编程语言的设计原理

Design Principles of Programming Languages

Haiyan Zhao，Di Wang

赵海燕，王迪

Peking University, Spring Term 2024

Recap: untyped lambda-calculus

Syntax

t	::=		*terms:*
		x	*variable*
		$\lambda x.t$	*abstraction*
		$t\ t$	*application*
v	::=		*values:*
		$\lambda x.t$	*abstraction value*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

- Terminology:
  - terms in the pure λ-calculus are often called λ-*terms*
  - terms of the form $\lambda x.\ t$ are called λ-*abstractions* or just abstractions

Syntactic conventions

- The λ-calculus provides *only one-argument functions*, all multi-argument functions must be written in curried style.
- The following *conventions* make the linear forms of terms easier to read and write:
  - Application *associates to the left*

    e.g., $t\ u\ v$ means $(t\ u)\ v$, not $t\ (u\ v)$
  - Bodies of λ- abstractions *extend as far to the right as possible*

    e.g., $\lambda x.\ \lambda y. x\ y$ means $\lambda x.\ (\lambda y.\ x\ y)$, not $\lambda x.\ (\lambda y.\ x)\ y$

Scope

- *An occurrence* of the variable x is said to be ***bound*** when it occurs in the body t of an abstraction λx.t, i.e.,
  - the λ-abstraction term λx.t binds the variable x , and the scope of this binding is the body t.
  - λx is a *binder* whose *scope* is t.
  - a binder can be *renamed* as necessary
    - so-called: *alpha-renaming*
    - e.g., $\lambda x.\mathrm{x} = \lambda \mathrm{y}.\ \mathrm{y}$

Operational Semantics

- *Beta-reduction*: the only computation (substitution)

$$(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12},$$

  - the term obtained by *replacing all free occurrences* of x in t_{12} by t_2
  - a term of the form *(λx.t) v* — a *λ-abstraction* applied to a *value* — is called a *redex* (short for "*reducible expression*")
  - the operation of rewriting a *redex* according to the above rule is called *beta-reduction*
- Examples:

$$(\lambda x.\ x)\ y \rightarrow\ y$$

$$(\lambda x.\ x\ (\lambda x.\ x))\ (u\ r) \rightarrow u\ r\ (\lambda x.\ x)$$

Evaluation Strategies

- Full beta-reduction
  - *any redex* may be reduced *at any time*.
  - **confluent** under full beta-reduction
- normal order strategy
  - The *leftmost, outmost redex* is always reduced *first*.
- *call-by-name* strategy
  - a *more restrictive normal order* strategy, *allowing no reduction inside abstraction*.
- *call-by-value* strategy
  - *only outermost redexes* are reduced and
  - where a redex is reduced *only when its right-hand side has already been reduced to a value*
  - **strict** in the sense that *the arguments to functions are always evaluated*, *whether or not they are used* by the body of the function.
  - reflects standard conventions found in most mainstream languages.
  - adopted in our course

Operational Semantics

- Computation rule

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

- Congruence rules

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

Programming in the Lambda Calculus

Multiple Arguments

Church Booleans

Pairs

Church Numerals

Recursion

Church Booleans

- Boolean values can be encoded as:

$$\text{tru} = \lambda t.\lambda f.\, t$$
$$\text{fls} = \lambda t.\lambda f.\, f$$

- Boolean conditional and operators can be encoded as:

$$\text{test} = \lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n$$

$$\text{not} = \lambda b.\ b\ \text{fls}\ \text{tru}$$

$$\text{and} = \lambda b.\lambda c.\ b\ c\ \text{fls}$$
$$or = \lambda a.\lambda b.\, a\ tru\ b$$

Church Numerals

- *Encoding Church numerals*
  - *Basic* idea: represent the number n by ***a function*** that "repeats ***some action*** n *times*", making numbers into ***active entities***

$$
\begin{aligned}
c_0 &= \lambda s.\ \lambda z.\ z \\
c_1 &= \lambda s.\ \lambda z.\ s\ z \\
c_2 &= \lambda s.\ \lambda z.\ s\ (s\ z) \\
c_3 &= \lambda s.\ \lambda z.\ s\ (s\ (s\ z))
\end{aligned}
$$

  - each number n is represented by *a term* c_n taking *two arguments*, s and z (for "successor" and "zero"), and applies s, n times, to z.

