A Graph-Rewriting Perspective of the Beta-Law

work in progress

Dan R. Ghica
Todd Waugh Ambridge
(University of Birmingham)

Koko Muroya
(University of Birmingham
& RIMS, Kyoto University)

LOLA 2018 (Oxford), 7 July 2018

Call-by-value beta-law

$$(\lambda x. t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \dots$

values $v ::= x \mid \lambda x.t \mid \dots$

golden standard of (functional) program equivalence and compiler optimisation

“A function can be applied to a value before evaluation without changing the outcome”

Call-by-value beta-law

$$(\lambda x.\, t)\; v \;=\; t\,[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \ldots$

values $v ::= x \mid \lambda x.t \mid \ldots$

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

Call-by-value beta-law

$$(\lambda x.t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid n \mid \mathrm{succ}(n) \mid \ldots$

values $v ::= x \mid \lambda x.t \mid n \mid \ldots$

basic operations (nat, int, float, ...)

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

Call-by-value beta-law

$$(\lambda x.t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \langle t,t\rangle \mid \mathrm{fst}(t) \mid \mathrm{snd}(t) \mid \ldots$

values $v ::= x \mid \lambda x.t \mid \langle v,v\rangle \mid$

algebraic
data structures

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

Call-by-value beta-law

$$(\lambda x.t)\,v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \mu x.t$

values $v ::= x \mid \lambda x.t \mid \ldots$

recursion

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

Call-by-value beta-law

$$(\lambda x.t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \text{if } t \text{ then } t \text{ else } t$

values $v ::= x \mid \lambda x.t \mid \ldots$

conditional statement

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

Call-by-value beta-law

$$(\lambda x.\, t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid op(t, \ldots, t) \mid \ldots$

values $v ::= x \mid \lambda x.t \mid \ldots$

algebraic effects & handlers

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

Call-by-value beta-law

$$(\lambda x.t)\ v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \mathrm{callcc}(t) \mid \dots$

values $v ::= x \mid \lambda x.t \mid \dots$

control operators

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

Call-by-value beta-law

$$(\lambda x.\, t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \ldots$

values $v ::= x \mid \lambda x.t \mid \ldots$

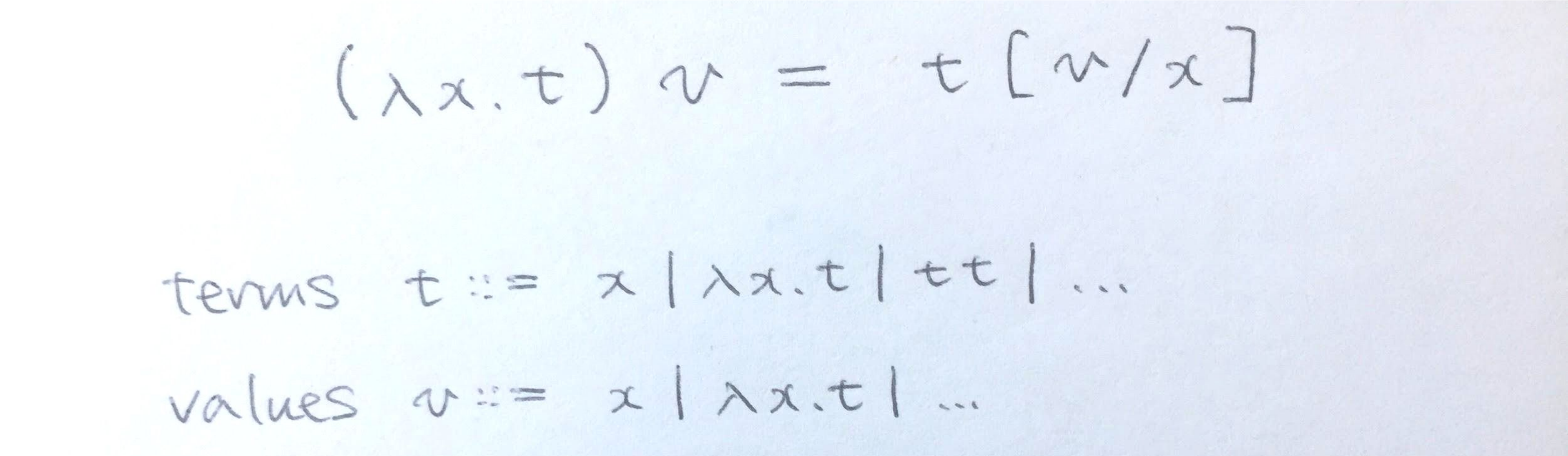
golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

justification by (operational) semantics, but how?

Muroya (U. B’ham. & RIMS, Kyoto U.)

Call-by-value beta-law



golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions

justification by (operational) semantics, but how?

$$(\lambda x.t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \ldots$

values $v ::= x \mid \lambda x.t \mid \ldots$

Question

Given an extension of untyped λ-calculus,

what semantic property of the extension

validates the call-by-value beta-law?

$$(\lambda x.t)\, v = t[v/x]$$

terms $t ::= x \mid \lambda x.t \mid t\,t \mid \dots$

values $v ::= x \mid \lambda x.t \mid \dots$

Question

Given an extension of untyped λ-calculus,

what *operational*-semantic property of the extension

validates the call-by-value beta-law?

Question

Given an extension of untyped λ-calculus,

what *operational*-semantic property of the extension

validates the call-by-value beta-law?

Answer?

Question

Given an extension of untyped λ-calculus,

what *operational*-semantic property of the extension

validates the call-by-value beta-law?

Answer?

A formal answer is yet to be stated…

But a graph-rewriting perspective provides:

- a useful & robust method
- key observations

Methodology

$$t ::= x \mid \lambda x.t \mid t\,t \mid k \mid$$
$$v ::= x \mid \lambda x.t \mid k \mid$$

Given an operational semantics of an extended λ-calculus:

closed term $\qquad t \Downarrow k \qquad$ basic constant

define the contextual equivalence by:

$$t \simeq t' \overset{\Delta}{\iff} {}^{\forall}C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed,}$$
$$C[t] \Downarrow k \iff C[t'] \Downarrow k'$$
$$\text{Moreover, } k = k'$$

prove the beta-law:

$$(\lambda x.t)\,v \simeq t[v/x]$$

and **observe** *some sufficient condition*.

Methodology

$t ::= x \mid \lambda x.t \mid t\,t \mid k \mid \ldots$

$v ::= x \mid \lambda x.t \mid k \mid \ldots$

Given an operational semantics of an extended λ-calculus:

closed term

$$t \Downarrow k$$

basic constant

define the contextual equivalence by:

$$t \simeq t' \overset{\Delta}{\Longleftrightarrow} \forall C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed,}$$

$$C[t] \Downarrow k \Longleftrightarrow C[t'] \Downarrow k'$$

$$\text{Moreover, } k = k'$$

prove the beta-law:

$$(\lambda x.t)\,v \simeq t[v/x]$$

and **observe** *some sufficient condition*.

Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step reduction

$$t \Downarrow k \overset{\triangle}{\Longleftrightarrow} t \to^* k$$

Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step reduction

$$t \Downarrow k \overset{\triangle}{\Longleftrightarrow} t \to^{*} k$$

… obscures a sub-term of interest :-(

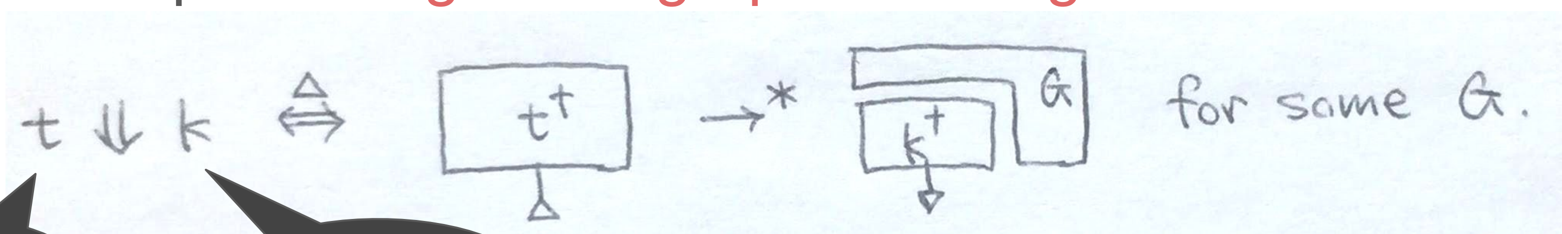
$$C[t] \equiv E_0[r_0] \to u_1 \equiv E_1[r_1] \to u_2 \equiv \cdots$$

redex searching
(i.e. decomposition into evaluation context & redex)
obscures \`t\`

Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step "token-guided" graph-rewriting



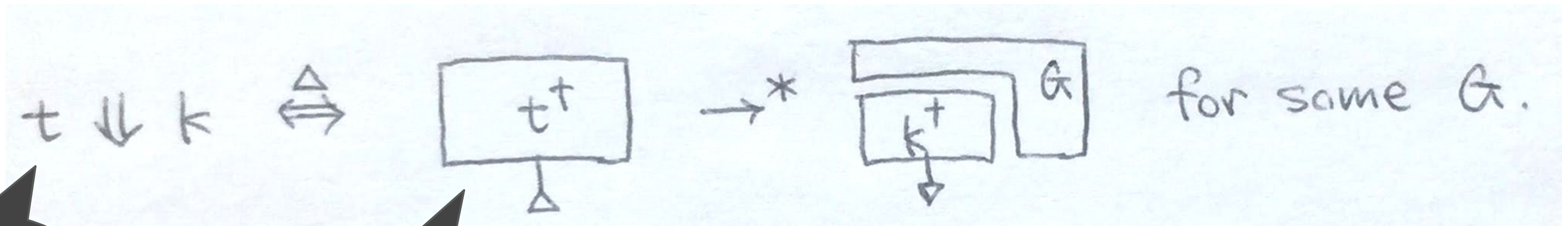
closed term

basic constant

Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step “token-guided” graph-rewriting



closed term

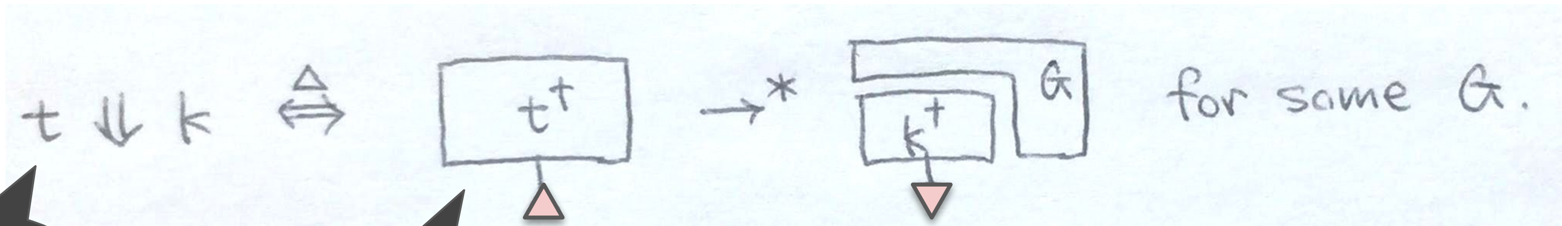
graph representation with unique open edge

Muroya (U. B’ham. & RIMS, Kyoto U.)

Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step “token-guided” graph-rewriting



closed term

graph representation with unique open edge

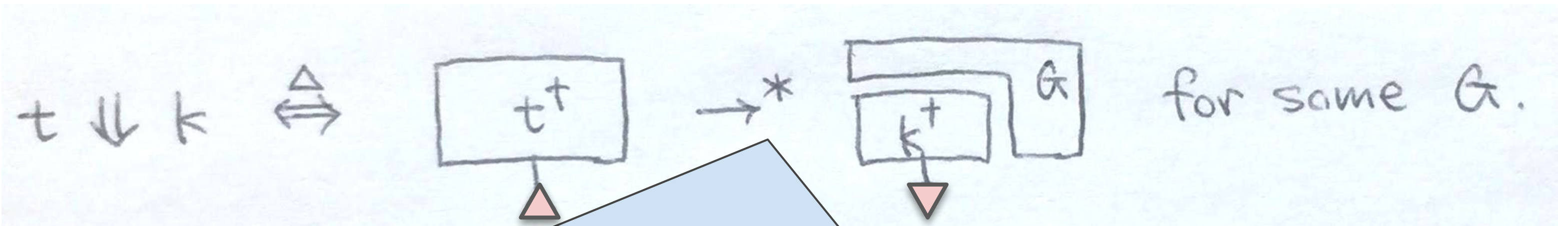
distinguished edge/node as “token”

Muroya (U. B’ham. & RIMS, Kyoto U.)

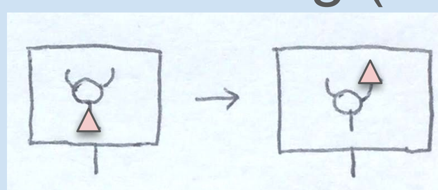
Which operational semantics?

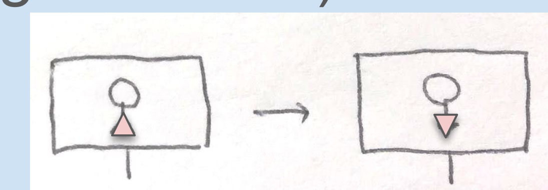
- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step “token-guided” graph-rewriting

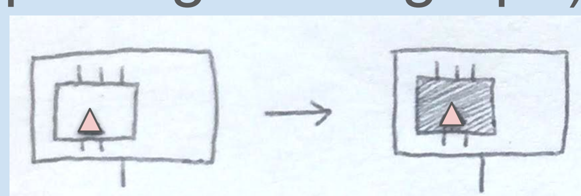


- redex searching (moving the token)





- rewriting (replacing a sub-graph)

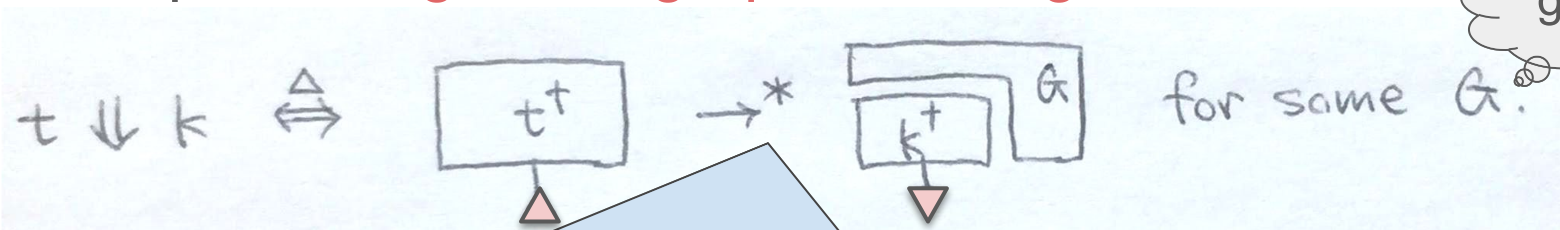


Muroya (U. B’ham. & RIMS, Kyoto U.)

Which operational semantics?

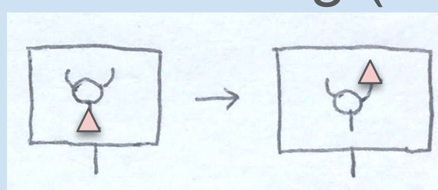
- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

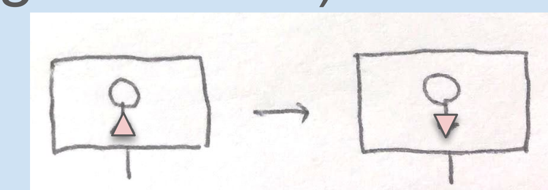
small-step "token-guided" graph-rewriting



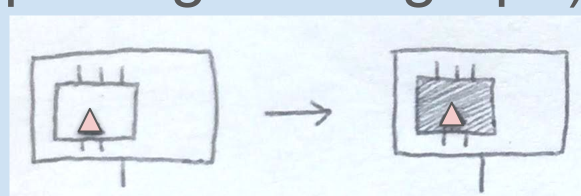
garbage

- redex searching (moving the token)





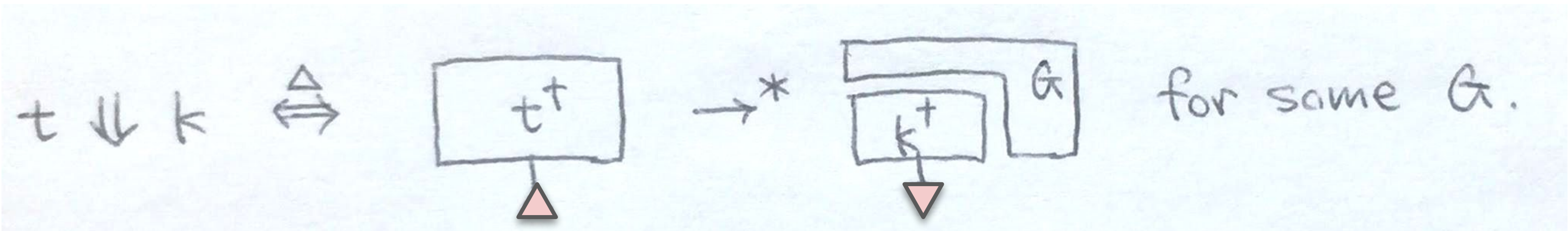
- rewriting (replacing a sub-graph)



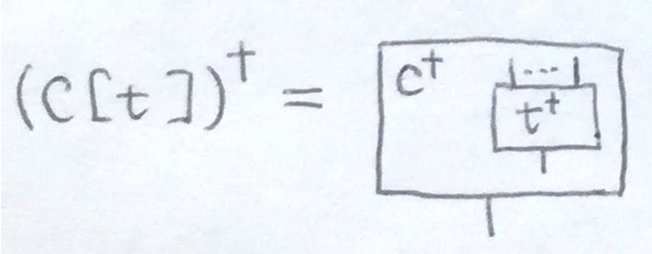
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step “token-guided” graph-rewriting



… keeps a sub-term of interest traceable :-)

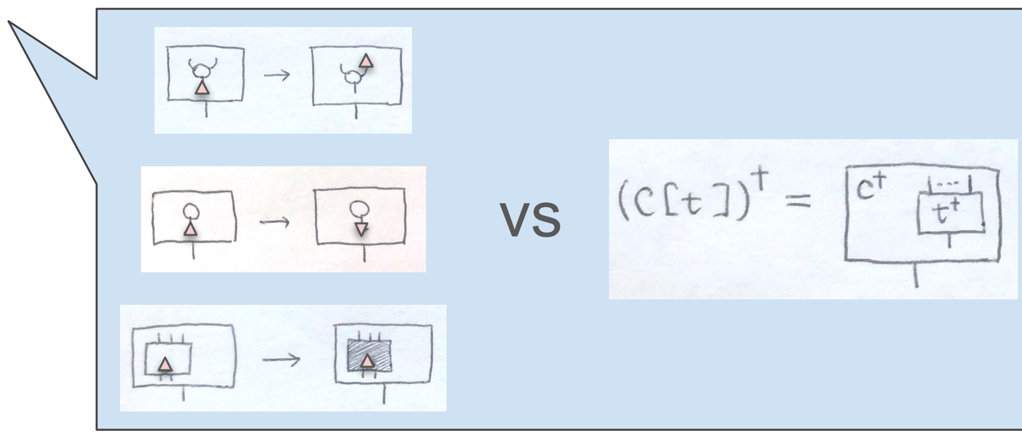


Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step "token-guided" graph-rewriting

- visible interaction between the token △ and a sub-graph
    - redex searching
    - rewriting



step-wise reasoning to prove a contextual equivalence

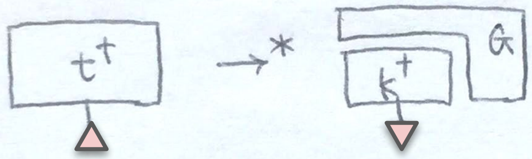
Methodology

$t ::= x \mid \lambda x.t \mid t\,t \mid k \mid$

$v ::= x \mid \lambda x.t \mid k \mid$

Given operational semantics of an extended λ-calculus:

closed term

$t \Downarrow k \;\overset{\Delta}{\Longleftrightarrow}\;$  for some G.

define the contextual equivalence by:

$t \simeq t' \;\overset{\Delta}{\Longleftrightarrow}\; {}^\forall C$ s.t. $C[t]$ and $C[t']$ are closed,

$C[t] \Downarrow k \iff C[t'] \Downarrow k'$

Moreover, $k = k'$

same basic constants

prove the beta-law:

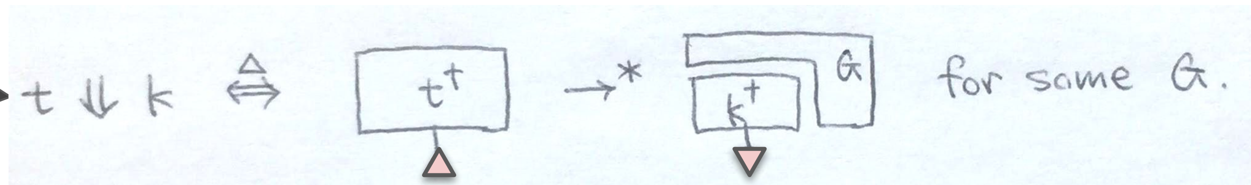
$$(\lambda x.t)\,v \simeq t[v/x]$$

and **observe** *some sufficient condition*.

