\beta} t\{s/x\}} \qquad \frac{t \to_{\ell\beta} t'}{\lambda x.t \to_{\ell\beta} \lambda x.t'} \qquad \frac{t \to_{\ell\beta} t' \quad t \neq \lambda x.r}{ts \to_{\ell\beta} t's} \qquad \frac{t \to_{\ell\beta} t' \quad s \text{ neutral}}{st \to_{\ell\beta} st'}$$

where neutral means $s = xs_1 \dots x_n$ and $s_1, \dots, s_n$ normal, for some $n \in \mathbb{N}$.

Rmk. $\to_{h\beta} \subsetneq \to_{l\beta} \subsetneq \to_\beta$. For strictness, consider $I = \lambda x.x$ and $t = (Ix)(Iy)(Iz)$. Then,

- $t \to_{h\beta} x(Iy)(Iz)$ but $t \not\to_{h\beta} (Ix)y(Iz)$ and $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{\ell\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{\ell\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ and $x(Iy)(Iz) \to_\beta x(Iy)z$.

Properties of different reductions

Rmk. Reductions $\to_{h\beta}$ and $\to_{\ell\beta}$ are deterministic (they can fire at most one redex). So:

$$\text{If } t \to_r s_1 \text{ and } t \to_r s_2 \text{ then } s_1 = s_2, \text{ for } r \in \{h\beta, l\beta\}.$$

Reduction $\to_\beta$ is not deterministic, it chooses among several β-redexes to fire in a term.

$$\overbrace{(\lambda x.xx)(\underbrace{(\lambda z.z)y}_{\text{inner } \beta\text{-redex}})}^{\text{outermost } \beta\text{-redex}} \to_\beta ((\lambda z.z)y)((\lambda z.z)y)$$

$$(\lambda x.xx)((\lambda z.z)y) \to_\beta (\lambda x.xx)z$$

Notation. $t \to^* s$ means that $t = t_0 \overbrace{\to t_1 \to \cdots \to}^{\text{for some } n \in \mathbb{N}} t_n = s$ (in particular, $t = s$ for $n = 0$).

Theorem (Confluence)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$, then there is a term r such that $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$.

Def. Let $r \in \{\beta, \ell\beta, h\beta\}$. A term t is r-normal if there is no s such that $t \to_r s$.

Corollary (Uniqueness of normal form)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$ where s_1 and s_2 are β-normal, then $s_1 = s_2$.

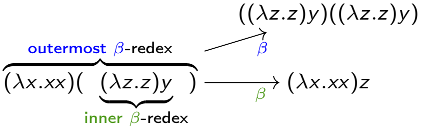
Proof. By confluence, $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$ for some r. By normality, $s_1 = r = s_2$. □

Properties of different reductions

Rmk. Reductions $\to_{h\beta}$ and $\to_{\ell\beta}$ are deterministic (they can fire at most one redex). So:

$$\text{If } t \to_r s_1 \text{ and } t \to_r s_2 \text{ then } s_1 = s_2, \text{ for } r \in \{h\beta, l\beta\}.$$

Reduction $\to_\beta$ is not deterministic, it chooses among several β-redexes to fire in a term.



Notation. $t \to^* s$ means that $t = t_0 \overbrace{\to t_1 \to \cdots \to}^{\text{for some } n \in \mathbb{N}} t_n = s$ (in particular, $t = s$ for $n = 0$).

Theorem (Confluence)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$, then there is a term r such that $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$.

Def. Let $r \in \{\beta, \ell\beta, h\beta\}$. A term t is r-normal if there is no s such that $t \to_r s$.

Corollary (Uniqueness of normal form)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$ where s_1 and s_2 are β-normal, then $s_1 = s_2$.

Proof. By confluence, $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$ for some r. By normality, $s_1 = r = s_2$. □

Properties of different reductions

Rmk. Reductions $\to_{h\beta}$ and $\to_{\ell\beta}$ are deterministic (they can fire at most one redex). So:

$$\text{If } t \to_r s_1 \text{ and } t \to_r s_2 \text{ then } s_1 = s_2, \quad \text{for } r \in \{h\beta, l\beta\}.$$

Reduction $\to_\beta$ is not deterministic, it chooses among several β-redexes to fire in a term.

$$\overbrace{(\lambda x.xx)(\underbrace{(\lambda z.z)y}_{\text{inner } \beta\text{-redex}})}^{\text{outermost } \beta\text{-redex}} \to_\beta ((\lambda z.z)y)((\lambda z.z)y)$$

$$(\lambda x.xx)((\lambda z.z)y) \to_\beta (\lambda x.xx)z$$

Notation. $t \to^* s$ means that $t = t_0 \overbrace{\to t_1 \to \cdots \to}^{\text{for some } n \in \mathbb{N}} t_n = s$ (in particular, $t = s$ for $n = 0$).

Theorem (Confluence)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$, then there is a term r such that $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$.

Def. Let $r \in \{\beta, \ell\beta, h\beta\}$. A term t is r-normal if there is no s such that $t \to_r s$.

Corollary (Uniqueness of normal form)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$ where s_1 and s_2 are β-normal, then $s_1 = s_2$.

Proof. By confluence, $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$ for some r. By normality, $s_1 = r = s_2$. □

Properties of different reductions

Rmk. Reductions $\to_{h\beta}$ and $\to_{\ell\beta}$ are deterministic (they can fire at most one redex). So:

$$\text{If } t \to_r s_1 \text{ and } t \to_r s_2 \text{ then } s_1 = s_2, \ \text{ for } r \in \{h\beta, l\beta\}.$$

Reduction $\to_\beta$ is not deterministic, it chooses among several β-redexes to fire in a term.

$$\overbrace{(\lambda x.xx)(\underbrace{(\lambda z.z)y}_{\text{inner } \beta\text{-redex}})}^{\text{outermost } \beta\text{-redex}} \to_\beta ((\lambda z.z)y)((\lambda z.z)y)$$

$$(\lambda x.xx)((\lambda z.z)y) \to_\beta (\lambda x.xx)z$$

Notation. $t \to^* s$ means that $t = t_0 \overbrace{\to t_1 \to \cdots \to}^{\text{for some } n \in \mathbb{N}} t_n = s$ (in particular, $t = s$ for $n = 0$).

Theorem (Confluence)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$, then there is a term r such that $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$.

Def. Let $r \in \{\beta, \ell\beta, h\beta\}$. A term t is r-normal if there is no s such that $t \to_r s$.

Corollary (Uniqueness of normal form)

If $t \to_\beta^* s_1$ and $t \to_\beta^* s_2$ where s_1 and s_2 are β-normal, then $s_1 = s_2$.