Multiple Arguments

- In general, $\lambda x.\ \lambda y.\ t$ is a function that, given a value v for x, yields a function that, given a value u for y, yields t with v in place of x and u in place of y.
  - i.e., $\lambda x.\ \lambda y.\ t$ is a *two-argument function*.
- λ-abstraction that does nothing but *immediately yields another abstraction* — is very common in the λ-calculus.

Recursion
in the Lambda Calculus

Recursion

- Basic Idea:

  A *recursive* definition:

  h = <body containing *h*>

First try: Self-application function: Divergence

$$\text{Omega} = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

- Note that omega evaluates *in one step* to *itself* !
  – evaluation of omega **never reaches a normal form**: it diverges.

- Terms with no normal form are said to diverge.

- Divergent computation does not seem very useful in itself. However, there are **variants** of omega that are **very useful** ...

Recursion

- Suppose f is some λ-abstraction, and consider the following term:

$$Y_f = (\lambda x. f\ (x\ x))\ (\lambda x. f\ (x\ x));$$

$$
\begin{aligned}
Y_f &= \\
&\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))} \\
&\longrightarrow \\
&f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}) \\
&\longrightarrow \\
&f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})) \\
&\longrightarrow \\
&f\ (f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}))) \\
&\longrightarrow \\
&\ldots
\end{aligned}
$$

Recursion

- Y_f is still not very useful, since (like omega), all it does is diverge.
- Is there any way we could "slow it down"?

Recursion: Delaying divergence

$$\text{delay} = \lambda y.\ \text{omega}$$

- Note that delay is a *value* — it will only diverge when actually applying it to an argument, i.e., we *can safely pass it as an argument to other functions*, return it *as a result from functions*, etc.

$$(\lambda p.\ \text{fst (pair p fls) tru) delay}$$
$$\longrightarrow$$
$$\text{fst (pair delay fls) tru}$$
$$\longrightarrow$$
$$\text{delay tru}$$
$$\longrightarrow$$
$$\text{omega}$$
$$\longrightarrow$$
$$\ldots\ldots$$

Recursion: Delaying divergence

- Here is a variant of omega in which the *delay* and *divergence* are a bit more tightly intertwined:

$$\text{omegav} = \lambda y.\left(\lambda x.\,(\lambda y.\,x\,x\,y)\right)\left(\lambda x.\,(\lambda y.\,x\,x\,y)\right)\,y$$

- Note that omegav is a normal form. However, if we apply it to any argument v, it diverges:

$$\text{omegav } v =$$
$$\underline{(\lambda y.\,(\lambda x.\,(\lambda y.\,x\,x\,y))\;\;(\lambda x.\,(\lambda y.\,x\,x\,y))\;y)\;\;v}$$
$$\longrightarrow$$
$$\underline{(\lambda x.\,(\lambda\,y.\,x\,x\,y))\;\;(\lambda x.\,(\lambda y.\,x\,x\,y))}\;\;v$$
$$\longrightarrow$$
$$\lambda y.\,(\lambda x.\,(\lambda y.\,x\,x\,y))\;\;(\lambda x.\,(\lambda\,y.\,x\,x\,y))\;y)\;\;v$$
$$=$$
$$\text{omegav}\;\;v$$

Recursion: another Delayed variant

- Suppose f is a function. Define

$$Z_f = \lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ y$$

by combining the "added f" from Y_f with the "delayed divergence" of omegav.

