Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Natural Semantics

- Aka Structural Operational Semantics, aka "Big Step Semantics"
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$$(C, m) \Downarrow m'$$

or

$$(E, m) \Downarrow v$$

Simple Imperative Programming Language

- $I \in$ *Identifiers*
- $N \in$ *Numerals*
- B ::= true | false | B & B | B or B | not B | $E < E$ | $E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E \mid (E)$
- C ::= skip | C;C | I := E | if B then C else C fi | while B do C od

Natural Semantics of Atomic Expressions

- Identifiers: $(I,m) \Downarrow m(I)$
- Numerals are values: $(N,m) \Downarrow N$
- Booleans: (true,m) $\Downarrow$ true
  (false ,m) $\Downarrow$ false

Booleans:

$$\frac{(B, m) \Downarrow \text{false}}{(B \,\&\, B', m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \,\&\, B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \text{ or } B', m) \Downarrow \text{true}} \qquad \frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(\text{not } B, m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$

Relations

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b}{(E \sim E', m) \Downarrow b}$$

- By $U \sim V = b$, we mean does (the meaning of) the relation $\sim$ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V

Arithmetic Expressions

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U\ op\ V = N}{(E\ op\ E', m) \Downarrow N}$$

where *N* is the specified value for *U op V*

Commands

Skip: $(\text{skip}, m) \Downarrow m$

Assignment: $$\frac{(E,m) \Downarrow V}{(I{:=}E,m) \Downarrow m[I \text{ <-- } V]}$$ (={I -> V}+m)

Sequencing: $$\frac{(C,m) \Downarrow m' \quad (C',m') \Downarrow m''}{(C;C', m) \Downarrow m''}$$

If Then Else Command

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

$$\frac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

While Command

$$\frac{(B,m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m)\Downarrow\text{true} \quad (C,m)\Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m')\Downarrow m''}{(\text{while } B \text{ do C od}, m) \Downarrow m''}$$

Example: If Then Else Rule

$$\frac{}{(\text{if x > 5 then y:= 2 + 3 else y:=3 + 4 fi}, \{x \text{ -> } 7\}) \Downarrow ?}$$

Example: If Then Else Rule

$$\frac{\frac{}{(x > 5, \{x \text{ -> } 7\})\Downarrow ?}}{(\text{if x > 5 then y:= 2 + 3 else y:=3 + 4 fi}, \{x \text{ -> } 7\}) \Downarrow ?}$$

Example: Arith Relation

? > ? = ?
(x,{x->7})⇓? (5,{x->7})⇓?
(x > 5, {x -> 7})⇓?
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Identifier(s)

7 > 5 = true
(x,{x->7})⇓7 (5,{x->7})⇓5
(x > 5, {x -> 7})⇓?
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Arith Relation

7 > 5 = true
(x,{x->7})⇓7 (5,{x->7})⇓5
(x > 5, {x -> 7})⇓true
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: If Then Else Rule

7 > 5 = true
(x,{x->7})⇓7 (5,{x->7})⇓5 (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true ⇓ ? .
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Assignment

(2+3, {x->7})⇓?
7 > 5 = true
(x,{x->7})⇓7 (5,{x->7})⇓5 (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true ⇓ ?
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Arith Op

? + ? = ?
(2,{x->7})⇓? (3,{x->7}) ⇓?
7 > 5 = true (2+3, {x->7})⇓?
(x,{x->7})⇓7 (5,{x->7})⇓5 (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true ⇓? .
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Numerals

2 + 3 = 5
(2,{x->7})⇓2 (3,{x->7}) ⇓3
7 > 5 = true (2+3, {x->7})⇓?
(x,{x->7})⇓7 (5,{x->7})⇓5 (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true ⇓ ?
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Arith Op

2 + 3 = 5
(2,{x->7})⇓2 (3,{x->7}) ⇓3
7 > 5 = true (2+3, {x->7})⇓5
(x,{x->7})⇓7 (5,{x->7})⇓5 (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true ⇓?
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Assignment

2 + 3 = 5
(2,{x->7})⇓2 (3,{x->7}) ⇓3
7 > 5 = true (2+3, {x->7})⇓5
(x,{x->7})⇓7 (5,{x->7})⇓5 (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true ⇓ {x->7, y->5}
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓?