Case study: linear λ-calculus + “linear” recursion

Given operational semantics:

closed term



define the contextual equivalence by:

$$t \simeq t' \overset{\triangle}{\Leftrightarrow} {}^{\forall}C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed,}$$

$$C[t] \Downarrow k \Leftrightarrow C[t'] \Downarrow k'$$

$$\text{Moreover, } k = k'$$

same basic constants

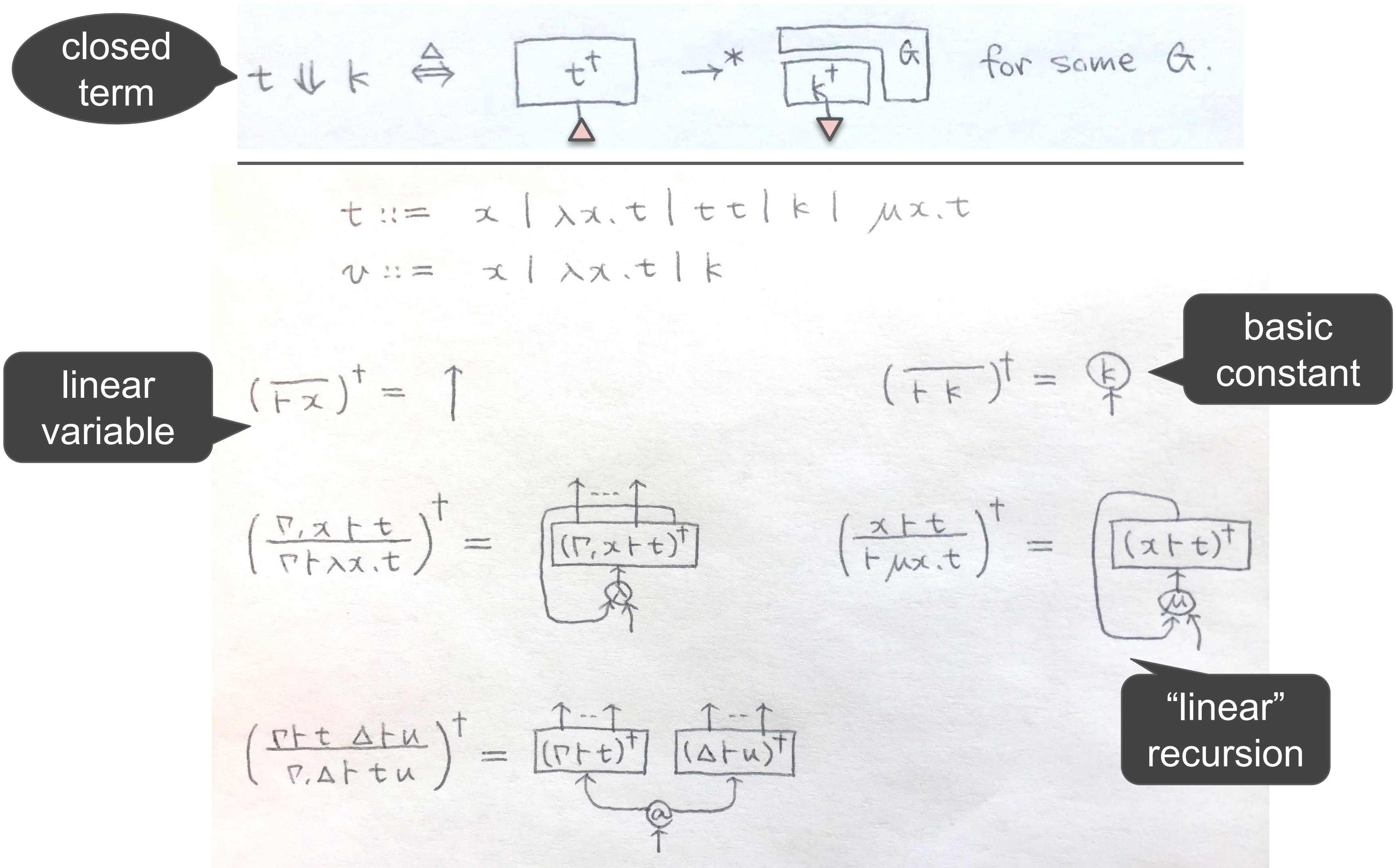
prove the beta-law:

$$(\lambda x.t)\, v \simeq t[v/x]$$

and **observe** *some sufficient condition*.

Case study: linear λ-calculus + “linear” recursion

Given operational semantics:



Muroya (U. B’ham. & RIMS, Kyoto U.)

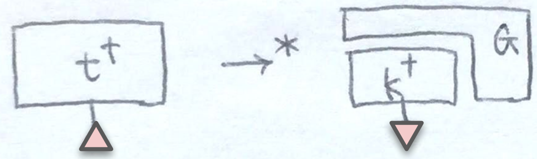
Case study: linear λ-calculus + "linear" recursion

$t ::= x \mid \lambda x.t \mid t\,t \mid k \mid \mu x.t$

$v ::= x \mid \lambda x.t \mid k$

Given operational semantics:

closed term

$t \Downarrow k \overset{\triangle}{\iff}$  for some G.

define the contextual equivalence by:

$t \simeq t' \overset{\triangle}{\iff} {}^\forall C$ s.t. $C[t]$ and $C[t']$ are closed,

$C[t] \Downarrow k \iff C[t'] \Downarrow k'$

Moreover, $k = k'$

same basic constants

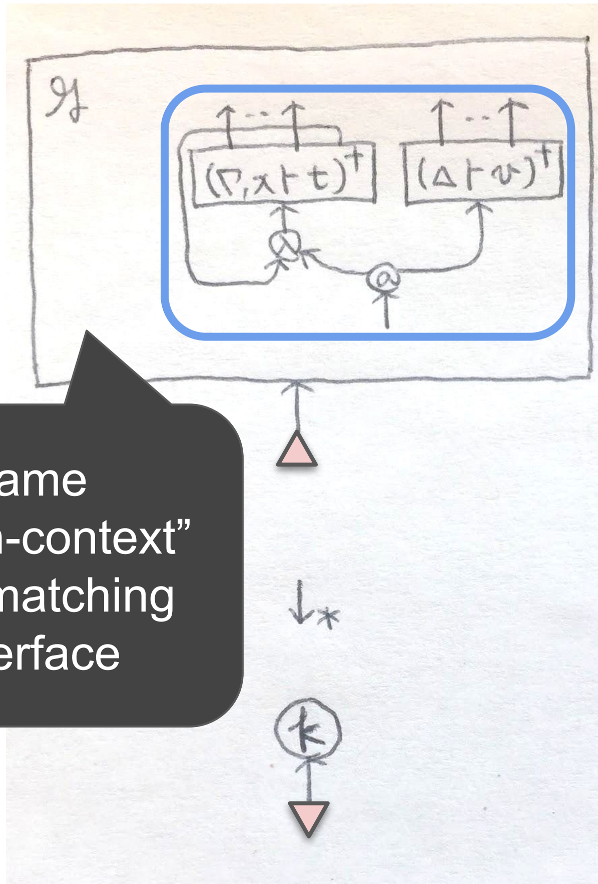
prove the beta-law:

$$(\lambda x.t)\,v \simeq t[v/x]$$

and **observe** *some sufficient condition*.