- Apply Z_f to an argument v, something interesting happens:

$$Z_f\ \ v =$$

$$\underline{(\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ y)}\ \ v$$

$$\longrightarrow$$

$$\underline{(\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ F\ (\lambda y.\ x\ x\ y))}\ v$$

$$\longrightarrow$$

$$f\ \ (\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ y)\ \ v$$

$$=$$

$$f\ \ Z_f\ \ v$$

Recursion: another Delayed variant

$$Z_f\ v =$$
$$(\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ y)\ \ v$$
$$\longrightarrow$$
$$(\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ F\ (\lambda y.\ x\ x\ y))\ v$$
$$\longrightarrow$$
$$f\ (\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ y)\ \ v$$
$$=$$
$$f\ Z_f\ v$$

- Since Z_f and v are both *values*, the next computation step will be **the reduction of** $f\ Z_f$ — that is, f gets to do some computation before it "diverges"

Recursion: Generic Z

If we define

$$Z = \lambda f.\ Z_f$$

i.e.,

$$Z = \lambda f.\ \lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ y$$

then we can obtain the behavior of Z_f for any f we like, simply by applying Z to f.

$$Z\ \ f \longrightarrow Z_f$$

Recursion

- Fixed-point combinator

$$\text{fix} \;=\; \lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\; (\lambda x.\, f\, (\lambda y.\, x\, x\, y));$$

$$\text{fix}\ f \;=\; f\ \ (\lambda y.\, (\text{fix}\ f)\ y)$$

- Z = λf. λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y

Z here is essentially the same as the fix given in the textbook

Recursion

- Basic Idea:

  A *recursive* definition:

  h = <body containing *h*>

  g = λf . <body containing f >

  h = fix g

Recursion

- Example:

$fac = \lambda n.\ \text{if } eq\ n\ c0$

$\quad\quad \text{then } c1$

$\quad\quad \text{else } times\ n\ (fac\ (\text{pred } n))$

$g = \lambda f\ .\ \lambda n.\ \text{if eq } n\ c0$
$\quad\quad \text{then } c1$
$\quad\quad \text{else times } n\ (f\ (\text{pred } n))$
$fac = \text{fix } g$

Exercise: Check that fac c3 → c6.

Recursion

$$\text{fix} = \lambda f.\ \big(\lambda x.\ f\ (\lambda y.\, x\, x\, y)\big)\ \big(\lambda x.\ f\ (\lambda y.\, x\, x\, y)\big)$$

$$Y_f = (\lambda x.\, f\ (x\, x))\ (\lambda x.\ f\, (x\, x));$$

- Assuming call-by-value
  - (x x) in Y_f is not a value
  - while $(\lambda y.\, x\, x\, y)$ is a value
  - Y_f will diverge for any **f**

Formalities (Formal Definitions)

Syntax (free variables)

Substitution

Operational Semantics

Syntax

- **Definition** [Terms]:

  Let $\mathcal{V}$ be a *countable set* of variable names.

  The set of terms is *the smallest set* $\mathcal{T}$ such that

  1. $x \in \mathcal{T}$ for every $x \in \mathcal{V}$;
  2. if $t_1 \in \mathcal{T}$ and $x \in \mathcal{V}$, then $\lambda x.t_1 \in \mathcal{T}$;
  3. if $t_1 \in \mathcal{T}$ and $t_2 \in \mathcal{T}$, then $t_1\ t_2 \in \mathcal{T}$.

Syntax

- **Definition:** Free Variables of term t, written as $FV(t)$:

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$$

- Please prove that |FV(t)| size(t) for every term t

Operational Semantics

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x.t}$	*abstraction*
		$\mathtt{t\ t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x.t}$	*abstraction value*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda\mathtt{x.t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-AppAbs)}$$

Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & [x \mapsto s]t_1\ [x \mapsto s]t_2 &
\end{array}
$$

Alpha-conversion: Terms that *differ only in the names of bound variables* are interchangeable *in all contexts*.