Proof. By confluence, $s_1 \to_\beta^* r$ and $s_2 \to_\beta^* r$ for some r. By normality, $s_1 = r = s_2$. □

Normalization, strong normalization and divergence

Def. Let t be a term and $r \in \{\beta, \ell\beta, h\beta\}$.

1. t is r-normalizing if there is a r-normal term s such that $t \to_r^* s$.
2. t is strongly r-normalizing if there is no $(t_i)_{i \in \mathbb{N}}$ such that $t = t_0$ and $t_i \to_r t_{i+1}$.

Ex. Every β-normal form is strongly β-normalizing. Let $\delta = \lambda x.xx$.

- $\delta\delta$ is not β-normalizing: if $\delta\delta \to_\beta t$ then $t = \delta\delta$.
- $(\lambda x.y)(\delta\delta)$ is β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta y$ which is β-normal) but not strongly β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta (\lambda x.y)(\delta\delta) \to_\beta \dots$).

Rmk. Strong normalization implies normalization, but the converse fails, see above.

Rmk. Strong normalization and normalization coincide for $\to_{h\beta}$ and $\to_{l\beta}$, not for $\to_\beta$.

Rmk. In the simply typed λ-calculus, every term is β-normalizing (actually, strongly).

Normalization, strong normalization and divergence

Def. Let t be a term and $r \in \{\beta, \ell\beta, h\beta\}$.

1. t is r-normalizing if there is a r-normal term s such that $t \to_r^* s$.
2. t is strongly r-normalizing if there is no $(t_i)_{i \in \mathbb{N}}$ such that $t = t_0$ and $t_i \to_r t_{i+1}$.

Ex. Every β-normal form is strongly β-normalizing. Let $\delta = \lambda x.xx$.

- $\delta\delta$ is not β-normalizing: if $\delta\delta \to_\beta t$ then $t = \delta\delta$.
- $(\lambda x.y)(\delta\delta)$ is β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta y$ which is β-normal) but not strongly β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta (\lambda x.y)(\delta\delta) \to_\beta \dots$).

Rmk. Strong normalization implies normalization, but the converse fails, see above.

Rmk. Strong normalization and normalization coincide for $\to_{h\beta}$ and $\to_{l\beta}$, not for $\to_\beta$.

Rmk. In the simply typed λ-calculus, every term is β-normalizing (actually, strongly).

Normalization, strong normalization and divergence

Def. Let t be a term and $r \in \{\beta, \ell\beta, h\beta\}$.

1. t is r-normalizing if there is a r-normal term s such that $t \to_r^* s$.
2. t is strongly r-normalizing if there is no $(t_i)_{i \in \mathbb{N}}$ such that $t = t_0$ and $t_i \to_r t_{i+1}$.

Ex. Every β-normal form is strongly β-normalizing. Let $\delta = \lambda x.xx$.

- $\delta\delta$ is not β-normalizing: if $\delta\delta \to_\beta t$ then $t = \delta\delta$.
- $(\lambda x.y)(\delta\delta)$ is β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta y$ which is β-normal) but not strongly β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta (\lambda x.y)(\delta\delta) \to_\beta \dots$).

Rmk. Strong normalization implies normalization, but the converse fails, see above.

Rmk. Strong normalization and normalization coincide for $\to_{h\beta}$ and $\to_{l\beta}$, not for $\to_\beta$.

Rmk. In the simply typed λ-calculus, every term is β-normalizing (actually, strongly).

Normalization, strong normalization and divergence

Def. Let t be a term and $r \in \{\beta, \ell\beta, h\beta\}$.

1. t is r-normalizing if there is a r-normal term s such that $t \to_r^* s$.
2. t is strongly r-normalizing if there is no $(t_i)_{i \in \mathbb{N}}$ such that $t = t_0$ and $t_i \to_r t_{i+1}$.

Ex. Every β-normal form is strongly β-normalizing. Let $\delta = \lambda x.xx$.

- $\delta\delta$ is not β-normalizing: if $\delta\delta \to_\beta t$ then $t = \delta\delta$.
- $(\lambda x.y)(\delta\delta)$ is β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta y$ which is β-normal) but not strongly β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta (\lambda x.y)(\delta\delta) \to_\beta \dots$).

Rmk. Strong normalization implies normalization, but the converse fails, see above.

Rmk. Strong normalization and normalization coincide for $\to_{h\beta}$ and $\to_{l\beta}$, not for $\to_\beta$.

Rmk. In the simply typed λ-calculus, every term is β-normalizing (actually, strongly).

Normalization, strong normalization and divergence

Def. Let t be a term and $r \in \{\beta, \ell\beta, h\beta\}$.

1. t is r-normalizing if there is a r-normal term s such that $t \to_r^* s$.
2. t is strongly r-normalizing if there is no $(t_i)_{i\in\mathbb{N}}$ such that $t = t_0$ and $t_i \to_r t_{i+1}$.

Ex. Every β-normal form is strongly β-normalizing. Let $\delta = \lambda x.xx$.

- $\delta\delta$ is not β-normalizing: if $\delta\delta \to_\beta t$ then $t = \delta\delta$.
- $(\lambda x.y)(\delta\delta)$ is β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta y$ which is β-normal) but not strongly β-normalizing (indeed $(\lambda x.y)(\delta\delta) \to_\beta (\lambda x.y)(\delta\delta) \to_\beta \dots$).

Rmk. Strong normalization implies normalization, but the converse fails, see above.

Rmk. Strong normalization and normalization coincide for $\to_{h\beta}$ and $\to_{l\beta}$, not for $\to_\beta$.

Rmk. In the simply typed λ-calculus, every term is β-normalizing (actually, strongly).

Fixed point combinator

Def. A fixed point of a term t is a term s such that $s \to_\beta^* ts$.
A fixed point combinator is a term Y such that Yt is a fixed point of t, for every term t.

Proposition (Fixed point combinator)

Let $A = \lambda a.\lambda f.f(aaf)$ and $\Theta = AA$. Then, Θ is a fixed point combinator.

Proof. $\Theta = (\lambda a.\lambda f.f(aaf))A \to_{h\beta} \lambda f.f(AAf) = \lambda f.f(\Theta f)$. Therefore, for every term t,

$$\Theta t \to_{h\beta} (\lambda f.f(\Theta f))t \to_{h\beta} t(\Theta t). \qquad \square$$

Rmk. Θ is $h\beta$-normalizing but not β-normalizing.

Rmk. Θ is not a term of the simply typed λ-calculus, because of the subterm aa.