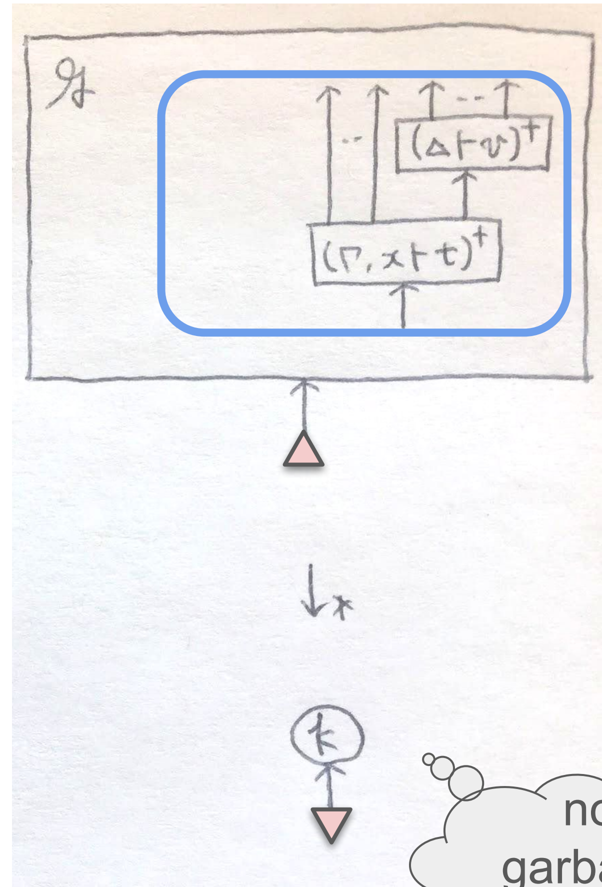
Case study: linear λ-calculus + "linear" recursion

… **prove** the beta-law:



same "graph-context" with matching interface

⟺

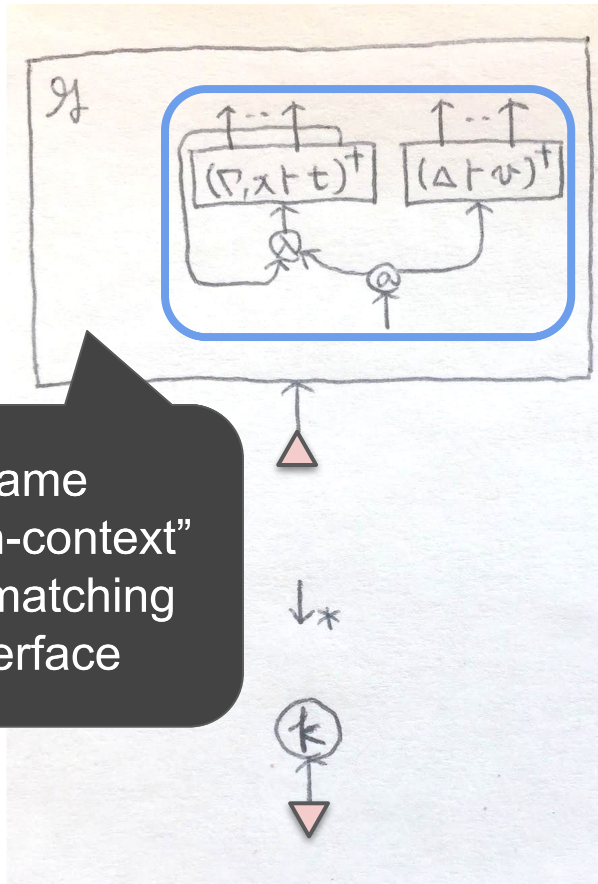


no garbage created

Muroya (U. B'ham. & RIMS, Kyoto U.)

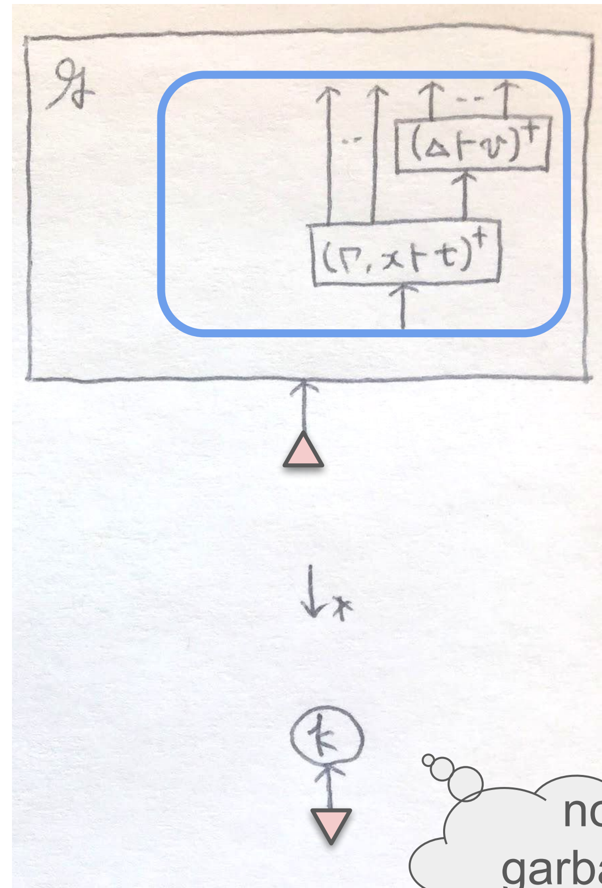
Case study: linear λ-calculus + “linear” recursion

… **prove** the beta-law *by step-wise reasoning*:



same “graph-context” with matching interface

⟺

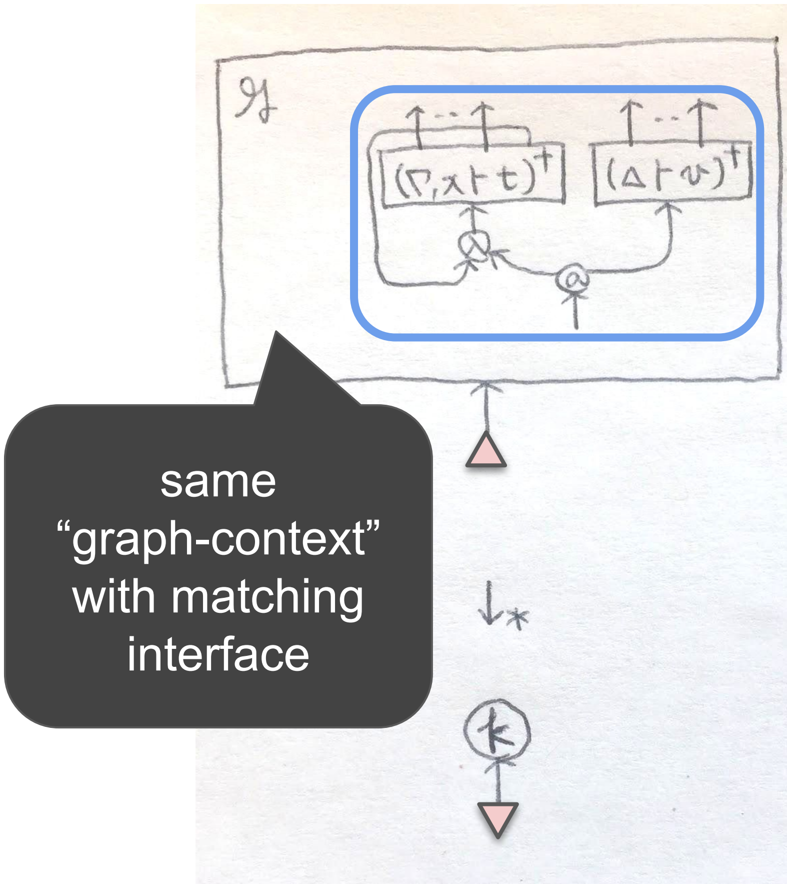


no garbage created

Muroya (U. B’ham. & RIMS, Kyoto U.)

Case study: linear λ-calculus + “linear” recursion

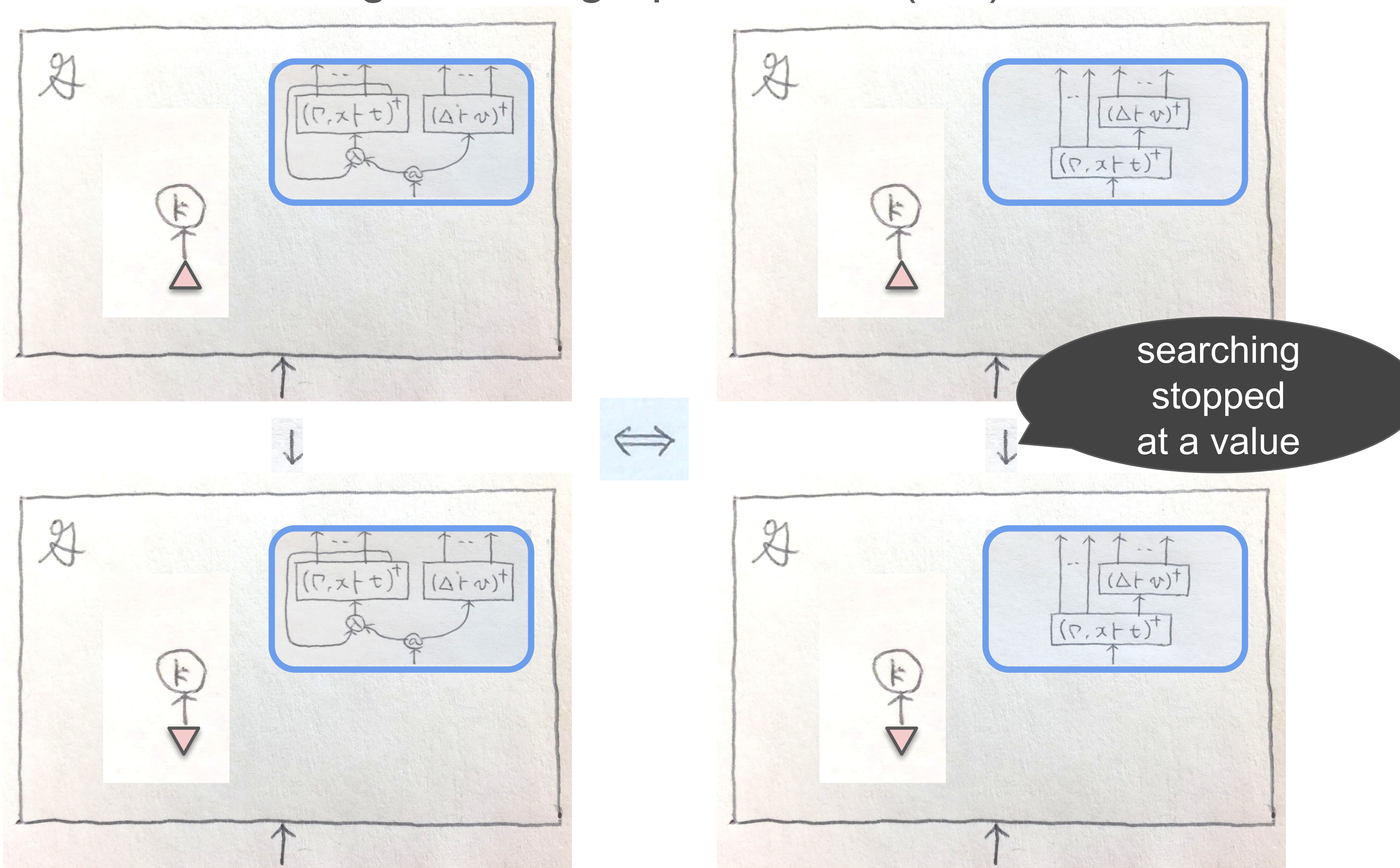
… **prove** the beta-law *by step-wise reasoning*:



1. redex searching “within” graph-context
2. rewriting “in” graph-context
3. visiting the hole

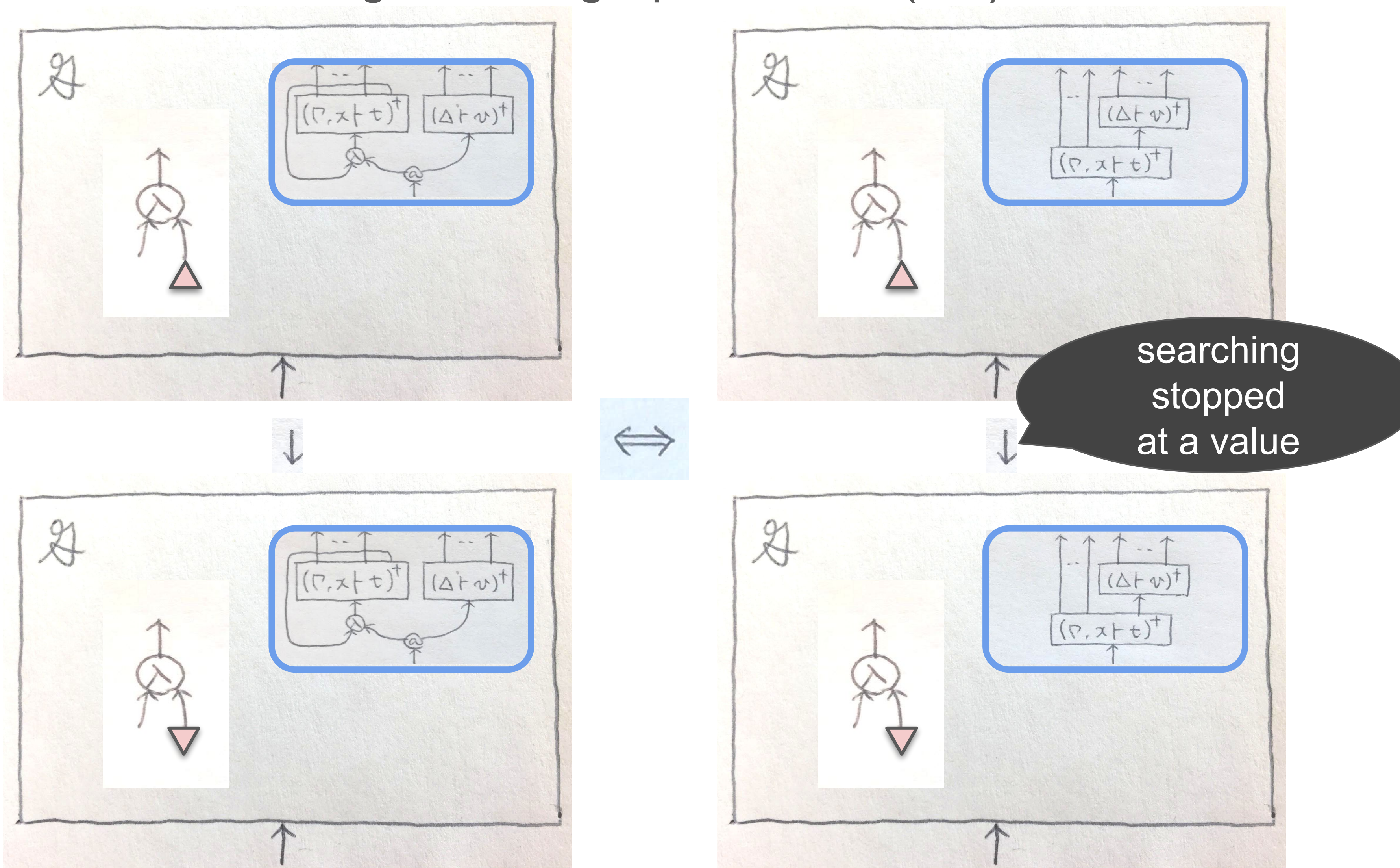
Case study: linear λ-calculus + “linear” recursion

1. redex searching “within” graph-context (1/6)



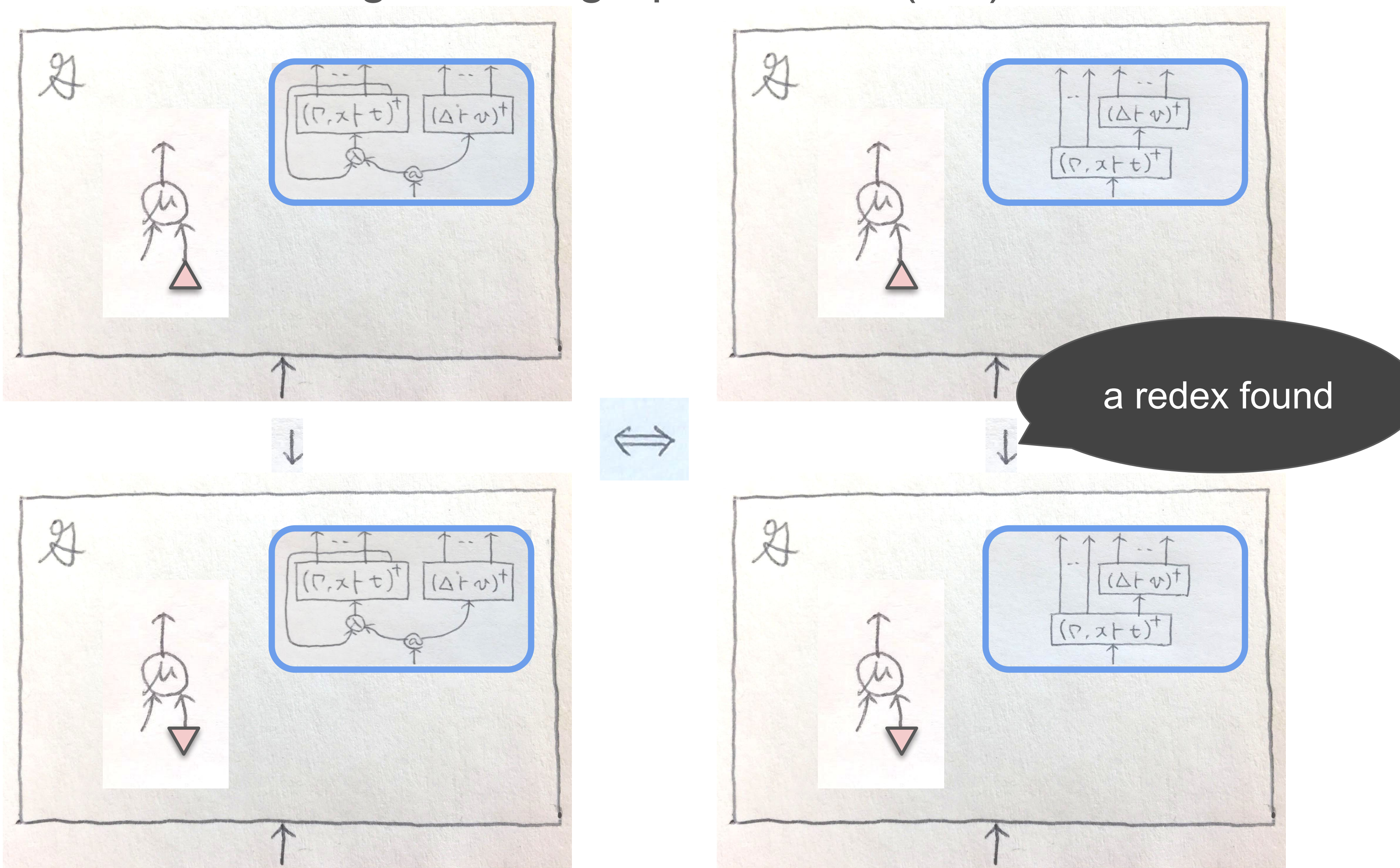
Case study: linear λ-calculus + “linear” recursion

1. redex searching “within” graph-context (2/6)



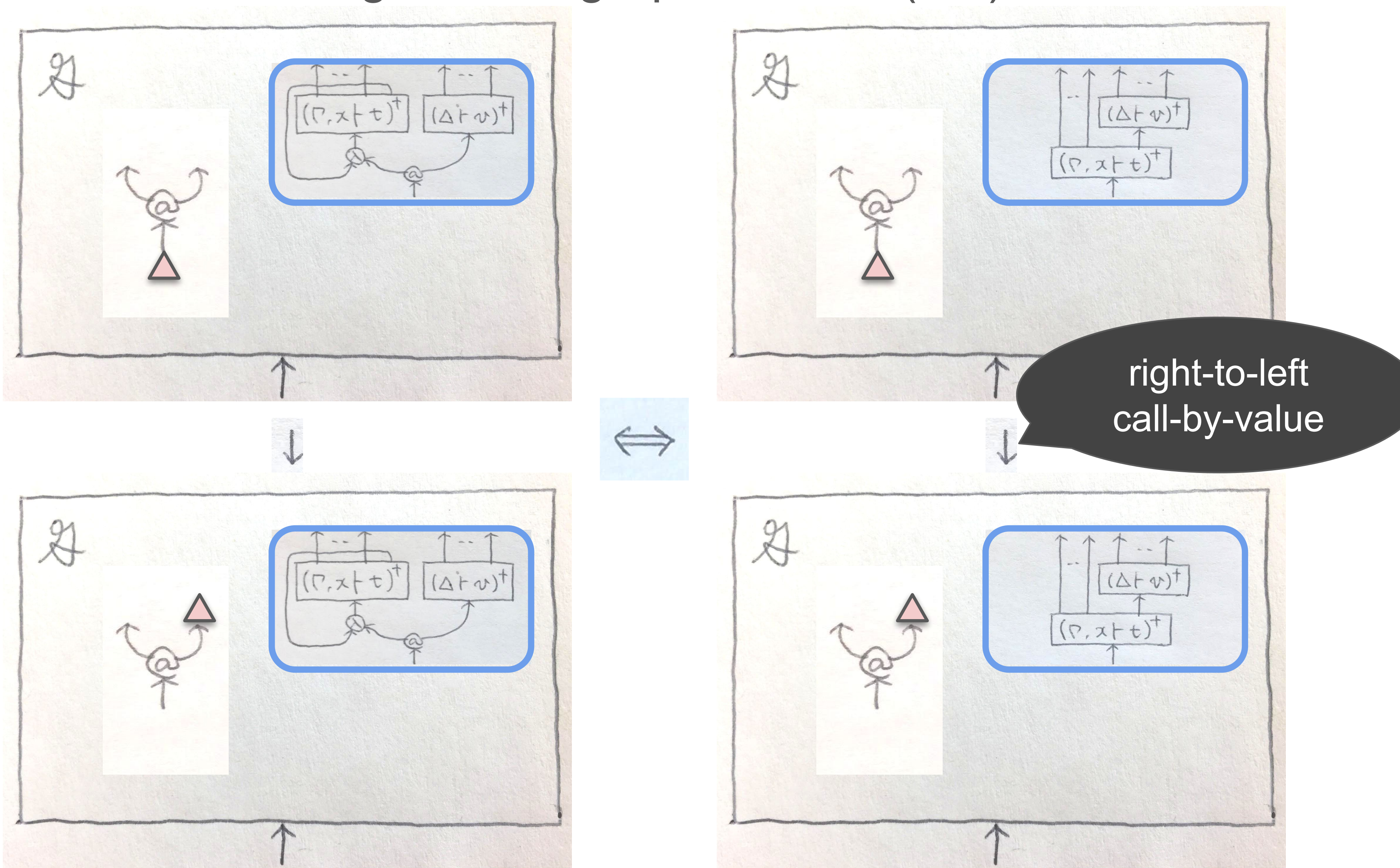
Case study: linear λ-calculus + “linear” recursion