Example: If Then Else Rule

2 + 3 = 5
(2,{x->7})⇓2 (3,{x->7}) ⇓3
7 > 5 = true (2+3, {x->7})⇓5
(x,{x->7})⇓7 (5,{x->7})⇓5 (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true ⇓ {x->7, y->5}
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ {x->7, y->5}

Let in Command

(*E*,*m*) ⇓v (*C*,*m*[*I*<-*v*]) ⇓ *m'*
(let *I* = *E* in *C*, *m*) ⇓ *m''*

Where *m''* (*y*) = *m'* (*y*) for *y*≠ *I* and
m'' (*I*) = *m* (*I*) if *m*(*I*) is defined,
and *m''* (*I*) is undefined otherwise

Example

(x,{x->5}) ⇓ 5 (3,{x->5}) ⇓ 3
(x+3,{x->5}) ⇓ 8
(5,{x->17}) ⇓ 5 (x:=x+3,{x->5}) ⇓ {x->8}
(let x = 5 in (x:=x+3), {x -> 17}) ⇓ ?

Example

$$\frac{(5,\{x\text{->}17\}) \Downarrow 5 \quad \dfrac{\dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x:=x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}}{(\text{let } x = 5 \text{ in } (x:=x+3), \{x \text{ -> } 17\}) \Downarrow \{x\text{->}17\}}$$

Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explictly*
- Clash of constructs apparent in awkward semantics

Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Interpreter

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations

Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next "state"
  - To get final value, put in a loop

Natural Semantics Example

- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- …
- compute_com(IfExp(b,c1,c2),m) =

  if compute_exp (b,m) = Bool(true)

  then compute_com (c1,m)

  else compute_com (c2,m)

Natural Semantics Example

- compute_com(While(b,c), m) =
  if compute_exp (b,m) = Bool(false)
  then m
  else compute_com
  (While(b,c), compute_com(c,m))

- May fail to terminate - exceed stack limits
- Returns no useful information then

Transition Semantics

- Form of operational semantics
- Describes how each program construct transforms machine state by *transitions*
- Rules look like
  (C, *m*) --> (C', *m'*) or (C,*m*) --> *m'*
- *C, C'* is code remaining to be executed
- m, m' represent the state/store/memory/environment
  - Partial mapping from identifiers to values
  - Sometimes *m* (or *C*) not needed
- Indicates exactly one step of computation

Expressions and Values

- *C, C'* used for commands; *E, E' for* expressions; *U,V* for values
- Special class of expressions designated as *values*
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
  - Other possibilities exist

Evaluation Semantics

- Transitions successfully stops when E/*C* is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final *meaning* of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence

Simple Imperative Programming Language

- *I* ∈ *Identifiers*
- *N* ∈ *Numerals*
- *B* ::= true | false | *B* & *B* | *B* or *B* | not *B* | *E* < *E* | *E* = *E*
- *E::= N | I | E + E | E * E | E - E | - E*
- C::= skip | *C;C* | *I* ::= *E*
  | if *B* then *C* else *C* fi | while *B* do *C* od

Transitions for Expressions

- Numerals are values

- Boolean values = {true, false}

- Identifiers: (*I,m*) --> (*m*(*I*), *m*)

Boolean Operations:

- Operators: (short-circuit)

$(\text{false} \;\&\; B,\ m) \longrightarrow (\text{false}, m)$

$(\text{true} \;\&\; B,\ m) \longrightarrow (B, m)$

$$\frac{(B,\ m) \longrightarrow (B'',\ m)}{(B \;\&\; B',\ m) \longrightarrow (B'' \;\&\; B',\ m)}$$

$(\text{true or } B,\ m) \longrightarrow (\text{true}, m)$

$(\text{false or } B,\ m) \longrightarrow (B, m)$

$$\frac{(B,\ m) \longrightarrow (B'',\ m)}{(B \text{ or } B',\ m) \longrightarrow (B'' \text{ or } B', m)}$$

$(\text{not true},\ m) \longrightarrow (\text{false}, m)$

$(\text{not false},\ m) \longrightarrow (\text{true}, m)$

$$\frac{(B,\ m) \longrightarrow (B',\ m)}{(\text{not } B,\ m) \longrightarrow (\text{not } B',\ m)}$$

Relations

$$\frac{(E,\ m) \longrightarrow (E'',m)}{(E \sim E',\ m) \longrightarrow (E'' \sim E',m)}$$

$$\frac{(E,\ m) \longrightarrow (E',m)}{(V \sim E,\ m) \longrightarrow (V \sim E',m)}$$

$(U \sim V,\ m) \longrightarrow (\text{true}, m)$ or (false, m) depending on whether $U \sim V$ holds or not