Rmk. Fixed point combinators such has Θ are crucial to represent recursive functions.

Fixed point combinator

Def. A fixed point of a term t is a term s such that $s \to_\beta^* ts$.
A fixed point combinator is a term Y such that Yt is a fixed point of t, for every term t.

Proposition (Fixed point combinator)

Let $A = \lambda a.\lambda f.f(aaf)$ and $\Theta = AA$. Then, Θ is a fixed point combinator.

Proof. $\Theta = (\lambda a.\lambda f.f(aaf))A \to_{h\beta} \lambda f.f(AAf) = \lambda f.f(\Theta f)$. Therefore, for every term t,

$$\Theta t \to_{h\beta} (\lambda f.f(\Theta f))t \to_{h\beta} t(\Theta t).$$ □

Rmk. Θ is $h\beta$-normalizing but not β-normalizing.

Rmk. Θ is not a term of the simply typed λ-calculus, because of the subterm aa.

Rmk. Fixed point combinators such has Θ are crucial to represent recursive functions.

Fixed point combinator

Def. A fixed point of a term t is a term s such that $s \to_\beta^* ts$.
A fixed point combinator is a term Y such that Yt is a fixed point of t, for every term t.

Proposition (Fixed point combinator)

Let $A = \lambda a.\lambda f.f(aaf)$ and $\Theta = AA$. Then, Θ is a fixed point combinator.

Proof. $\Theta = (\lambda a.\lambda f.f(aaf))A \to_{h\beta} \lambda f.f(AAf) = \lambda f.f(\Theta f)$. Therefore, for every term t,

$$\Theta t \to_{h\beta} (\lambda f.f(\Theta f))t \to_{h\beta} t(\Theta t). \qquad \square$$

Rmk. Θ is $h\beta$-normalizing but not β-normalizing.

Rmk. Θ is not a term of the simply typed λ-calculus, because of the subterm aa.

Rmk. Fixed point combinators such has Θ are crucial to represent recursive functions.

Simply typed versus untyped

The simply typed λ-calculus (in Curry-style) is a restriction of the untyped λ-calculus
$\rightsquigarrow$ the latter just take terms and β-reduction from the former without checking typability.

But the untyped λ-calculus can also be seen as a "special case" of the simply type one.
Consider that the simple types are generated by only one ground type X.

Def. Let $\equiv$ be the least congruence on simple types generated by $X \equiv X \Rightarrow X$, that is:

$$\frac{}{X \equiv X} \qquad \frac{A \equiv B}{B \equiv A} \qquad \frac{A \equiv B \quad B \equiv C}{A \equiv C} \qquad \frac{}{X \equiv X \Rightarrow X} \qquad \frac{A \equiv A' \quad B \equiv B'}{A \Rightarrow B \equiv A' \Rightarrow B'}$$

Rmk. $A \equiv X$ for every simple type A (proof by induction on A) $\rightsquigarrow$ All types are the same!

Proposition (Untyped = simply typed + recursive type identity $\equiv$)

Every untyped term is typable in Curry's simply typed λ-calculus extended with the rule:

$$\frac{\Gamma \vdash t : A \quad A \equiv B}{\Gamma \vdash t : B}\equiv$$

Proof. By straightforward induction on t (exercise!). □

Simply typed versus untyped

The simply typed λ-calculus (in Curry-style) is a restriction of the untyped λ-calculus
$\rightsquigarrow$ the latter just take terms and β-reduction from the former without checking typability.

But the untyped λ-calculus can also be seen as a "special case" of the simply type one. Consider that the simple types are generated by only one ground type X.

Def. Let $\equiv$ be the least congruence on simple types generated by $X \equiv X \Rightarrow X$, that is:

$$\frac{}{X \equiv X} \qquad \frac{A \equiv B}{B \equiv A} \qquad \frac{A \equiv B \quad B \equiv C}{A \equiv C} \qquad \frac{}{X \equiv X \Rightarrow X} \qquad \frac{A \equiv A' \quad B \equiv B'}{A \Rightarrow B \equiv A' \Rightarrow B'}$$

Rmk. $A \equiv X$ for every simple type A (proof by induction on A) $\rightsquigarrow$ All types are the same!

Proposition (Untyped = simply typed + recursive type identity $\equiv$)

Every untyped term is typable in Curry's simply typed λ-calculus extended with the rule:

$$\frac{\Gamma \vdash t : A \quad A \equiv B}{\Gamma \vdash t : B}\equiv$$

Proof. By straightforward induction on t (exercise!). □

Simply typed versus untyped

The simply typed λ-calculus (in Curry-style) is a restriction of the untyped λ-calculus
$\rightsquigarrow$ the latter just take terms and β-reduction from the former without checking typability.

But the untyped λ-calculus can also be seen as a "special case" of the simply type one.
Consider that the simple types are generated by only one ground type X.

Def. Let $\equiv$ be the least congruence on simple types generated by $X \equiv X \Rightarrow X$, that is:

$$\frac{}{X \equiv X} \qquad \frac{A \equiv B}{B \equiv A} \qquad \frac{A \equiv B \quad B \equiv C}{A \equiv C} \qquad \frac{}{X \equiv X \Rightarrow X} \qquad \frac{A \equiv A' \quad B \equiv B'}{A \Rightarrow B \equiv A' \Rightarrow B'}$$

Rmk. $A \equiv X$ for every simple type A (proof by induction on A) $\rightsquigarrow$ All types are the same!

Proposition (Untyped = simply typed + recursive type identity $\equiv$)

Every untyped term is typable in Curry's simply typed λ-calculus extended with the rule:

$$\frac{\Gamma \vdash t : A \quad A \equiv B}{\Gamma \vdash t : B}\equiv$$

Proof. By straightforward induction on t (exercise!). □

Outline

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to^*_{h\beta} s$ and $\underline{\bot}\, s\, t \to^*_{h\beta} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to^*_{\beta} \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to^*_{\beta} \underline{\top}$.

$$\underline{not} =$$

2. To encode the AND: $\underline{and} s\, t \to^*_{\beta} \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and} s\, t \to^*_{\beta} \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} =$$

3. To encode the OR: $\underline{or} s\, t \to^*_{\beta} \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or} s\, t \to^*_{\beta} \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} =$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to^*_{\beta} s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to^*_{\beta} t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to^*_{h\beta} s$ and $\underline{\bot}\, s\, t \to^*_{h\beta} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to^*_\beta \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to^*_\beta \underline{\top}$.

$$\underline{not} =$$

2. To encode the AND: $\underline{and}\, s\, t \to^*_\beta \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and}\, s\, t \to^*_\beta \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} =$$

3. To encode the OR: $\underline{or}\, s\, t \to^*_\beta \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or}\, s\, t \to^*_\beta \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} =$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to^*_\beta s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to^*_\beta t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to^*_{h\beta} s$ and $\underline{\bot}\, s\, t \to^*_{h\beta} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to^*_\beta \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to^*_\beta \underline{\top}$.

$$\underline{not} =$$

2. To encode the AND: $\underline{and}\, s\, t \to^*_\beta \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and}\, s\, t \to^*_\beta \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} =$$

3. To encode the OR: $\underline{or}\, s\, t \to^*_\beta \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or}\, s\, t \to^*_\beta \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} =$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to^*_\beta s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to^*_\beta t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to^*_{h\beta} s$ and $\underline{\bot}\, s\, t \to^*_{h\beta} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to^*_\beta \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to^*_\beta \underline{\top}$.

