Type Systems

Lecture 2 Oct. 27th, 2004
Sebastian Maneth

http://lampwww.epfl.ch/teaching/typeSystems/2004

Today

1. What is the Lambda Calculus?
2. Its Syntax and Semantics
3. Church Booleans and Church Numerals
4. Lazy vs. Eager Evaluation (call-by-name vs. call-by-value)
5. Recursion
6. Nameless Implementation: deBruijn Indices

1. What is the Lambda Calculus

introduced in late 1930's by Alonzo Church and Stephen Kleene

used in 1936 by Church to prove the undecidability of the Entscheidungsproblem

is a formal system designed to investigate

- function definition
- function application
- recursion

1. What is the Lambda Calculus

introduced in late 1930's by Alonzo Church and Stephen Kleene

can compute the same as Turing Machines, which is everything we can (intuitively) compute (Church-Turing Thesis).

is a formal system designed to investigate

- function definition
- function application
- recursion

1. What is the Lambda Calculus

what do we want?

→ a small core language, into which other language constructs can be translated.

There are many such languages:

Turing Machines
μ–Recursive Functions
Chomsky's Type-0 Grammars
Cellular Automata
etc.

→ why do we pick out the Lambda Calulus?

because types are about values of **program variables.**

2. Syntax of the Lambda Calculus

Let V be a countable set of variable names.

The set of lambda terms (over V) is the smallest set T such that

1. if $x \in V$, then $\boxed{x \in T}$ variable
2. if $x \in V$ and $t_1 \in T$, then $\boxed{\lambda x.\ t_1 \in T}$ abstraction
3. if $t_1, t_2 \in T$, then $\boxed{t_1\ t_2 \in T}$ application

Function abstraction: instead of $f(x) = x + 5$

write $f = \boxed{\lambda x.x + 5}$

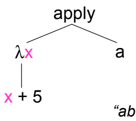
a lambda term (i.e., $\in T$)
representing a *nameless function*, which adds 5 to its parameter

2. Syntax of the Lambda Calculus

Function application: instead of $f(x)$

write $f\ x$

Example: $(\lambda x.\ x + 5)\ a$



Abstract Syntax Tree

(AST)

"surface syntax" *"abstract syntax"*

Conventions (to save parenthesis)

application is **left** associative: $x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$

scope of **abstraction** extends **as far to the right as possible**:

$\lambda x.\ x\ y = \lambda x.\ (x\ y) \neq (\lambda\ x.\ x)\ y$

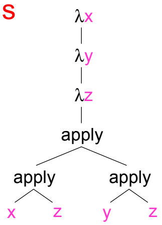
2. Syntax of the Lambda Calculus

Example: $\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z)$

$= \lambda x.(\lambda y.\lambda z.\ x\ z\ (y\ z))$

$= \lambda x.(\lambda y.(\lambda z.\ (x\ z\ (y\ z))$

$= \lambda x.\ (\lambda y.\ (\lambda z.\ ((x\ z)\ (y\ z))))$



"surface syntax" *"abstract syntax"*

Conventions (to save parenthesis)

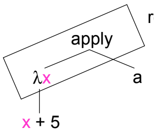
application is **left** associative: $x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$

scope of **abstraction** extends **as far to the right as possible**:

$\lambda x.\ x\ y = \lambda x.\ (x\ y) \neq (\lambda\ x.\ x)\ y$

2. Semantics of the Lambda Calculus

$(\lambda x.\underbrace{x + 5}) a$ SUBSTITUTE a for x in $x + 5$



redex (REDucible EXpression): $(\lambda x.\, t)\ t_1$

can be reduced to (evaluates to):

$\xrightarrow{\beta\text{–reduction}}$ $[x \rightarrow a]\ x + 5 \quad = \quad a + 5$

To compute in Lambda Calculus, ALL you do is SUBSTITUTE!!

---

2. Semantics of the Lambda Calculus

Example:

$(\lambda x.\lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x)$

---

2. Semantics of the Lambda Calculus

Example:

$(\lambda x.\lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x)$

$= ((\lambda x.\lambda y.\ f\ (y\ x))\ 5)\ (\lambda x.\ x)$ because App binds to the left!