1. redex searching “within” graph-context (3/6)



Case study: linear λ-calculus + “linear” recursion

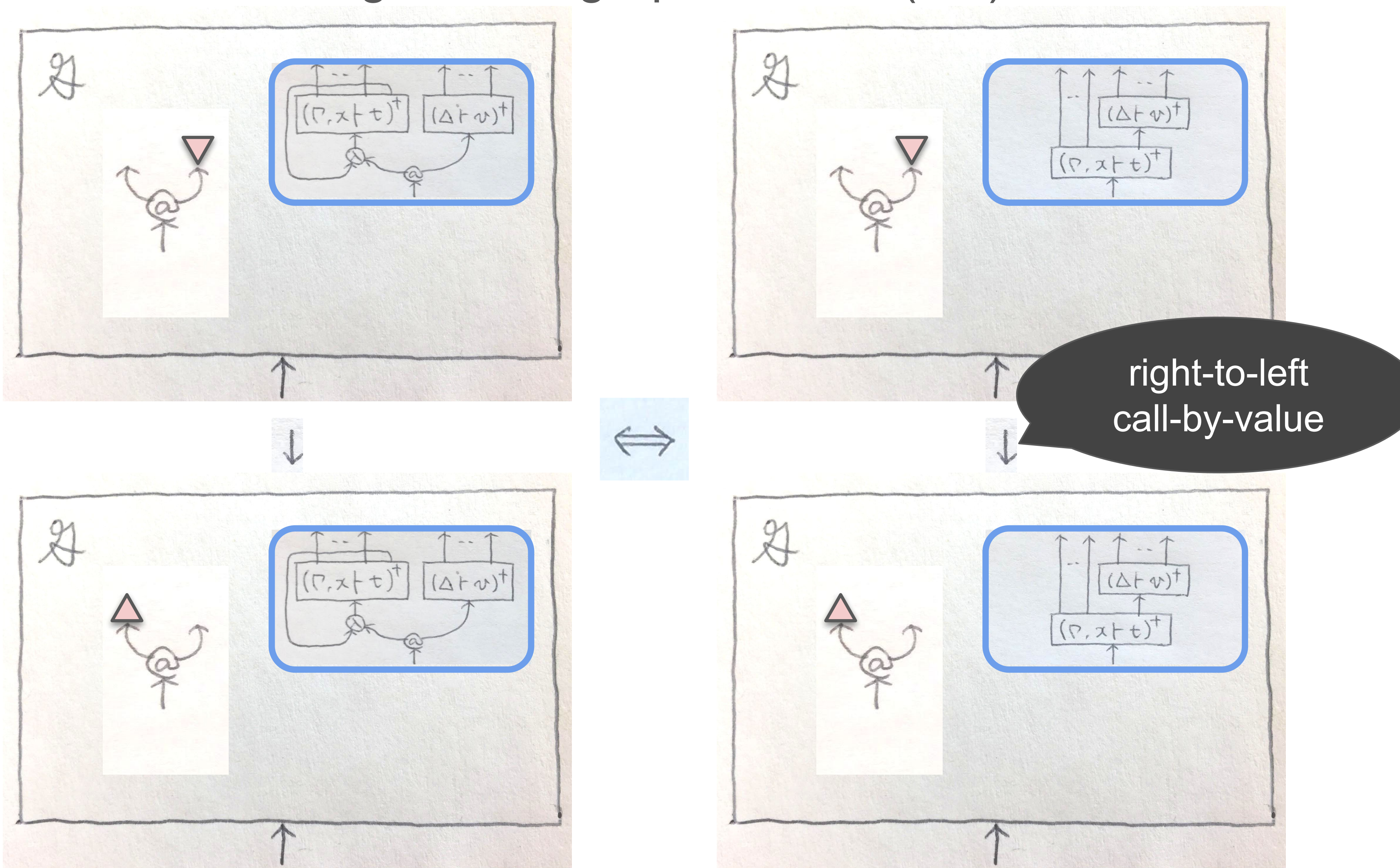
1. redex searching “within” graph-context (4/6)



Muroya (U. B’ham. & RIMS, Kyoto U.)

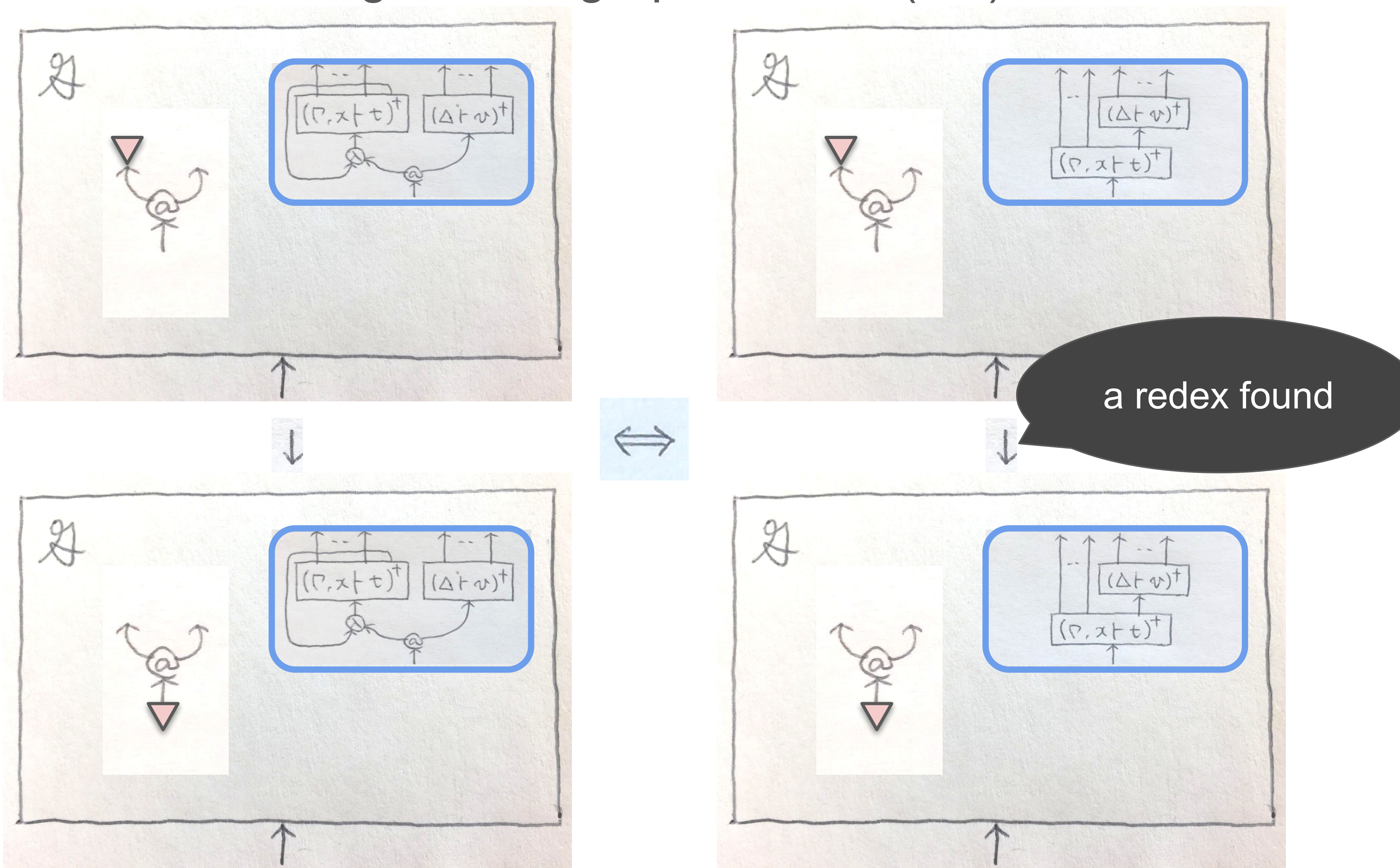
Case study: linear λ-calculus + “linear” recursion

1. redex searching “within” graph-context (5/6)



Case study: linear λ-calculus + “linear” recursion

1. redex searching “within” graph-context (6/6)