$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$

2. To encode the AND: $\underline{and}\, s\, t \to^*_\beta \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and}\, s\, t \to^*_\beta \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} =$$

3. To encode the OR: $\underline{or}\, s\, t \to^*_\beta \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or}\, s\, t \to^*_\beta \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} =$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to^*_\beta s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to^*_\beta t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represent true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to_{h\beta}^{*} s$ and $\underline{\bot}\, s\, t \to_{h\beta}^{*} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to_{\beta}^{*} \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to_{\beta}^{*} \underline{\top}$.

$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$

2. To encode the AND: $\underline{and}\, s\, t \to_{\beta}^{*} \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} =$$

3. To encode the OR: $\underline{or}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} =$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to_{\beta}^{*} s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to_{\beta}^{*} t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to_{h\beta}^{*} s$ and $\underline{\bot}\, s\, t \to_{h\beta}^{*} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to_{\beta}^{*} \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to_{\beta}^{*} \underline{\top}$.

$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$

2. To encode the AND: $\underline{and} s\, t \to_{\beta}^{*} \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and} s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} = \lambda p.\lambda q.pqp$$

3. To encode the OR: $\underline{or} s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or} s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} =$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to_{\beta}^{*} s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to_{\beta}^{*} t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to_{h\beta}^{*} s$ and $\underline{\bot}\, s\, t \to_{h\beta}^{*} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to_{\beta}^{*} \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to_{\beta}^{*} \underline{\top}$.

$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$

2. To encode the AND: $\underline{and}\, s\, t \to_{\beta}^{*} \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} = \lambda p.\lambda q.pqp$$

3. To encode the OR: $\underline{or}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} =$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to_{\beta}^{*} s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to_{\beta}^{*} t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to^*_{h\beta} s$ and $\underline{\bot}\, s\, t \to^*_{h\beta} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to^*_{\beta} \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to^*_{\beta} \underline{\top}$.

$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$

2. To encode the AND: $\underline{and} s\, t \to^*_{\beta} \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and} s\, t \to^*_{\beta} \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} = \lambda p.\lambda q.pqp$$

3. To encode the OR: $\underline{or} s\, t \to^*_{\beta} \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or} s\, t \to^*_{\beta} \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} = \lambda p.\lambda q.ppq$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to^*_{\beta} s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to^*_{\beta} t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to_{h\beta}^{*} s$ and $\underline{\bot}\, s\, t \to_{h\beta}^{*} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to_{\beta}^{*} \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to_{\beta}^{*} \underline{\top}$.

$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$

2. To encode the AND: $\underline{and}\, s\, t \to_{\beta}^{*} \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} = \lambda p.\lambda q.pqp$$

3. To encode the OR: $\underline{or}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or}\, s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} = \lambda p.\lambda q.ppq$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to_{\beta}^{*} s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to_{\beta}^{*} t$ if $r = \underline{\bot}$.

$$\underline{if} =$$

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every term s, t, we have $\underline{\top}\, s\, t \to_{h\beta}^{*} s$ and $\underline{\bot}\, s\, t \to_{h\beta}^{*} t$.

1. We look for a term to encode the NOT: $\underline{not}\,\underline{\top} \to_{\beta}^{*} \underline{\bot}$ and $\underline{not}\,\underline{\bot} \to_{\beta}^{*} \underline{\top}$.

$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$

2. To encode the AND: $\underline{and} s\, t \to_{\beta}^{*} \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and} s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.

$$\underline{and} = \lambda p.\lambda q.pqp$$

3. To encode the OR: $\underline{or} s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or} s\, t \to_{\beta}^{*} \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.

$$\underline{or} = \lambda p.\lambda q.ppq$$

4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to_{\beta}^{*} s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to_{\beta}^{*} t$ if $r = \underline{\bot}$.

$$\underline{if} = \lambda p.\lambda a.\lambda b.pab$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f \ldots (f}_{n \text{ times } f} x) \ldots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x\text{)}$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to_{h\beta}^* s^n t = \overbrace{s(s \ldots (s}^{n \text{ times } s} t) \ldots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\, \underline{n} \to_\beta^* \underline{n+1}$.

   $$\underline{succ} =$$

2. To encode the addition: $\underline{add}\, \underline{m}\, \underline{n} \to_\beta^* \underline{m+m}$.

   $$\underline{add} =$$

3. To encode the multiplication: $\underline{mult}\, \underline{m}\, \underline{n} \to_\beta^* \underline{m \times n}$.

   $$\underline{mult} =$$

4. To encode the exponentiation: $\underline{pow}\, \underline{m}\, \underline{n} \to_\beta^* \underline{m^n}$.

   $$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f \dots (f}_{n \text{ times } f}\, x) \dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x\text{)}$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to^*_{h\beta} s^n t = \overbrace{s(s \dots (s}^{n \text{ times } s}\, t) \dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to^*_\beta \underline{n+1}$.

$$\underline{succ} =$$

2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to^*_\beta \underline{m+m}$.

$$\underline{add} =$$

3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to^*_\beta \underline{m \times n}$.

$$\underline{mult} =$$

4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to^*_\beta \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f\dots(f}_{n \text{ times } f}\, x)\dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x)$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to_{h\beta}^{*} s^n t = \overbrace{s(s\dots(s}^{n \text{ times } s}\, t)\dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to_{\beta}^{*} \underline{n+1}$.

$$\underline{succ} =$$

2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m+m}$.

$$\underline{add} =$$

3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m \times n}$.

$$\underline{mult} =$$

4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f\dots(f}_{n \text{ times } f}\, x)\dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x)$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to_{h\beta}^{*} s^n t = \overbrace{s(s\dots(s}^{n \text{ times } s}\, t)\dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to_{\beta}^{*} \underline{n+1}$.