---

2. Semantics of the Lambda Calculus

Example:

$(\lambda x.\lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x)$

$= ((\lambda x.\lambda y.\ f\ (y\ x))\ 5)\ (\lambda x.\ x)$ because App binds to the left!

$\xrightarrow{\beta\text{–red.}}$ $[x \rightarrow 5](\lambda y.\ f\ (y\ x))\ (\lambda x.\ x)$

$= (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x)$

2. Semantics of the Lambda Calculus

Example:

$(\lambda x.\lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x)$

$= ((\lambda x.\lambda y.\ f\ (y\ x))\ 5)\ (\lambda x.\ x)$ because App binds to the left!

$\xrightarrow{\beta\text{–red.}}\ [\,x \to 5\,](\lambda y.\ f\ (y\ x))\ (\lambda x.\ x)$

$= (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x)$

$\xrightarrow{\beta\text{–red.}}\ [\,y \to \lambda x.\ x\,](\ f\ (y\ 5)\ )$

$= f\ (\lambda x.\ x\ \ 5)$

---

2. Semantics of the Lambda Calculus

Example:

$(\lambda x.\lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x)$

$= ((\lambda x.\lambda y.\ f\ (y\ x))\ 5)\ (\lambda x.\ x)$ because App binds to the left!

$\xrightarrow{\beta\text{–red.}}\ [\,x \to 5\,](\lambda y.\ f\ (y\ x))\ (\lambda x.\ x)$

$= (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x)$

$\xrightarrow{\beta\text{–red.}}\ [\,y \to \lambda x.\ x\,](\ f\ (y\ 5)\ )$

$= f\ (\lambda x.\ x\ \ 5)$

$\xrightarrow{\beta\text{–red.}}\ f\ 5$ (normal form = cannot be reduced further)

---

2. Semantics of the Lambda Calculus

Example: Does every λ-term have a normal form?

→ NO!!!

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\xrightarrow{\beta\text{–red.}}\ [\,x \to (\lambda x.\ x\ x)\,]\ (x\ x)$

---

2. Semantics of the Lambda Calculus

Example: Does every λ-term have a normal form?

→ NO!!!

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\xrightarrow{\beta\text{–red.}}\ [\,x \to (\lambda x.\ x\ x)\,]\ (x\ x)$

$= (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

2. Semantics of the Lambda Calculus

Example: Does every λ-term have a normal form?

→ NO!!!

$$(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$$

$$\xrightarrow{\beta\text{–red.}} \;[\, x \rightarrow (\lambda x.\, x\, x)\,]\;(x\, x)$$

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

$$\xrightarrow{\beta\text{–red.}} \;(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$$

$$\xrightarrow{\beta\text{–red.}} \;(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$$

$$\vdots$$

---

2. Semantics of the Lambda Calculus

Example: Does every λ-term have a normal form?

→ NO!!!

$\underbrace{(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)}_{=:\ \text{omega}}$ is called the omega combinator

combinator = closed lambda term

= lambda term with *no free variables*

The simplest combinator, identity: `id` := $\lambda x.\, x$

---

2. Semantics of the Lambda Calculus

Free vs. Bound Variables:

$$\lambda x.\, x\, y \;=\; \lambda x.\boxed{(\overset{\text{bound}}{x}\ \overset{\text{free}}{y})}$$

scope of x
x is bound in its scope

Define the set of free variables of a term t, FV(t), as

if $t = x \in V$, then $FV(t) = \{ x \}$

if $t = \lambda x.\, t_1$, then $FV(t) = FV(t_1) \setminus \{ x \}$

if $t = t_1\ t_2$, then $FV(t) = FV(t_1) \cup FV(t_2)$

---

3. Church Booleans and Numerals

How to encode BOOLEANS into the lambda calculus?