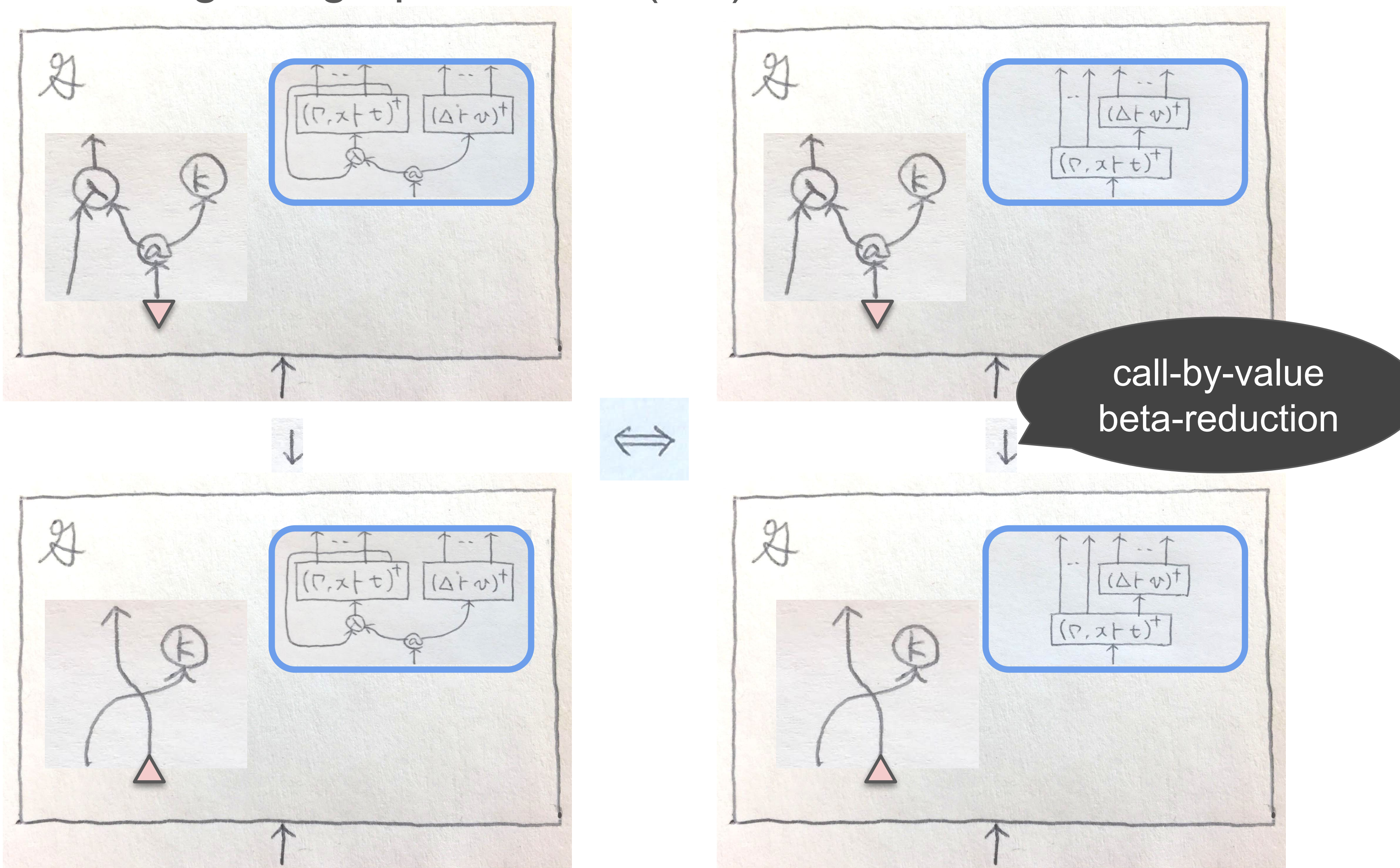
Case study: linear λ-calculus + “linear” recursion

1. redex searching within graph-context (6 cases)

observation: only one node is inspected at each step

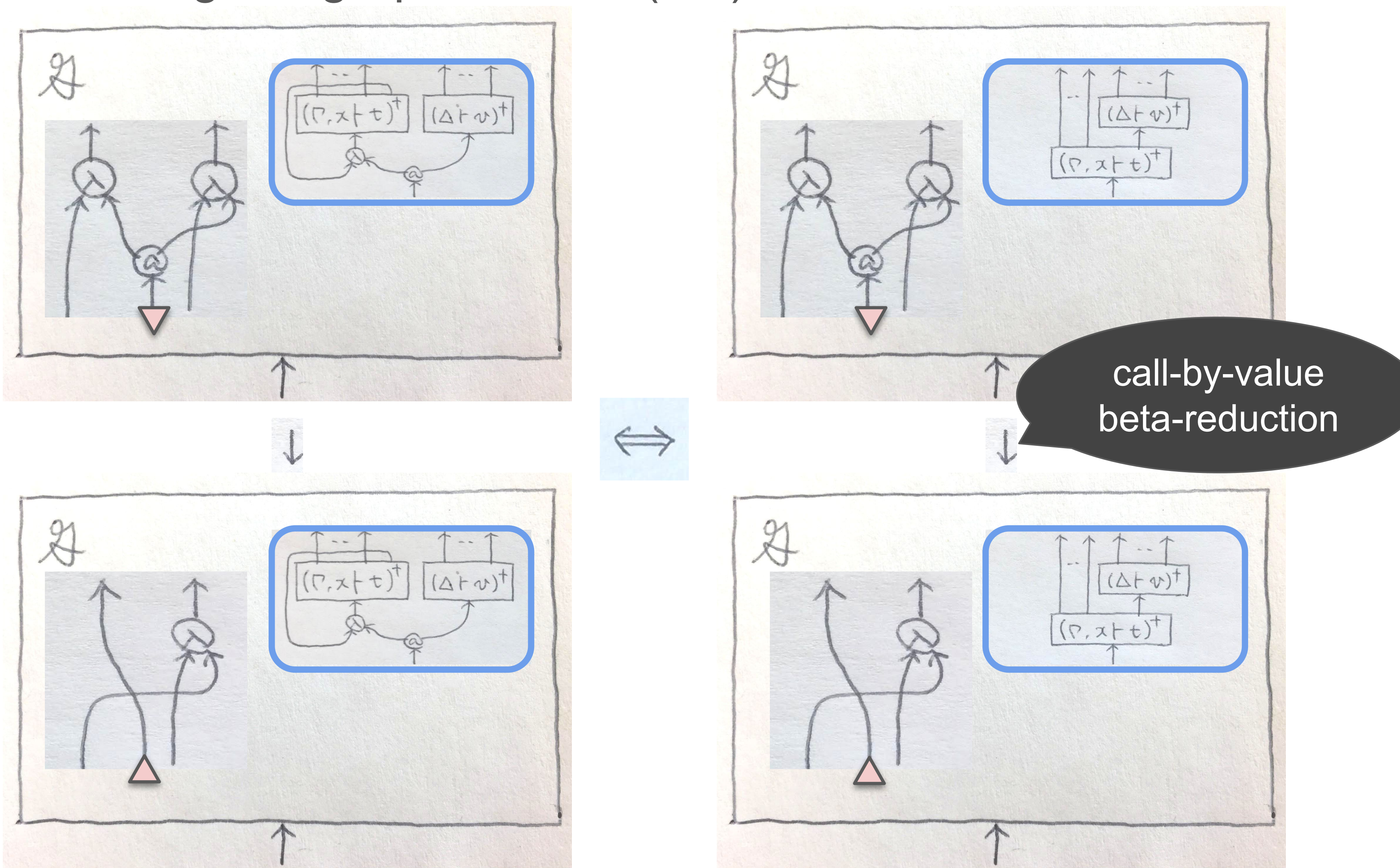
Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context (1/3)



Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context (2/3)



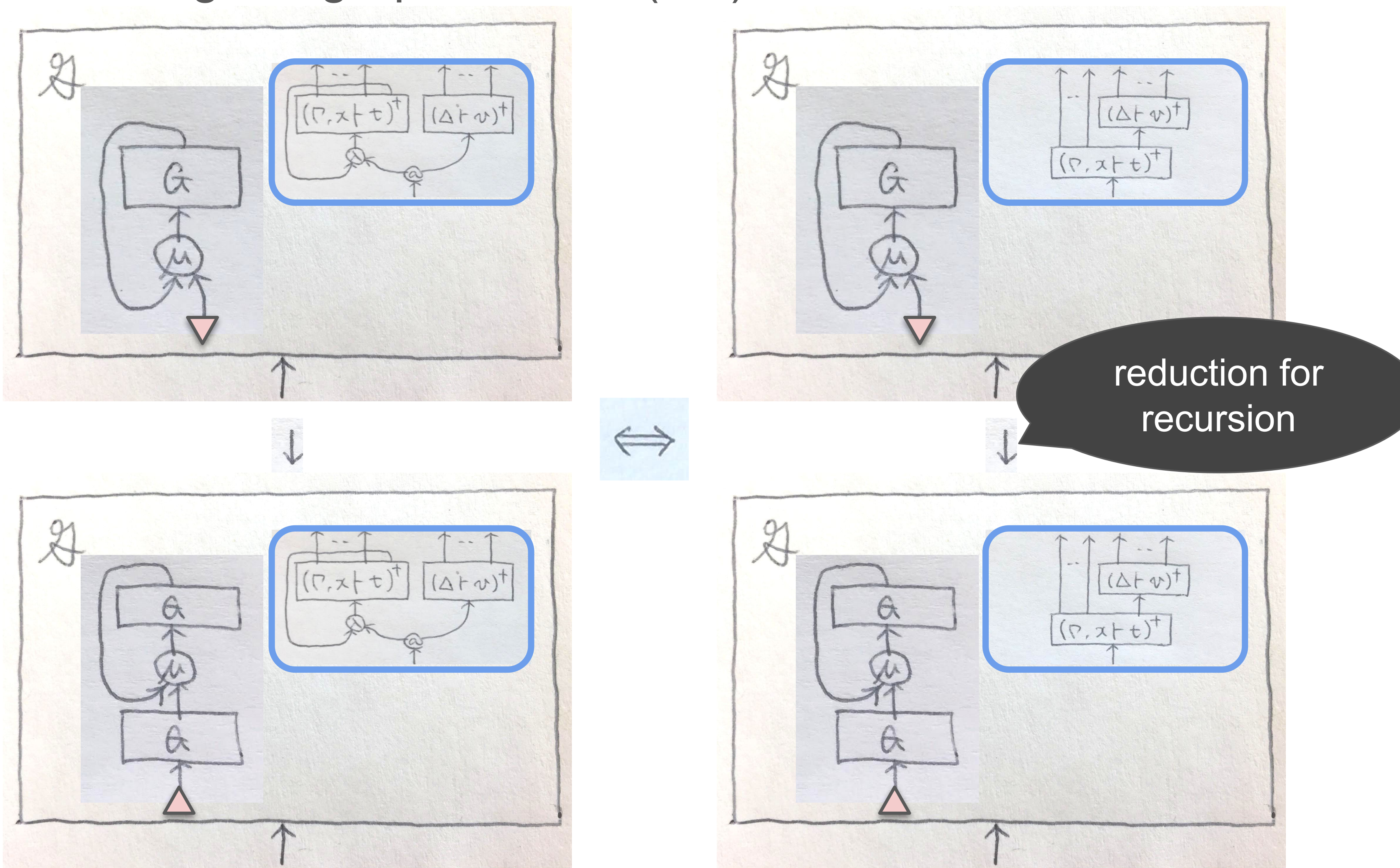
Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context

observation: the hole is not involved

Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context (3/3)