$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m+m}$.

$$\underline{add} =$$

3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m \times n}$.

$$\underline{mult} =$$

4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f\dots(f}_{n \text{ times } f}\,x)\dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x\text{)}$$

Rmk. For every term s, t, we have $\underline{n}\,s\,t \to_{h\beta}^* s^n t = \overbrace{s(s\dots(s}^{n \text{ times } s}\,t)\dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to_\beta^* \underline{n+1}$.

$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m+m}$.

$$\underline{add} =$$

3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m \times n}$.

$$\underline{mult} =$$

4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f \dots (f}_{n \text{ times } f} x)\dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x\text{)}$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to_{h\beta}^{*} s^n t = \overbrace{s(s \dots (s}^{n \text{ times } s}\, t)\dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to_\beta^* \underline{n+1}$.

$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m+m}$.

$$\underline{add} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m \times n}$.

$$\underline{mult} =$$

4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f\dots(f}_{n \text{ times } f}\, x)\dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x)$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to_{h\beta}^{*} s^n t = \overbrace{s(s\dots(s}^{n \text{ times } s}\, t)\dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to_{\beta}^{*} \underline{n+1}$.

$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m+m}$.

$$\underline{add} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m \times n}$.

$$\underline{mult} =$$

4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to_{\beta}^{*} \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f\dots(f}_{n \text{ times } f}x)\dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x)$$

Rmk. For every term s, t, we have $\underline{n}\,s\,t \to_{h\beta}^* s^n t = \overbrace{s(s\dots(s}^{n \text{ times } s}\,t)\dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to_\beta^* \underline{n+1}$.

$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m+m}$.

$$\underline{add} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m \times n}$.

$$\underline{mult} = \lambda m.\lambda n.\lambda f.m(nf)$$

4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f \dots (f}_{n \text{ times } f}\, x)\dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x)$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to_{h\beta}^* s^n t = \overbrace{s(s \dots (s}^{n \text{ times } s}\, t)\dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\, \underline{n} \to_\beta^* \underline{n+1}$.

$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

2. To encode the addition: $\underline{add}\, \underline{m}\, \underline{n} \to_\beta^* \underline{m+m}$.

$$\underline{add} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

3. To encode the multiplication: $\underline{mult}\, \underline{m}\, \underline{n} \to_\beta^* \underline{m \times n}$.

$$\underline{mult} = \lambda m.\lambda n.\lambda f.m(nf)$$

4. To encode the exponentiation: $\underline{pow}\, \underline{m}\, \underline{n} \to_\beta^* \underline{m^n}$.

$$\underline{pow} =$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f \dots (f}_{n \text{ times } f}\, x) \dots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x\text{)}$$

Rmk. For every term s, t, we have $\underline{n}\, s\, t \to^*_{h\beta} s^n t = \overbrace{s(s \dots (s}^{n \text{ times } s}\, t) \dots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\, \underline{n} \to^*_\beta \underline{n+1}$.

$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

2. To encode the addition: $\underline{add}\, \underline{m}\, \underline{n} \to^*_\beta \underline{m+m}$.

$$\underline{add} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

3. To encode the multiplication: $\underline{mult}\, \underline{m}\, \underline{n} \to^*_\beta \underline{m \times n}$.

$$\underline{mult} = \lambda m.\lambda n.\lambda f.m(nf)$$

4. To encode the exponentiation: $\underline{pow}\, \underline{m}\, \underline{n} \to^*_\beta \underline{m^n}$.

$$\underline{pow} = \lambda m.\lambda n.nm$$

More about encoding arithmetic: recursion

We can encode the functions: $\mathit{iszero}\colon \mathbb{N} \to \{\bot, \top\}$ testing if a natural number is 0 or not, and the predecessor $\mathit{pred}\colon \mathbb{N} \to \mathbb{N}$ such that $\mathit{pred}(0) = 0$ and $\mathit{pred}(n+1) = n$.

$$\underline{\mathit{iszero}} = \lambda n.n(\lambda x.\underline{\bot})\underline{\top} \qquad \underline{\mathit{iszero}}\,\underline{n} \to_\beta^* \begin{cases} \underline{\top} & \text{if } n = 0 \\ \underline{\bot} & \text{otherwise.} \end{cases} \qquad \underline{\mathit{pred}} = \dots$$

Question. How can the λ-calculus represent the factorial (typical recursive function)?

$$\mathit{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \mathit{fact}(n-1) & \text{otherwise.} \end{cases}$$

Let us rewrite the definition in a λ-calculus-like style, using IF-THEN-ELSE and $\underline{\mathit{mult}}$:

$$F := \lambda f.\lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(f\,(\underline{\mathit{pred}}\,n)))$$

$$\underline{\mathit{fact}} := YF \quad \to_\beta^* F(YF) = F\,\underline{\mathit{fact}} \to_\beta \lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(\underline{\mathit{fact}}\,(\underline{\mathit{pred}}\,n)))$$

More about encoding arithmetic: recursion

We can encode the functions: $\mathit{iszero}\colon \mathbb{N} \to \{\bot, \top\}$ testing if a natural number is 0 or not, and the predecessor $\mathit{pred}\colon \mathbb{N} \to \mathbb{N}$ such that $\mathit{pred}(0) = 0$ and $\mathit{pred}(n+1) = n$.

$$\underline{\mathit{iszero}} = \lambda n.n(\lambda x.\underline{\bot})\underline{\top} \qquad \underline{\mathit{iszero}}\,\underline{n} \to_\beta^* \begin{cases} \underline{\top} & \text{if } n = 0 \\ \underline{\bot} & \text{otherwise.} \end{cases} \qquad \underline{\mathit{pred}} = \dots$$

Question. How can the λ-calculus represent the factorial (typical recursive function)?

$$\mathit{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \mathit{fact}(n-1) & \text{otherwise.} \end{cases}$$

Let us rewrite the definition in a λ-calculus-like style, using IF-THEN-ELSE and $\underline{\mathit{mult}}$:

$\underline{\mathit{fact}}$ should satisfies the equation: $\underline{\mathit{fact}}\,\underline{n} = \underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,\underline{n})\,\underline{1}\,(\underline{\mathit{mult}}\,\underline{n}\,(\underline{\mathit{fact}}\,(\underline{\mathit{pred}}\,\underline{n})))$

$$F := \lambda f.\lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(f\,(\underline{\mathit{pred}}\,n)))$$

$$\underline{\mathit{fact}} := YF \;\to_\beta^* F(YF) = F\,\underline{\mathit{fact}} \to_\beta \lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(\underline{\mathit{fact}}\,(\underline{\mathit{pred}}\,n)))$$

More about encoding arithmetic: recursion

We can encode the functions: *iszero*: $\mathbb{N} \to \{\bot, \top\}$ testing if a natural number is 0 or not, and the predecessor *pred*: $\mathbb{N} \to \mathbb{N}$ such that $pred(0) = 0$ and $pred(n+1) = n$.

$$\underline{iszero} = \lambda n.n(\lambda x.\underline{\bot})\underline{\top} \qquad \underline{iszero}\ \underline{n} \to_\beta^* \begin{cases} \underline{\top} & \text{if } n = 0 \\ \underline{\bot} & \text{otherwise.} \end{cases} \qquad \underline{pred} = \dots$$

Question. How can the λ-calculus represent the factorial (typical recursive function)?