`tru` → takes two arguments, selects the FIRST
`fls` → takes two arguments, selects the SECOND

THEN: if-then-else can be defined as:

`test x u w` = "apply x to u w"

$$= \underbrace{(\lambda k.\, \lambda m.\, \lambda n.\, k\, m\, n)}_{=:\ \texttt{test}}\ x\ u\ w$$

`tru` := $\lambda m.\, \lambda n.\, m$
`fls` := $\lambda m.\, \lambda n.\, n$

$$\texttt{test tru u w} \xrightarrow{\beta\text{–red.}} \ldots \xrightarrow{\beta\text{–red.}} u$$

3. Church Booleans and Numerals

How to encode BOOLEANS into the lambda calculus?

tru → takes two arguments, selects the FIRST
fls → takes two arguments, selects the SECOND

tru := λm. λn. m
fls := λm. λn. n
test := λk. λm. λn. k m n

How to do "and" on these BOOLEANS?

and u w = "apply u to w fls"

$$:= \underbrace{(\lambda m.\ \lambda n.\ m\ n\ \texttt{fls})}_{=:\ \texttt{and}}\ u\ w$$

---

3. Church Booleans and Numerals

How to encode BOOLEANS into the lambda calculus?

tru → takes two arguments, selects the FIRST
fls → takes two arguments, selects the SECOND

tru := λm. λn. m
fls := λm. λn. n
test := λk. λm. λn. k m n

How to do "and" on these BOOLEANS?

and u w = "apply u to w fls"

$$:= \underbrace{(\lambda m.\ \lambda n.\ m\ n\ \texttt{fls})}_{=:\ \texttt{and}}\ u\ w$$

→ **Define** the or and not functions!

---

3. Church Booleans and Numerals

How to encode NUMBERS into the lambda calculus?

c0 := λs. λz. z
c1 := λs. λz. s z
c2 := λs. λz. s (s z)
c3 := λs. λz. s (s (s z))
etc.

THEN, the successor function can be defined as

scc := λn. λs. λz. s (n s z)

$$\texttt{scc c0} \xrightarrow{\beta\text{–red.}} \lambda s.\ \lambda z.\ s\ (\texttt{c0}\ s\ z) \xrightarrow{\beta\text{–red.}} \lambda s.\ \lambda z.\ s\ z\ =\ \texttt{c1}$$

just like fls!
Select the second argument.

---

3. Church Booleans and Numerals

How to encode NUMBERS into the lambda calculus?

c0 := λs. λz. z
c1 := λs. λz. s z
c2 := λs. λz. s (s z)
c3 := λs. λz. s (s (s z))

scc := λn. λs. λz. s (n s z)

How to do "plus" and "times" on these Church Numerals?

plus := λm. λn. λs. λz. m s (n s z)

"apply m times the successor to n"

3. Church Booleans and Numerals

How to encode NUMBERS into the lambda calculus?

`c0` := λs. λz. z
`c1` := λs. λz. s z
`c2` := λs. λz. s (s z)
`c3` := λs. λz. s (s (s z))

`scc` := λn. λs. λz. s (n s z)

How to do “`plus`” and “`times`” on these Church Numerals?

`plus` := λm. λn. λs. λz. m s (n s z)
↑
“apply m times the successor to n”

`times` := λm. λn. m (`plus` n) `c0`
↑
“apply m times (`plus` n) to `c0`”

---

3. Church Booleans and Numerals

How to encode NUMBERS into the lambda calculus?

`c0` := λs. λz. z
`c1` := λs. λz. s z
`c2` := λs. λz. s (s z)
`c3` := λs. λz. s (s (s z))

`scc` := λn. λs. λz. s (n s z)
`plus` := λm. λn. λs. λz. m s (n s z)

Questions:

1. Write a function `subt` for subtraction on Church Numerals.
2. How can other datatypes be encoded into the lambda calculus, like, e.g., lists, trees, arrays, and variant records?

---

4. Lazy vs. Eager Evaluation

What does this lambda term evaluate to??

`tru id omega`

---

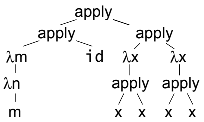
4. Lazy vs. Eager Evaluation

What does this lambda term evaluate to??

`tru id omega`

(λm. λn. m) (λx. x) ((λx. x x) (λx. x x))

→ where to start evaluating? which redex??



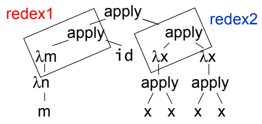
4. Lazy vs. Eager Evaluation

What does this lambda term evaluate to??