Example:

$[x \mapsto y\ z]\ (\lambda y.\ x\ y)$

$= [x \mapsto y\ z]\ (\lambda w.\ x\ w)$

$= \lambda w.\ y\ z\ w$

Chapter 6
Nameless Representation of Terms

Terms and Contexts

Shifting and Substitution

Bound Variables

- Recall that bound variables can be renamed, at any moment, to enable substitution:

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & [x \mapsto s]t_1\ [x \mapsto s]t_2 &
\end{array}
$$

- Variable Representation
  - Represent variables symbolically, with variable renaming mechanism
  - Represent variables symbolically, with bound variables are all different
  - “Canonically” represent variables in a way such that renaming is unnecessary
  - No use of variables: combinatory logic

Terms and Contexts

Nameless Terms

- *De Bruijin* idea: Replacing named variables by *natural numbers*, where the number k stands for "the variable bound by the $k'th$ enclosing λ". e.g.,
  - λx.x λ.0
  - λx.λy. x (y x) λ.λ. 1 (0 1)

  De Bruijin terms **vs** *De Bruijin indices*

- *e.g.,* the corresponding nameless term for the following:

  c0 = λs. λz. z;

  c2 = λs. λz. s (s z);

  plus = λm. λn. λs. λz. m s (n z s);

  fix = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y));

  foo = (λx. (λx. x)) (λx. x);

Nameless Terms

- Need to keep careful track of how many free variables each term may contain.

Definition [Terms]: Let $\mathcal{T}$ be *the smallest family of sets* $\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots\}$ such that

1. $k \in \mathcal{T}_n$ whenever $0 \leq k < n$;
2. if $t_1 \in \mathcal{T}_n$ and $n>0$, then $\lambda.t_1 \in \mathcal{T}_{n-1}$;
3. if $t_1 \in \mathcal{T}_n$ and $t_2 \in \mathcal{T}_n$, then $(t_1\ t_2) \in \mathcal{T}_n$.

- **Note:**
  - terms with **no free variables** are called the **0-term**s; 1-terms (one **free variables), …**
  - $\mathcal{T}_n$ are set of terms with at most n free variables, n-terms, numbered between 0 and n-1: a given element of $\mathcal{T}_n$ need not have free variables with all these numbers, or indeed any free variables at all. When t is closed, for example, it will be an element of $\mathcal{T}_n$ for every n.
  - two ordinary terms are *equivalent* modulo renaming of bound variables iff they have the same de Bruijn representation.

Name Context

- To deal with terms containing free variables, to represent

$$\lambda x.\ y\ x$$

x as a nameless term.

We know what to do with x, but we cannot see the binder for y, so it is *not clear how "far away"* it might be and we do not know what number to assign to it.

Name Context

Definition: Suppose x_0 through x_n are variable names from $\mathcal{V}$. The naming context

$\Gamma = x_n, x_{n-1}, \ldots x_1, x_0$ assigns to each x_i the *de Bruijn index* i.

Note that the *rightmost variable* in the sequence is given the index *0*; this matches the way we count *λ binders* — ***from right to left*** — when converting a named term to nameless form.

We write **dom(*Γ*)** for the set $\{x_n, \ldots x_1, x_0\}$ of variable names mentioned in Γ .

- e.g., $\Gamma = x \mapsto 4; y \mapsto 3; z \mapsto 2; a \mapsto 1; b \mapsto 0$, under this Γ, we have
  - x (y z) ? 4 (3 2)
  - λw. y w λ. 4 0
  - λw. λa. x λ. λ. 6

Shifting and Substitution

How to define substitution $[k \mapsto s]\ t$?

Shifting

- Under the naming context Γ : $x \mapsto 1, z \mapsto 2$

  $[1 \mapsto 2\ (\lambda.\ 0)]\ \lambda.\ 2 \longrightarrow\ ?$

  i.e., $[\ x \mapsto z\ (\lambda w.\ w)\ ]\ \lambda y.\ x \longrightarrow ?$

- When a substitution goes under a λ-abstraction, as in $[1 \mapsto s](\lambda.2)$ (i.e.,$[x \mapsto s](\lambda y.x)$, assuming that 1 is the index of x in the outer context), *the context* in which the substitution is taking place becomes *one variable longer than the original*;
- We need to *increment the indices* of the *free variables* in s so that they keep referring to *the same names in the new context* as they did before.
- e.g., s = 2 (λ. 0), , i.e., s = z (λw.w), assuming 2 is the index of z in the outer context, we need to shift the 2 but not the 0
- An auxiliary operation: renumber the indices of the free variables in a term.