Arithmetic Expressions

$$\frac{(E,\ m) \longrightarrow (E'',m)}{(E \text{ op } E',\ m) \longrightarrow (E'' \text{ op } E',m)}$$

$$\frac{(E,\ m) \longrightarrow (E',m)}{(V \text{ op } E,\ m) \longrightarrow (V \text{ op } E',m)}$$

$(U \text{ op } V,\ m) \longrightarrow (N,m)$ where N is the specified value for $U \text{ op } V$

Commands - in English

- skip means done evaluating
- When evaluating an assignment, evaluate the expression first
- If the expression being assigned is already a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory

Commands

$(\text{skip},\ m) \longrightarrow m$

$$\frac{(E,m) \longrightarrow (E',m)}{(I::=E,m) \longrightarrow (I::=E',m)}$$

$(I::=V,m) \longrightarrow m[I \leftarrow\!\!- V]$

$$\frac{(C,m) \longrightarrow (C'',m')}{(C;C',\ m) \longrightarrow (C'';C',m')} \qquad \frac{(C,m) \longrightarrow m'}{(C;C',\ m) \longrightarrow (C',m')}$$

If Then Else Command - in English

- If the boolean guard in an if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

If Then Else Command

(if true then *C* else *C'* fi, *m*) --> (*C*, *m*)

(if false then *C* else *C'* fi, *m*) --> (*C'*, *m*)

$$\frac{(B,m) \dashrightarrow (B',m)}{\text{(if } B \text{ then } C \text{ else } C' \text{ fi}, m) \dashrightarrow \text{(if } B' \text{ then } C \text{ else } C' \text{ fi}, m)}$$

What should while transition to?

(while B do C od, m) → ?

Wrong! BAD

(B, m) → (B', m)

(while B do C od, m) -→ (while B' do C od, m)

While Command

(while *B* do *C* od, *m*) -->
(if *B* then *C*; while *B* do *C* od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.

Example Evaluation

- First step:

____

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x > 5, {x -> 7}) --> ?

____

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x,{x -> 7}) --> (7, {x -> 7})

(x > 5, {x -> 7}) --> ?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x,{x -> 7}) --> (7, {x -> 7})

(x > 5, {x -> 7}) --> (7 > 5, {x -> 7})

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x,{x -> 7}) --> (7, {x -> 7})

(x > 5, {x -> 7}) --> (7 > 5, {x -> 7})

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

Example Evaluation

- Second Step:

(7 > 5, {x -> 7}) --> (true, {x -> 7})

(if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> (if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

- Third Step:

(if true then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})
-->(y:=2+3, {x->7})

Example Evaluation

- Fourth Step:

(2+3, {x-> 7}) --> (5, {x -> 7})

(y:=2+3, {x->7}) --> (y:=5, {x->7})

- Fifth Step:

(y:=5, {x->7}) --> {y -> 5, x -> 7}

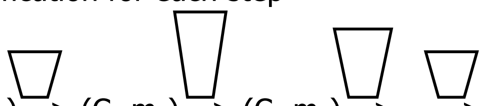
Example Evaluation

- Bottom Line:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})
-->(if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})
-->(y:=2+3, {x->7})
--> (y:=5, {x->7}) --> {y -> 5, x -> 7}

Transition Semantics Evaluation

- A sequence of steps with trees of justification for each step

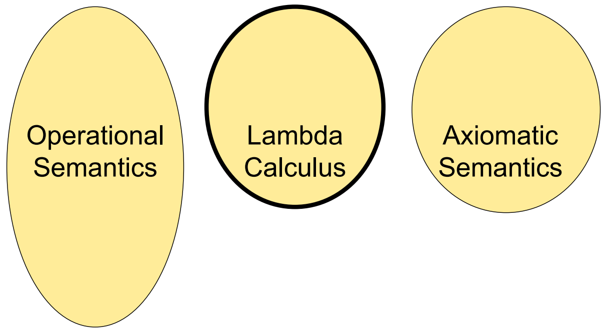


(C_1,m_1) --> (C_2,m_2) --> (C_3,m_3) --> … --> m

- Let -->* be the transitive closure of -->
- Ie, the smallest transitive relation containing -->

Programming Languages & Compilers

III : Language Semantics



Lambda Calculus - Motivation

- Aim is to capture the essence of functions, function applications, and evaluation
- λ–calculus is a theory of computation
- "The Lambda Calculus: Its Syntax and Semantics". H. P. Barendregt. North Holland, 1984

Lambda Calculus - Motivation

- All *sequential programs* may be viewed as functions from input (initial state and input values) to output (resulting state and output values).
- λ-calculus is a mathematical formalism of functions and functional computations
- Two flavors: typed and untyped