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{otherwise.} \end{cases}$$

Let us rewrite the definition in a λ-calculus-like style, using IF-THEN-ELSE and $\underline{mult}$:

$\underline{fact}$ should satisfies the equation: $\underline{fact} = \lambda n.\underline{if}\,(\underline{iszero}\, n)\,\underline{1}\,(\underline{mult}\, n\,(\underline{fact}\,(\underline{pred}\, n)))$

$F := \lambda f.\lambda n.\underline{if}\,(\underline{iszero}\, n)\,\underline{1}\,(\underline{mult}\, n\,(f\,(\underline{pred}\, n)))$

$\underline{fact} := YF \quad \to_\beta^* F(YF) = F\,\underline{fact} \to_\beta \lambda n.\underline{if}\,(\underline{iszero}\, n)\,\underline{1}\,(\underline{mult}\, n\,(\underline{fact}\,(\underline{pred}\, n)))$

More about encoding arithmetic: recursion

We can encode the functions: $\mathit{iszero}\colon \mathbb{N} \to \{\bot, \top\}$ testing if a natural number is 0 or not, and the predecessor $\mathit{pred}\colon \mathbb{N} \to \mathbb{N}$ such that $\mathit{pred}(0) = 0$ and $\mathit{pred}(n+1) = n$.

$$\underline{\mathit{iszero}} = \lambda n.n(\lambda x.\underline{\bot})\underline{\top} \qquad \underline{\mathit{iszero}}\,\underline{n} \to_\beta^* \begin{cases} \underline{\top} & \text{if } n = 0 \\ \underline{\bot} & \text{otherwise.} \end{cases} \qquad \underline{\mathit{pred}} = \dots$$

Question. How can the λ-calculus represent the factorial (typical recursive function)?

$$\mathit{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \mathit{fact}(n-1) & \text{otherwise.} \end{cases}$$

Let us rewrite the definition in a λ-calculus-like style, using IF-THEN-ELSE and $\underline{\mathit{mult}}$:

$\underline{\mathit{fact}}$ should satisfies the equation: $\underline{\mathit{fact}} = \lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(\underline{\mathit{fact}}\,(\underline{\mathit{pred}}\,n)))$

$$F := \lambda f.\lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(f\,(\underline{\mathit{pred}}\,n)))$$

$$\underline{\mathit{fact}} := YF \ \to_\beta^* F(YF) = F\,\underline{\mathit{fact}} \to_\beta \lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(\underline{\mathit{fact}}\,(\underline{\mathit{pred}}\,n)))$$

More about encoding arithmetic: recursion

We can encode the functions: $\mathit{iszero}\colon \mathbb{N} \to \{\bot, \top\}$ testing if a natural number is 0 or not, and the predecessor $\mathit{pred}\colon \mathbb{N} \to \mathbb{N}$ such that $\mathit{pred}(0) = 0$ and $\mathit{pred}(n+1) = n$.

$$\underline{\mathit{iszero}} = \lambda n.n(\lambda x.\underline{\bot})\underline{\top} \qquad \underline{\mathit{iszero}}\,\underline{n} \to_\beta^* \begin{cases} \underline{\top} & \text{if } n = 0 \\ \underline{\bot} & \text{otherwise.} \end{cases} \qquad \underline{\mathit{pred}} = \dots$$

Question. How can the λ-calculus represent the factorial (typical recursive function)?

$$\mathit{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \mathit{fact}(n-1) & \text{otherwise.} \end{cases}$$

Let us rewrite the definition in a λ-calculus-like style, using IF-THEN-ELSE and $\underline{\mathit{mult}}$:

$\underline{\mathit{fact}}$ should satisfies the equation: $\underline{\mathit{fact}} = \lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(\underline{\mathit{fact}}\,(\underline{\mathit{pred}}\,n)))$

$$F := \lambda f.\lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(f\,(\underline{\mathit{pred}}\,n)))$$

$$\underline{\mathit{fact}} := YF \to_\beta^* F(YF) = F\,\underline{\mathit{fact}} \to_\beta \lambda n.\underline{\mathit{if}}\,(\underline{\mathit{iszero}}\,n)\,\underline{1}\,(\underline{\mathit{mult}}\,n\,(\underline{\mathit{fact}}\,(\underline{\mathit{pred}}\,n)))$$

More about encoding arithmetic: recursion

We can encode the functions: $iszero \colon \mathbb{N} \to \{\bot, \top\}$ testing if a natural number is 0 or not, and the predecessor $pred \colon \mathbb{N} \to \mathbb{N}$ such that $pred(0) = 0$ and $pred(n+1) = n$.

$$\underline{iszero} = \lambda n.n(\lambda x.\underline{\bot})\underline{\top} \qquad \underline{iszero}\ \underline{n} \to_\beta^* \begin{cases} \underline{\top} & \text{if } n = 0 \\ \underline{\bot} & \text{otherwise.} \end{cases} \qquad \underline{pred} = \dots$$

Question. How can the λ-calculus represent the factorial (typical recursive function)?

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{otherwise.} \end{cases}$$

Let us rewrite the definition in a λ-calculus-like style, using IF-THEN-ELSE and $\underline{mult}$:

$\underline{fact}$ should satisfies the equation: $\underline{fact} = \lambda n.\underline{if}\,(\underline{iszero}\,n)\,\underline{1}\,(\underline{mult}\,n\,(\underline{fact}\,(\underline{pred}\,n)))$

$$F := \lambda f.\lambda n.\underline{if}\,(\underline{iszero}\,n)\,\underline{1}\,(\underline{mult}\,n\,(f\,(\underline{pred}\,n)))$$

$$\underline{fact} := YF \ \to_\beta^* F(YF) = F\,\underline{fact} \to_\beta \lambda n.\underline{if}\,(\underline{iszero}\,n)\,\underline{1}\,(\underline{mult}\,n\,(\underline{fact}\,(\underline{pred}\,n)))$$

The untyped λ-calculus is Turing-complete!