`tru id omega`

(λm. λn. m) (λx. x) ((λx. x x) (λx. x x))

→ where to start evaluating? which redex??



---

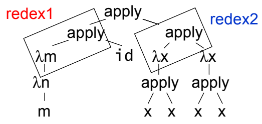
4. Lazy vs. Eager Evaluation

What does this lambda term evaluate to??

`tru id omega`

(λm. λn. m) (λx. x) ((λx. x x) (λx. x x))

→ where to start evaluating? which redex??

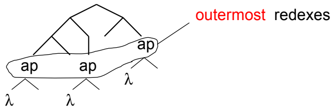


→ if we always reduce redex2 then this lambda term has NO semantics.

---

4. Lazy vs. Eager Evaluation

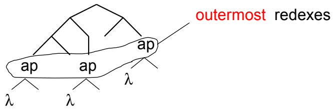
A redex if outermost, if in the AST it has no ancestor that is a redex.



---

4. Lazy vs. Eager Evaluation

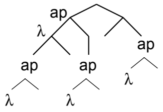
A redex if outermost, if in the AST it has no ancestor that is a redex.



A redex if leftmost, if in the AST it has no redex to the left of it.

4. Lazy vs. Eager Evaluation

A redex if outermost, if in the AST it has no ancestor that is a redex.



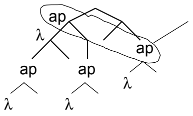
outermost redexes

A redex if leftmost, if in the AST it has no redex to the left of it.

---

4. Lazy vs. Eager Evaluation

A redex if outermost, if in the AST it has no ancestor that is a redex.



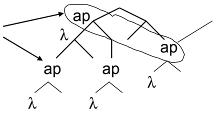
outermost redexes

A redex if leftmost, if in the AST it has no redex to the left of it.

---

4. Lazy vs. Eager Evaluation

A redex if outermost, if in the AST it has no ancestor that is a redex.



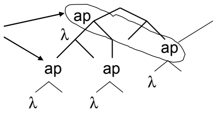
leftmost

outermost redexes

A redex if leftmost, if in the AST it has no redex to the left of it.

---

4. Lazy vs. Eager Evaluation



leftmost

outermost redexes

Evaluation Strategies:

lazy	**normal order**	always reduce leftmost outermost redex first
	call-by-name	like normal order, but NOT inside abstractions
	call-by-need	like call-by-name but with sharing
eager	**call-by-value**	reduce only “value-redexes” (= argument is a value) and do this leftmost

right branch of ap

4. Lazy vs. Eager Evaluation

Lazy seems better than **eager**, because more terms can be evaluated!

→ can you define an infinite list consisting of all prime numbers?
(with **lazy** evaluation you can fetch the first n numbers of this list!)

```
> fetch c3 primelist
```

should compute the list [2, 3, 5]

If a term evaluates to a normal form n using **eager** evaluation,
then it also evaluates to n using **lazy** evaluation.

→ can you prove this?!?
→ what about the number of eval. steps needed by **eager** vs. **lazy**?

Lazy is hard to implement efficiently because copies of unevaluated lambda terms must be shared in order not to have duplicate reductions

4. Lazy vs. Eager Evaluation

Lazy seems better than **eager**, because more terms can be evaluated!

→ can you define an infinite list consisting of all prime numbers?
(with **lazy** evaluation you can fetch the first n numbers of this list!)

```
> fetch c3 primelist
```

should compute the list [2, 3, 5]

If a term evaluates to a normal form n using **eager** evaluation,
then it also evaluates to n using **lazy** evaluation.

→ can you prove this?!?
→ what about the number of eval. steps needed by **eager** vs. **lazy**?

Lazy is hard to implement it efficiently because lots of duplicate reductions might be done.

→ Most FL's use call-by-value. Also the TaPL book!

