The λ-calculus: from simple types to non-idempotent intersection types
Day 2: The untyped λ-calculus

Giulio Guerrieri

Department of Informatics, University of Sussex (Brighton, UK)
LIS, Aix-Marseille Université (Marseille, France)
✉ giulio.guerrieri@lis-lab.fr https://pageperso.lis-lab.fr/~giulio.guerrieri/

34th European Summer School in Logic, Language and Information (ESSLLI 2023)
Ljubljana (Slovenia), 7-11 August 2023

Outline

1. The syntax and the operational semantics of the untyped λ-calculus

2. Programming with the untyped λ-calculus

3. Conclusion, exercises and bibliography

Outline

1. The syntax and the operational semantics of the untyped λ-calculus
2. Programming with the untyped λ-calculus
3. Conclusion, exercises and bibliography

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?
- $(\lambda x.xx)(\lambda x.xx) \to_\beta (xx)\{\lambda x.xx/x\} = (\lambda x.xx)(\lambda x.xx) \to_\beta \dots$ (normalization fails).

The untyped λ-calculus

Terms: $s, t ::= x$ (variable) $\mid$ $\lambda x.t$ (abstraction) $\mid$ st (application).

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

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

Terms: $s, t ::= x$ (variable) $\mid$ $\lambda x.t$ (abstraction) $\mid$ st (application).

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

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

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application).

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

Terms: $s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application).

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

Terms: $s, t ::= x$ (variable) $\mid$ $\lambda x.t$ (abstraction) $\mid$ st (application).

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

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

Terms: $\quad s, t ::= x$ (variable) $\mid \lambda x.t$ (abstraction) $\mid st$ (application).

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

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 structure of a term.

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

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

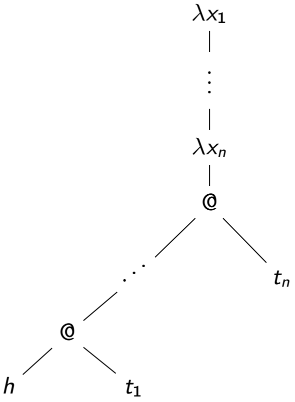
where h is either a variable (head variable) or a β-redex (head β-redex).

The structure of a term.

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

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

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



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

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

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 $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{l\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{l\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ 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 $\to_{h\beta}$ fires the leftmost-outermost β-redex in a term.

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

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 $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{l\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{l\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ 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 $\to_{h\beta}$ fires the leftmost-outermost β-redex in a term.

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

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 $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{l\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{l\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ 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 $\to_{h\beta}$ fires the leftmost-outermost β-redex in a term.

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

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 $t \not\to_{h\beta} (Ix)(Iy)z$;
- $x(Iy)(Iz) \to_{l\beta} xy(Iz)$ but $x(Iy)(Iz) \not\to_{l\beta} x(Iy)z$;
- $t \to_\beta (Ix)(Iy)z$ 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 $\to_{h\beta}$ fires the leftmost-outermost β-redex in a term.

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

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

Properties of different reductions

Rmk. Reductions $\to_{h\beta}$ and $\to_{l\beta}$ 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.

$$\begin{array}{ccl} & & ((\lambda z.z)y)((\lambda z.z)y) \\ & \nearrow_{\beta} & \\ \overbrace{(\lambda x.xx)(\underbrace{(\lambda z.z)y}_{\text{inner } \beta\text{-redex}})}^{\text{outermost } \beta\text{-redex}} & \xrightarrow[\beta]{} & (\lambda x.xx)z \end{array}$$

Notation. $t \to^* s$ means that $t = \overbrace{t_0 \to t_1 \to \cdots \to t_n}^{\text{for some } n \in \mathbb{N}} = s$ (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 $r \in \{\beta, l\beta, h\beta\}$. 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 $\to_{l\beta}$ 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.

$$\begin{tikzcd} & ((\lambda z.z)y)((\lambda z.z)y) \\ \overbrace{(\lambda x.xx)(\underbrace{(\lambda z.z)y}_{\text{inner } \beta\text{-redex}})}^{\text{outermost } \beta\text{-redex}} \arrow[ur, "\beta"'] \arrow[r, "\beta"'] & (\lambda x.xx)z \end{tikzcd}$$

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 $r \in \{\beta, l\beta, h\beta\}$. 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 $\to_{l\beta}$ 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.

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

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

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 $r \in \{\beta, l\beta, h\beta\}$. 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 $\to_{l\beta}$ 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.

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

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 $r \in \{\beta, l\beta, h\beta\}$. 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, $s_1 = r = s_2$. $\square$

Normalization, strong normalization and divergence

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

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 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 $r \in \{\beta, l\beta, h\beta\}$.

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 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 $r \in \{\beta, l\beta, h\beta\}$.

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 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 $r \in \{\beta, l\beta, h\beta\}$.

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 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 $r \in \{\beta, l\beta, h\beta\}$.

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 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. *Theta* 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. *Theta* 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. *Theta* 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.

Outline

1. The syntax and the operational semantics of the untyped λ-calculus

2. Programming with the untyped λ-calculus

3. Conclusion, exercises and bibliography

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 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 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 $r = \underline{\bot}$.

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

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).

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

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).

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

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).

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

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).

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

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

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

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

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).

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

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: $\underline{succ}\,\underline{n} \to_{\beta}^{*} \underline{n+1}$.

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

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).

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

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: *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 $\underline{mult}$:

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

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

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: $\mathit{iszero} \colon \mathbb{N} \to \{\bot, \top\}$ 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* : $\mathbb{N} \to \{\bot, \top\}$ 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: $\underline{fact} = \lambda n.\underline{if}\,(\underline{iszero}\,n)\,\underline{1}\,(\underline{mult}\,n\,(\underline{fact}\,(\underline{pred}\,n)))$

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

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

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

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

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

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 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.

Outline

1. The syntax and the operational semantics of the untyped λ-calculus
2. Programming with the untyped λ-calculus
3. Conclusion, exercises and bibliography

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`