Def. Let $f \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ be partial. A term Φ represents f when, for all $k_1, \dots, k_n \in \mathbb{N}$:

1. if $f(k_1, \dots, k_n)$ is undefined, then $\Phi\, \underline{k_1} \dots \underline{k_n}$ is not $h\beta$-normalizing;
2. if $f(k_1, \dots, k_n) = k \in \mathbb{N}$, then $\Phi\, \underline{k_1} \dots \underline{k_n} \to_\beta^* \underline{k}$.

Theorem (Representability)

Every partial recursive function $f \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ is representable by a term in the λ-calculus.

Rmk. According to Church's thesis, the λ-calculus can represent everything is computable.

Rmk. If Φ represents a partial function $f \colon \mathbb{N}^k \rightharpoonup \mathbb{N}$, then Φ could have whatever behavior when applied to arguments $t_1, \dots, t_k$ that are not Church numerals.

Rmk. In Point 1 of the definition, $h\beta$-normalizing can be replaced by β-normalizing.

The untyped λ-calculus is Turing-complete!

Def. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be partial. A term Φ represents f when, for all $k_1, \dots, k_n \in \mathbb{N}$:

1. if $f(k_1, \dots, k_n)$ is undefined, then $\Phi\, \underline{k_1} \dots \underline{k_n}$ is not $h\beta$-normalizing;
2. if $f(k_1, \dots, k_n) = k \in \mathbb{N}$, then $\Phi\, \underline{k_1} \dots \underline{k_n} \to_\beta^* \underline{k}$.

Theorem (Representability)

Every partial recursive function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is representable by a term in the λ-calculus.

Rmk. According to Church's thesis, the λ-calculus can represent everything is computable.

Rmk. If Φ represents a partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$, then Φ could have whatever behavior when applied to arguments $t_1, \dots, t_k$ that are not Church numerals.

Rmk. In Point 1 of the definition, $h\beta$-normalizing can be replaced by β-normalizing.

The untyped λ-calculus is Turing-complete!

Def. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be partial. A term Φ represents f when, for all $k_1, \ldots, k_n \in \mathbb{N}$:

1. if $f(k_1, \ldots, k_n)$ is undefined, then $\Phi\, \underline{k_1} \ldots \underline{k_n}$ is not $h\beta$-normalizing;
2. if $f(k_1, \ldots, k_n) = k \in \mathbb{N}$, then $\Phi\, \underline{k_1} \ldots \underline{k_n} \to_\beta^* \underline{k}$.

Theorem (Representability)

Every partial recursive function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is representable by a term in the λ-calculus.

Rmk. According to Church's thesis, the λ-calculus can represent everything is computable.

Rmk. If Φ represents a partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$, then Φ could have whatever behavior when applied to arguments $t_1, \ldots, t_k$ that are not Church numerals.

Rmk. In Point 1 of the definition, $h\beta$-normalizing can be replaced by β-normalizing.

The untyped λ-calculus is Turing-complete!

Def. Let $f \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ be partial. A term Φ represents f when, for all $k_1, \dots, k_n \in \mathbb{N}$:

1. if $f(k_1, \dots, k_n)$ is undefined, then $\Phi\, \underline{k_1} \dots \underline{k_n}$ is not $h\beta$-normalizing;
2. if $f(k_1, \dots, k_n) = k \in \mathbb{N}$, then $\Phi\, \underline{k_1} \dots \underline{k_n} \to_\beta^* \underline{k}$.

Theorem (Representability)

Every partial recursive function $f \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ is representable by a term in the λ-calculus.

Rmk. According to Church's thesis, the λ-calculus can represent everything is computable.

Rmk. If Φ represents a partial function $f \colon \mathbb{N}^k \rightharpoonup \mathbb{N}$, then Φ could have whatever behavior when applied to arguments $t_1, \dots, t_k$ that are not Church numerals.

Rmk. In Point 1 of the definition, $h\beta$-normalizing can be replaced by β-normalizing.

The λ-calculus as a programming language

λ-calculus = kernel of all functional programming languages (Haskell, OCaml, Lisp, ...).

- An abstraction $\lambda x.t$ is an anonymous function `fun x -> t` in OCaml.
- A application tu (resp. variable x) is an application `tu` (resp. variable `x`) in OCaml.

OCaml	λ-calculus
`(fun x -> x + 3)7`	$(\lambda x.\underline{add}\, x\, \underline{3})\, \underline{7}$
$\Downarrow$	$\downarrow_\beta$
`7 + 3`	$\underline{add}\, \underline{7}\, \underline{3}$
$\Downarrow$	$*\downarrow_\beta$
`10`	$\underline{10}$

- OCaml imposes eager evaluation: evaluate arguments before applying the function.
- In the λ-calculus the order of evaluation is irrelevant (because of confluence).

The λ-calculus as a programming language

λ-calculus = kernel of all functional programming languages (Haskell, OCaml, Lisp, ...).

- An abstraction $\lambda x.t$ is an anonymous function `fun x -> t` in OCaml.
- A application tu (resp. variable x) is an application `tu` (resp. variable `x`) in OCaml.

OCaml	λ-calculus
`(fun x -> x + 3)7`	$(\lambda x.\underline{add}\, x\, \underline{3})\, \underline{7}$
$\Downarrow$	$\downarrow_\beta$
`7 + 3`	$\underline{add}\, \underline{7}\, \underline{3}$
$\Downarrow$	$\downarrow^*_\beta$
`10`	$\underline{10}$

- OCaml imposes eager evaluation: evaluate arguments before applying the function.
- In the λ-calculus the order of evaluation is irrelevant (because of confluence).

The λ-calculus as a programming language

λ-calculus = kernel of all functional programming languages (Haskell, OCaml, Lisp, ...).

- An abstraction $\lambda x.t$ is an anonymous function `fun x -> t` in OCaml.
- A application tu (resp. variable x) is an application `tu` (resp. variable `x`) in OCaml.

OCaml	λ-calculus
`(fun x -> x + 3)7`	$(\lambda x.\underline{add}\, x\, \underline{3})\, \underline{7}$
$\Downarrow$	$\downarrow_\beta$
`7 + 3`	$\underline{add}\, \underline{7}\, \underline{3}$
$\Downarrow$	${}_*\!\downarrow_\beta$
`10`	$\underline{10}$

- OCaml imposes eager evaluation: evaluate arguments before applying the function.
- In the λ-calculus the order of evaluation is irrelevant (because of confluence).