5. Recursion

```
fct = λn.if eq n c0 then c1 else (times n (fct (prd n)))
```

↑ recursion

e.g. fct c3 needs to unroll 4 times the definition
(expand)

```
fct c3 = if eq c3 c0 then c1 else (times c3 (
           if eq c2 c0 then c1 else (times c2 (
             if eq c1 c0 then c1 else (times c1 (
               if eq c0 c0 then c1 else (..)..)
```

(evaluates to c6)

5. Recursion

```
fct = λn.if eq n c0 then c1 else (times n (fct (prd n)))
```

↑ recursion

e.g. fct c3 needs to unroll 4 times the definition
(expand)

```
fct c3 = if eq c3 c0 then c1 else (times c3 (
           if eq c2 c0 then c1 else (times c2 (
             if eq c1 c0 then c1 else (times c1 (
               if eq c0 c0 then c1 else (..)..)
```

(evaluates to c6)

→ Is there a combinator doing the unrolling, when applied to `fct`?

5. Recursion

`fct = λn.if eq n c0 then c1 else (times n (fct (prd n)))`

↑ recursion

such a combinator is similar to omega! (λx. x x) (λx. x x)

→ additionally to copying itself it should each time

split of one application of the definition of `fct`

→ Is there a combinator doing the unrolling, when applied to `fct`?

---

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

`g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))`

Y g c3 →

---

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

`g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))`

$$Y\ g\ c3 \rightarrow (\lambda x.\ g\ (x\ x))\ \underbrace{(\lambda x.\ g\ (x\ x))}_{=:\ h}\ c3$$

→ g (h h) c3

---

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

`g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))`

$$Y\ g\ c3 \rightarrow (\lambda x.\ g\ (x\ x))\ \underbrace{(\lambda x.\ g\ (x\ x))}_{=:\ h}\ c3$$

→ g (h h) c3

lazy!→ `λn.if eq n c0 then c1 else (times n (h h (prd n))) c3`

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))

$$Y\ g\ c3 \rightarrow (\lambda x.\ g\ (x\ x))\ \underbrace{(\lambda x.\ g\ (x\ x))}_{=:\ h}\ c3$$

→ g (h h) c3 **eager!** → g (g (h h)) c3 → g(g(g(h h) c3 …

lazy!→ λn.if eq n c0 then c1 else (times n (h h (prd n))) c3

---

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))

$$Y\ g\ c3 \rightarrow (\lambda x.\ g\ (x\ x))\ \underbrace{(\lambda x.\ g\ (x\ x))}_{=:\ h}\ c3$$

→ g (h h) c3

lazy!→ λn.if eq n c0 then c1 else (times n (h h (prd n))) c3

→ if eq c3 c0 then c1 else (times c3 (h h (prd c3)))

---

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))

$$Y\ g\ c3 \rightarrow (\lambda x.\ g\ (x\ x))\ \underbrace{(\lambda x.\ g\ (x\ x))}_{=:\ h}\ c3$$

→ g (h h) c3

lazy!→ λn.if eq n c0 then c1 else (times n (h h (prd n))) c3

→ if eq c3 c0 then c1 else (times c3 (h h (prd c3)))

→ times c3 (h h (prd c3))

---

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))

$$Y\ g\ c3 \rightarrow (\lambda x.\ g\ (x\ x))\ \underbrace{(\lambda x.\ g\ (x\ x))}_{=:\ h}\ c3$$

→ g (h h) c3

lazy!→ λn.if eq n c0 then c1 else (times n (h h (prd n))) c3

→ if eq c3 c0 then c1 else (times c3 (h h (prd c3)))

→ times c3 (h h (prd c3))

→ times c3 (g (h h) (prd c3))

5. Recursion

First, under call-by-name (**lazy**) evaluation.

(cbn) fixed-point combinator Y := λf. (λx. f (x x)) (λx. f (x x))

g := λfct. λn.if eq n c0 then c1 else (times n (fct (prd n)))

$$Y\ g\ c3 \to (\lambda x.\ g\ (x\ x))\ \underbrace{(\lambda x.\ g\ (x\ x))}_{=:\ h}\ c3$$

→ g (h h) c3

lazy!→ λn.if eq n c0 then c1 else (times n (h h (prd n))) c3

→ if eq c3 c0 then c1 else (times c3 (h h (prd c3)))

→ times c3 (h h (prd c3))