Shifting

Definition [Shifting]: The d-place shift of a term $\texttt{t}$ above cutoff c, written $\uparrow^d_c(\texttt{t})$, is defined as follows:

$$
\begin{aligned}
\uparrow^d_c(\texttt{k}) &= \begin{cases} \texttt{k} & \text{if } k < c \\ \texttt{k} + d & \text{if } k \geq c \end{cases} \\
\uparrow^d_c(\lambda.\texttt{t}_1) &= \lambda.\ \uparrow^d_{c+1}(\texttt{t}_1) \\
\uparrow^d_c(\texttt{t}_1\ \texttt{t}_2) &= \uparrow^d_c(\texttt{t}_1)\ \uparrow^d_c(\texttt{t}_2)
\end{aligned}
$$

We write $\uparrow^d(\texttt{t})$ for $\uparrow^d_0(\texttt{t})$. □

1. What is $\uparrow^2(\lambda.\lambda.\ \texttt{1 (0 2)})$?
2. What is $\uparrow^2(\lambda.\ \texttt{0 1 } (\lambda.\ \texttt{0 1 2}))$?

Substitution

DEFINITION [SUBSTITUTION]: The substitution of a term $\mathtt{s}$ for variable number $\mathtt{j}$ in a term $\mathtt{t}$, written $[\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}$, is defined as follows:

$$
\begin{array}{lcl}
[\mathtt{j} \mapsto \mathtt{s}]\mathtt{k} & = & \begin{cases} \mathtt{s} & \text{if } \mathtt{k} = \mathtt{j} \\ \mathtt{k} & \text{otherwise} \end{cases} \\
[\mathtt{j} \mapsto \mathtt{s}](\lambda.\mathtt{t}_1) & = & \lambda.\ [\mathtt{j}{+}1 \mapsto \uparrow^1(\mathtt{s})]\mathtt{t}_1 \\
[\mathtt{j} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & ([\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}_1\ [\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}_2)
\end{array}
$$

□

$$
\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{t}_1) & = & \lambda \mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \text{if } \mathtt{y} \neq \mathtt{x} \text{ and } \mathtt{y} \notin FV(\mathtt{s}) \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 &
\end{array}
$$

Evaluation

- To define the *evaluation relation* on nameless terms, the only thing we *need to change* (i.e., the only place where *variable names* are mentioned) is the *beta-reduction rule (computation rules),* while keep the other rules identical to what as Figure 5-3.

$$(\lambda x.\, t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12},$$

- How to change the above rule for nameless representation?

Evaluation

- Example:

$$(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12},$$

$$(\lambda.t_{12})\ v_2 \longrightarrow \uparrow^{-1}([0 \mapsto \uparrow^{1}(v_2)]t_{12})$$

$$(\lambda.1\ 0\ 2)\ (\lambda.0) \longrightarrow 0\ (\lambda.0)\ 1$$

Homework

- Read Chapter 6.
  - Do Exercise 6.2.5.

6.2.5 EXERCISE [★]: Convert the following uses of substitution to nameless form, assuming the global context is Γ = a,b, and calculate their results using the above definition. Do the answers correspond to the original definition of substitution on ordinary terms from §5.3?

1. [b ↦ a] (b (λx.λy.b))
2. [b ↦ a (λz.a)] (b (λx.b))
3. [b ↦ a] (λb. b a)
4. [b ↦ a] (λa. b a) □

- Read Chapter 7 and download & digest the *fulluntyped* implementation includes extensions such as numbers and booleans.

Evaluation

- $(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12},$

$$(\lambda.t_{12})\ v_2 \quad \longrightarrow \quad \uparrow^{-1}([0 \mapsto \uparrow^{1}(v_2)]t_{12})$$

$$(\lambda.1\ 0\ 2)\ (\lambda.0) \longrightarrow 0\ (\lambda.0)\ 1$$