Untyped λ-Calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e

    (Function creation, think fun x -> e)
  - Application: e_1 e_2
  - Parenthesized expression: (e)

Untyped λ-Calculus Grammar

- Formal BNF Grammar:
  - <expression> ::= <variable>
| <abstraction>
| <application>
| (<expression>)
  - <abstraction>
::= λ<variable>.<expression>
  - <application>
::= <expression> <expression>

Untyped λ-Calculus Terminology

- Occurrence: a location of a subterm in a term
- Variable binding: λ x. e is a binding of x in e
- Bound occurrence: all occurrences of x in λ x. e
- Free occurrence: one that is not bound
- Scope of binding: in λ x. e, all occurrences in e not in a subterm of the form λ x. e' (same x)
- Free variables: all variables having free occurrences in a term

Example

- Label occurrences and scope:

(λ x. y λ y. y (λ x. x y) x) x
1 2 3 4 5 6 7 8 9

Example

- Label occurrences and scope:
- free free
(λ x. y λ y. y (λ x. x y) x) x
1 2 3 4 5 6 7 8 9

Untyped λ-Calculus

- How do you compute with the λ-calculus?
- Roughly speaking, by substitution:
- $(\lambda\ x.\ e_1)\ e_2 \Rightarrow^* e_1\ [e_2 / x]$
- * Modulo all kinds of subtleties to avoid free variable capture

Transition Semantics for λ-Calculus

$$\frac{E \dashrightarrow E''}{E\ E' \dashrightarrow E''\ E'}$$

- Application (version 1 - Lazy Evaluation)
  $(\lambda\ x\ .\ E)\ E' \dashrightarrow E[E'/x]$
- Application (version 2 - Eager Evaluation)

$$\frac{E' \dashrightarrow E''}{(\lambda\ x\ .\ E)\ E' \dashrightarrow (\lambda\ x\ .\ E)\ E''}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \dashrightarrow E[V/x]}$$

V - variable or abstraction (value)

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is Turing Complete
  - Can express any sequential computation
- Problems:
  - How to express basic data: booleans, integers, etc?
  - How to express recursion?
  - Constants, if_then_else, etc, are conveniences; can be added as syntactic sugar

Typed vs Untyped λ-Calculus

- The *pure* λ-calculus has no notion of type: (f f) is a legal expression
- Types restrict which applications are valid
- Types are not syntactic sugar! They disallow some terms
- Simply typed λ-calculus is less powerful than the untyped λ-Calculus: NOT Turing Complete (no recursion)

Uses of λ-Calculus

- Typed and untyped λ-calculus used for theoretical study of sequential programming languages
- Sequential programming languages are essentially the λ-calculus, extended with predefined constructs, constants, types, and syntactic sugar
- Ocaml is close to the λ-Calculus:

  fun x -> exp --> λ x. exp

  let x = e_1 in e_2 --> (λ x. e_2)e_1

α Conversion

1. α-conversion:
2. λ x. exp --α--> λ y. (exp [y/x])
3. Provided that
   1. y is not free in exp
   2. No free occurrence of x in exp becomes bound in exp when replaced by y

λ x. x (λ y. x y) - × -> λ y. y(λ y.y y)

α Conversion Non-Examples

1. Error: y is not free in term second

   λ x. x y ~~--α-->~~ λ y. y y

2. Error: free occurrence of x becomes bound in wrong way when replaced by y

   λ x. λ y. x y ~~--α-->~~ λ y. λ y. y y

   (λ y. x y is exp; λ y. y y is exp[y/x])

But λ x. (λ y. y) x --α--> λ y. (λ y. y) y

And λ y. (λ y. y) y --α--> λ x. (λ y. y) x

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if
- it is an equivalence relation
- If e_1 ~ e_2 then
  - (e e_1) ~ (e e_2) and (e_1e) ~ (e_2 e)
  - λ x. e_1 ~ λ x. e_2

α Equivalence

- α equivalence is the smallest congruence containing α conversion
- One usually treats α-equivalent terms as equal - i.e. use α equivalence classes of terms

Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

- λ x. (λ y. y x) x --α--> λ z. (λ y. y z) z so
  λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z
- (λ y. y z) --α--> (λ x. x z) so
  (λ y. y z) ~α~ (λ x. x z) so
  (λ y. y z) z ~α~ (λ x. x z) z so
  λ z. (λ y. y z) z ~α~ λ z. (λ x. x z) z
- λ z. (λ x. x z) z --α--> λ y. (λ x. x y) y so
  λ z. (λ x. x z) z ~α~ λ y. (λ x. x y) y
- λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y