→ times c3 (g (h h) (prd c3)) → … → times c3 c2 c1 c1

---

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

fix g c3 →

---

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

$$\text{fix}\ g\ c3 \to (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ \underbrace{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))}_{=:\ h}\ c3$$

---

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

$$\text{fix}\ g\ c3 \to (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ \underbrace{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))}_{=:\ h}\ c3$$

→ g (**λy**. h h y) c3 “λ-guard”

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

$$\text{fix } g\ c3 \rightarrow (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ \underbrace{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))}_{=:\ h}\ c3$$

→ g (λy. h h y) c3 “λ-guard”

→ `λn.if eq n c0 then c1 else (times n ((λy. h h y)(prd n)))  c3`

---

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

$$\text{fix } g\ c3 \rightarrow (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ \underbrace{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))}_{=:\ h}\ c3$$

→ g (λy. h h y) c3 “λ-guard”

→ `λn.if eq n c0 then c1 else (times n ((λy. h h y)(prd n)))  c3`

→ `if eq c3 c0 then c1 else (times c3 ((λy. h h y)(prd c3)))`

---

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

$$\text{fix } g\ c3 \rightarrow (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ \underbrace{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))}_{=:\ h}\ c3$$

→ g (λy. h h y) c3 “λ-guard”

→ `λn.if eq n c0 then c1 else (times n ((λy. h h y)(prd n)))  c3`

→ `if eq c3 c0 then c1 else (times c3 ((λy. h h y)(prd c3)))`

→ `times c3 ((λy. h h y)(prd c3))`

---

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

$$\text{fix } g\ c3 \rightarrow (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ \underbrace{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))}_{=:\ h}\ c3$$

→ g (λy. h h y) c3 “λ-guard”

→ `λn.if eq n c0 then c1 else (times n ((λy. h h y)(prd n)))  c3`

→ `if eq c3 c0 then c1 else (times c3 ((λy. h h y)(prd c3)))`

→ `times c3 ((λy. h h y)(prd c3))` $\xrightarrow{\textit{“unguard”}}$ `times c3 h h (prd c3)`

5. Recursion

Now, under **eager** (call-by-value) evaluation.

(cbv) fixed-point combinator fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

$$\text{fix g c3} \to (\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ \underbrace{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))}_{=:\ h}\ \text{c3}$$

→ g (λy. h h y) c3 “λ-guard”

→ λn.if eq n c0 then c1 else (times n ((λy. h h y)(prd n))) c3

→ if eq c3 c0 then c1 else (times c3 ((λy. h h y)(prd c3)))

$$\to \text{times c3 } ((\lambda y.\ h\ h\ y)(\text{prd c3})) \overset{\text{“unguard”}}{\to} \text{times c3 h h (prd c3)}$$

→ times c3 g (λy. h h y) (prd c3) → … → times c3 c2 c1 c1

---

5. Recursion

Question: Can you feel why the lambda calculus is Turing complete?

Can you prove it? What does it take to be Turing complete?

---

6. Nameless Implementation: deBruijn Indices

redex (REDucible EXpression): (λx. t) s

β–reduction: (λx. t) s := [x → s] t

substitution [x → s] :
A. only replace the FREE occurrences of x in t!!
B. if replacing within (λy.u) then y should NOT be FREE in s!!

DEFINE [x → s] t, by induction on the structure of t:

---

6. Nameless Implementation: deBruijn Indices

redex (REDucible EXpression): (λx. t) s

β–reduction: (λx. t) s := [x → s] t

substitution [x → s] :
A. only replace the FREE occurrences of x in t!!
B. if replacing within (λy.u) then y should NOT be FREE in s!!

DEFINE [x → s] t, by induction on the structure of t:

1. [x → s] y =
2. [x → s] λy. t_1 =
3. [x → s] t_1 t_2 =

6. Nameless Implementation: deBruijn Indices

redex (REDucible EXpression): $(\lambda x.\ t)\ s$

β–reduction: $(\lambda x.\ t)\ s\ :=\ [x \rightarrow s]\ t$

substitution $[x \rightarrow s]$: **A**. only replace the FREE occurrences of x in t!!
B. if replacing within (λy.u) then y should NOT be FREE in s!!