The λ-calculus as a programming language

λ-calculus = kernel of all functional programming languages (Haskell, OCaml, Lisp, ...).

- An abstraction $\lambda x.t$ is an anonymous function `fun x -> t` in OCaml.
- A application tu (resp. variable x) is an application `tu` (resp. variable `x`) in OCaml.

OCaml	λ-calculus
`(fun x -> x + 3)7`	$(\lambda x.\underline{add}\, x\, \underline{3})\, \underline{7}$
$\Downarrow$	$\downarrow_\beta$
`7 + 3`	$\underline{add}\, \underline{7}\, \underline{3}$
$\Downarrow$	${}_*\!\downarrow_\beta$
`10`	$\underline{10}$

- OCaml imposes eager evaluation: evaluate arguments before applying the function.
- In the λ-calculus the order of evaluation is irrelevant (because of confluence).

Outline

What we have learned today?

1. Syntax and operational semantics of the untyped λ-calculus.
2. Different notions of β-reduction (full, leftmost, head).
3. Different notions of normalization (strong or not).
4. How to encode arithmetic and propositional classical logic in the untyped λ-calculus.
5. Representability of every partial recursive function in the untyped λ-calculus

Questions?

What we have learned today?

1. Syntax and operational semantics of the untyped λ-calculus.
2. Different notions of β-reduction (full, leftmost, head).
3. Different notions of normalization (strong or not).
4. How to encode arithmetic and propositional classical logic in the untyped λ-calculus.
5. Representability of every partial recursive function in the untyped λ-calculus

Questions?

What we have learned today?

1. Syntax and operational semantics of the untyped λ-calculus.
2. Different notions of β-reduction (full, leftmost, head).
3. Different notions of normalization (strong or not).
4. How to encode arithmetic and propositional classical logic in the untyped λ-calculus.
5. Representability of every partial recursive function in the untyped λ-calculus

Questions?

What we have learned today?

1. Syntax and operational semantics of the untyped λ-calculus.
2. Different notions of β-reduction (full, leftmost, head).
3. Different notions of normalization (strong or not).
4. How to encode arithmetic and propositional classical logic in the untyped λ-calculus.
5. Representability of every partial recursive function in the untyped λ-calculus

Questions?

What we have learned today?

1. Syntax and operational semantics of the untyped λ-calculus.
2. Different notions of β-reduction (full, leftmost, head).
3. Different notions of normalization (strong or not).
4. How to encode arithmetic and propositional classical logic in the untyped λ-calculus.
5. Representability of every partial recursive function in the untyped λ-calculus

Questions?

What we have learned today?

1. Syntax and operational semantics of the untyped λ-calculus.
2. Different notions of β-reduction (full, leftmost, head).
3. Different notions of normalization (strong or not).
4. How to encode arithmetic and propositional classical logic in the untyped λ-calculus.
5. Representability of every partial recursive function in the untyped λ-calculus

Questions?

Exercises

1. Write the tree representation of following terms (as on p. 7), specifying $m, n \in \mathbb{N}$ and the subtrees corresponding to $h, t_1, \dots, t_m$: x, I, $\lambda x.Ixx$, $\lambda x.I(xx)$, $\lambda x.xxx(xx)$, II.
2. The β-reduction graph of a term t is the directed graph with nodes $\{s \mid t \to_\beta^* s\}$ and with edges the single β-steps. Draw the β-reduction graph of the following terms:
    1. $(\lambda x.Ixx)(\lambda x.Ixx)$ where $I = \lambda z.z$.
    2. $(\lambda x.I(xx))(\lambda x.I(xx))$.
    3. $(II)(III)$.
    4. $\delta\delta$ where $\delta = \lambda x.xx$.
    5. $\delta_3\delta_3$ where $\delta_3 = \lambda x.xxx$.
    6. $\pi\pi\pi$ where $\pi = \lambda x.\lambda y.xyy$.
3. Consider the η-reduction $\to_\eta$ defined below, which can be fired everywhere in a term. Prove that $\to_\eta$ is strongly normalizing.
$$\lambda x.tx \to_\eta t \qquad \text{if } x \notin \mathsf{fv}(t)$$
4. Prove rigorously the remark and proposition on p. 13.
5. Find a term r such that $rt \to_\beta^* t(tr)$ for every t (*Hint:* use fixpoint combinator Θ).
6. Prove that $\underline{succ}\ \underline{n} \to_\beta^* \underline{n+1}$ for all $n \in \mathbb{N}$, and $\underline{add}\ \underline{m}\ \underline{n} \to_\beta^* \underline{m+n}$ for all $m, n \in \mathbb{N}$.
7. Find terms t, t', s, s' such that $t =_\alpha t'$, $s =_\alpha s'$ and $t[s/x] \neq_\alpha t'[s'/x]$ (where $=_\alpha$ is α-equivalence and $t[s/x]$ is naïve substitution, see p. 10 on Day 2 slides).
8. Define a term $\underline{add}$ that represents the addition of natural numbers starting from its inductive definition below (*Hint:* Use the fixpoint combinator Θ, $\underline{\text{pred}}$, $\underline{\text{iszero}}$).
9. Define a term $\underline{mul}$ that represents the multiplication of natural numbers starting from its inductive definition below (*Hint:* Use fixpoint combinator Θ, $\underline{\text{pred}}$, $\underline{\text{iszero}}$).

$$m + n = \begin{cases} m & \text{if } n = 0 \\ m + (n-1) & \text{otherwise;} \end{cases} \qquad m \times n = \begin{cases} 0 & \text{if } n = 0 \\ m + m \times (n-1) & \text{otherwise.} \end{cases}$$

Bibliography

- For more about the untyped λ-calculus:

Jean-Louis Krivine. *Lambda-Calculus. Types and Models*. Ellis Horwood. 1990. [Chapters 1-2] `https://www.irif.fr/~krivine/articles/Lambda.pdf`

Peter Selinger. *Lecture Notes on the Lambda Calculus*. vol. 0804, Department of Mathematics and Statistics, University of Ottawa. 2008 [Chapters 2-3] `http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf`

Henk P. Barendregt. *The Lambda-Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, vol. 103, North Holland, 1984. [Chapters 2-3, 6, 8]

- For an elegant proof of the confluence of β-reduction:

Masako Takahashi. *Parallel Reductions in λ-Calculus*. Information and Computation, vol. 118, issue 1, pages 120-127. 1995. `https://doi.org/10.1006/inco.1995.1057`