DEFINE $[x \rightarrow s]\ t$, by induction on the structure of t:

1. $[x \rightarrow s]\ y$ = s if y=x, and y otherwise
2. $[x \rightarrow s]\ \lambda y.\ t_1$ =
3. $[x \rightarrow s]\ t_1\ t_2$ =

---

6. Nameless Implementation: deBruijn Indices

redex (REDucible EXpression): $(\lambda x.\ t)\ s$

β–reduction: $(\lambda x.\ t)\ s\ :=\ [x \rightarrow s]\ t$

substitution $[x \rightarrow s]$: **A**. only replace the FREE occurrences of x in t!!
B. if replacing within (λy.u) then y should NOT be FREE in s!!

DEFINE $[x \rightarrow s]\ t$, by induction on the structure of t:

1. $[x \rightarrow s]\ y$ = s if y=x, and y otherwise
2. $[x \rightarrow s]\ \lambda y.\ t_1 = \lambda y.\ [x \rightarrow s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$ **A**,**B**
3. $[x \rightarrow s]\ t_1\ t_2$ =

---

6. Nameless Implementation: deBruijn Indices

redex (REDucible EXpression): $(\lambda x.\ t)\ s$

β–reduction: $(\lambda x.\ t)\ s\ :=\ [x \rightarrow s]\ t$

substitution $[x \rightarrow s]$: **A**. only replace the FREE occurrences of x in t!!
B. if replacing within (λy.u) then y should NOT be FREE in s!!

DEFINE $[x \rightarrow s]\ t$, by induction on the structure of t:

1. $[x \rightarrow s]\ y$ = s if y=x, and y otherwise
2. $[x \rightarrow s]\ \lambda y.\ t_1 = \lambda y.\ [x \rightarrow s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$ **A**,**B**
3. $[x \rightarrow s]\ t_1\ t_2 = ([x \rightarrow s]\ t_1)\ ([x \rightarrow s]\ t_2)$

---

6. Nameless Implementation: deBruijn Indices

redex (REDucible EXpression): $(\lambda x.\ t)\ s$

β–reduction: $(\lambda x.\ t)\ s\ :=\ [x \rightarrow s]\ t$

substitution $[x \rightarrow s]$: **A**. only replace the FREE occurrences of x in t!!
B. if replacing within (λy.u) then y should NOT be FREE in s!!

DEFINE $[x \rightarrow s]\ t$, by induction on the structure of t:

1. $[x \rightarrow s]\ y$ = s if y=x, and y otherwise
2. $[x \rightarrow s]\ \lambda y.\ t_1 = \lambda y.\ [x \rightarrow s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$ **A**,**B**
3. $[x \rightarrow s]\ t_1\ t_2 = ([x \rightarrow s]\ t_1)\ ([x \rightarrow s]\ t_2)$

➔ to appy 2., renaming of BOUND y's in t_1 might be necessary!!!
= "alpha-conversion"

6. Nameless Implementation: deBruijn Indices

Idea: let variable occurrences directly point to their binders,
rather than referring to them by name.

→ use natural numbers k, meaning "the k-th enclosing λ"

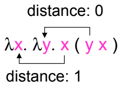
e.g. λx. λy. x (y x) BECOMES λ. λ. 1 (0 1)

---

6. Nameless Implementation: deBruijn Indices

Idea: let variable occurrences directly point to their binders,
rather than referring to them by name.

→ use natural numbers k, meaning "the k-th enclosing λ"



distance: 0

e.g. λx. λy. x (y x) BECOMES λ. λ. 1 (0 1)

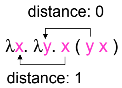
distance: 1

---

6. Nameless Implementation: deBruijn Indices

Idea: let variable occurrences directly point to their binders,
rather than referring to them by name.

→ use natural numbers k, meaning "the k-th enclosing λ"



distance: 0

e.g. λx. λy. x (y x) BECOMES λ. λ. 1 (0 1)

distance: 1

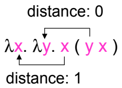
Then, every CLOSED term has a unique deBruijn representation!

---

6. Nameless Implementation: deBruijn Indices

Idea: let variable occurrences directly point to their binders,
rather than referring to them by name.

→ use natural numbers k, meaning "the k-th enclosing λ"



distance: 0

e.g. λx. λy. x (y x) BECOMES λ. λ. 1 (0 1)

distance: 1

Then, every CLOSED term has a unique deBruijn representation!

→ what to do with free variables??

use naming context $\Gamma \in V^*$. E.g., bca means b↔2, c↔1, a↔0

6. Nameless Implementation: deBruijn Indices

fix a naming context $\Gamma \in V^*$.

$$\text{lambda term} \underset{\text{restorenames}_\Gamma}{\overset{\text{removenames}_\Gamma}{\rightleftarrows}} \text{nameless lambda term}$$

$$(\Gamma = xu) \quad \lambda y.\ u\ y \underset{\text{resto}_\Gamma \longrightarrow}{\longleftarrow \text{remov}_\Gamma} \lambda.\ 1\ 0 \quad (\Gamma' = xuy)$$

substitution $\underset{\Gamma=xu}{[\,1 \rightarrow s\,]}(\lambda.\ \overset{}{2}\,)$ ↑ $\Gamma'=\Gamma y$ → increment all free vars in s by one!

$[\,j \rightarrow s\,](\lambda.\ t_1) \;=\; \lambda.\ [\,j+1 \rightarrow \text{shift}(1, s)\,]\ t_1$

shift function must keep track of BOUND vars in order
to ONLY shift the FREE vars.

---

6. Nameless Implementation: deBruijn Indices

$\text{shift}(d, s) \;:=\; \text{shiftb}(d, 0, s)$ ↑ DON'T shift vars with index <0 !!

substitution $\underset{\Gamma=xu}{[\,1 \rightarrow s\,]}(\lambda.\ 2\,)$ ↑ $\Gamma'=\Gamma y$ → increment all free vars in s by one!

$[\,j \rightarrow s\,](\lambda.\ t_1) \;=\; \lambda.\ [\,j+1 \rightarrow \text{shift}(1, s)\,]\ t_1$

shift function must keep track of BOUND vars in order
to ONLY shift the FREE vars.

---

6. Nameless Implementation: deBruijn Indices

$\text{shift}(d, s) \;:=\; \text{shiftb}(d, 0, s)$ ↑ DON'T shift vars with index <0 !!

$\text{shiftb}(d, b, k) \;=\; k$ if $k<b$, and $k+d$ otherwise
$\text{shiftb}(d, b, \lambda.\ t_1) \;=\; \lambda.\ \text{shiftb}(d, b+1, t_1)$
$\text{shiftb}(d, b, t_1\ t_2) \;=\; \text{shiftb}(d, b, t_1)\ \text{shiftb}(d, b, t_2)$

substitution $\underset{\Gamma=xu}{[\,1 \rightarrow s\,]}(\lambda.\ 2\,)$ ↑ $\Gamma'=\Gamma y$ → increment all free vars in s by one!

$[\,j \rightarrow s\,](\lambda.\ t_1) \;=\; \lambda.\ [\,j+1 \rightarrow \text{shift}(1, s)\,]\ t_1$

shift function must keep track of BOUND vars in order
to ONLY shift the FREE vars.

---

6. Nameless Implementation: deBruijn Indices

fix a naming context $\Gamma \in V^*$.

$\text{removenames}(\Gamma, x) \;=\;$ index of rightmost x in Γ
$\text{removenames}(\Gamma, \lambda x.\ t_1) \;=\; \lambda.\ \text{removenames}(\Gamma x, t_1)$
$\text{removenames}(\Gamma, t_1\ t_2) \;=\; \text{removenames}(\Gamma, t_1)\ \text{removenames}(\Gamma, t_2)$

$\text{restorenames}(\Gamma, k) \;=\;$ k-th name in Γ
$\text{restorenames}(\Gamma, \lambda.\ t) \;=\; \lambda x.\ \text{restorenames}(\Gamma x, t_1)$
x is the first name not in Γ
$\text{restorenames}(\Gamma, t_1\ t_2) \;=\; \text{restorenames}(\Gamma, t_1)\ \text{restorenames}(\Gamma, t_2)$