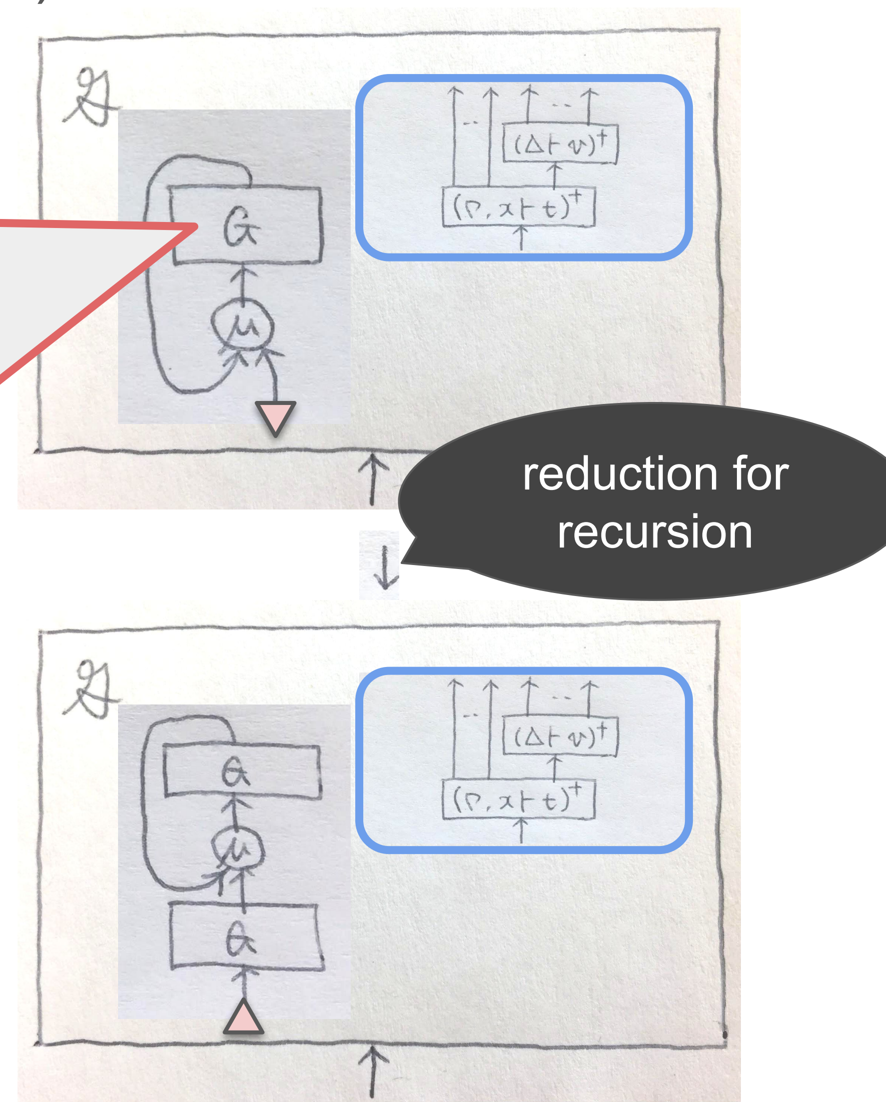
Muroya (U. B’ham. & RIMS, Kyoto U.)

Case study: linear λ-calculus + "linear" recursion

2. rewriting "in" graph-context (3/3)

\`G\` contains:

- all reachable nodes from \`μ\`
- *hence*,
    - none of the hole
    - or, all of the hole

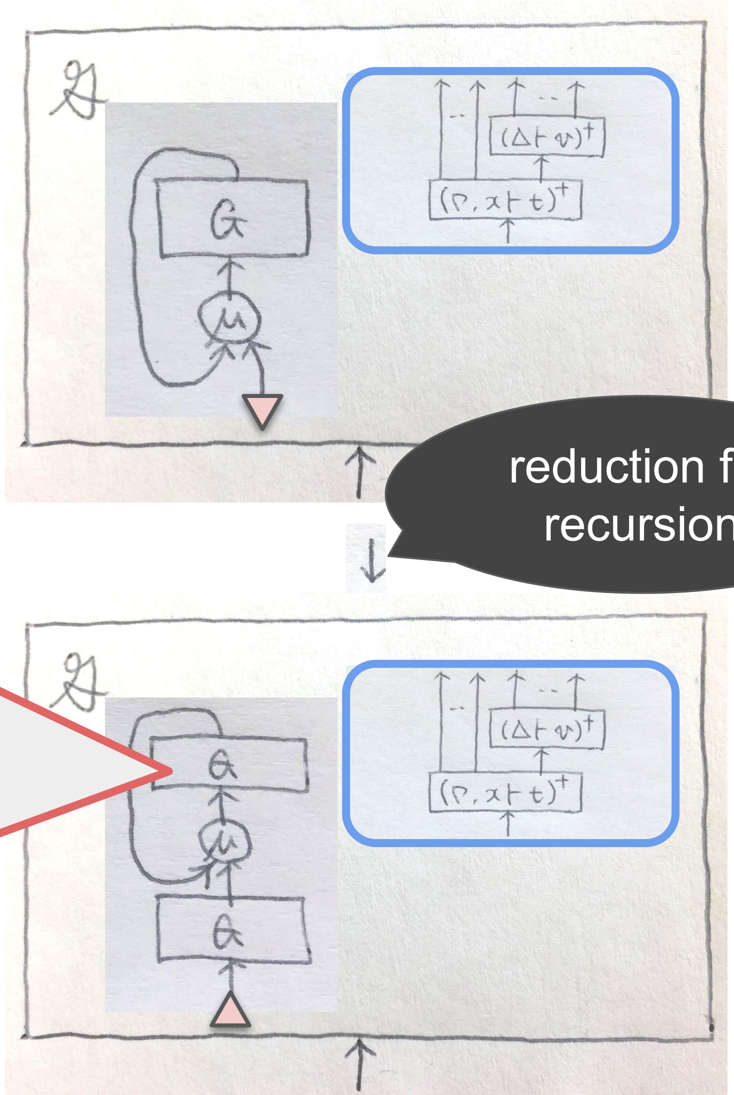


Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context (3/3)

the hole is

- not involved
- or, duplicated as a part of `G`



reduction for recursion

Case study: linear λ-calculus + “linear” recursion

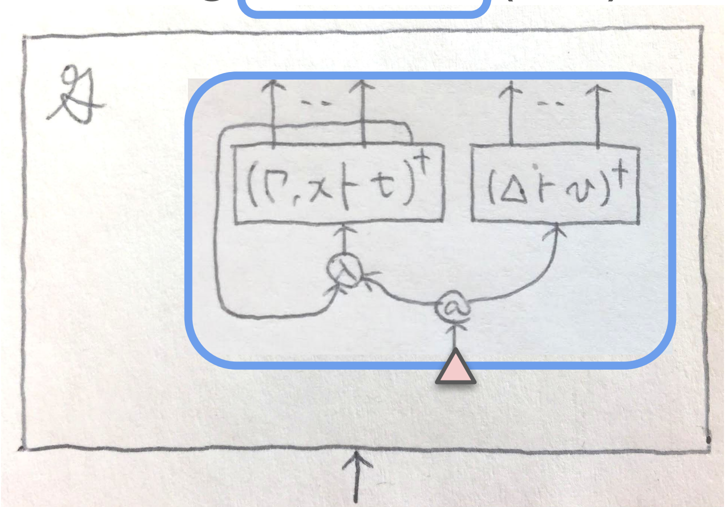
2. rewriting “in” graph-context

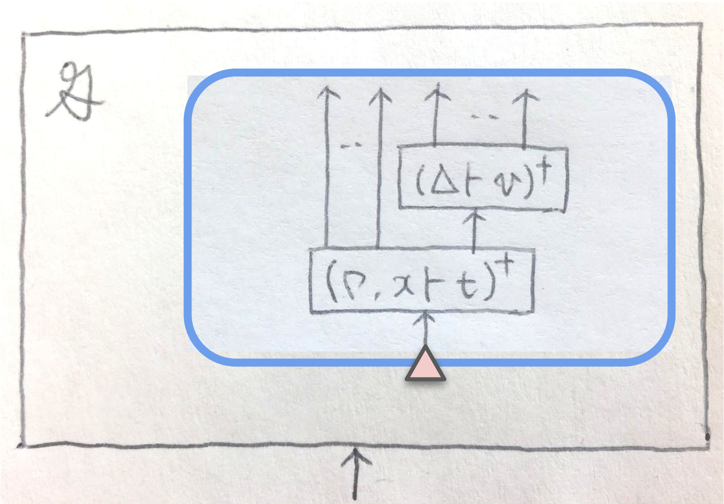
observation: the hole is not involved, or is duplicated *as a whole*

observation 2: each rewriting step is “history-free”

Case study: linear λ-calculus + "linear" recursion

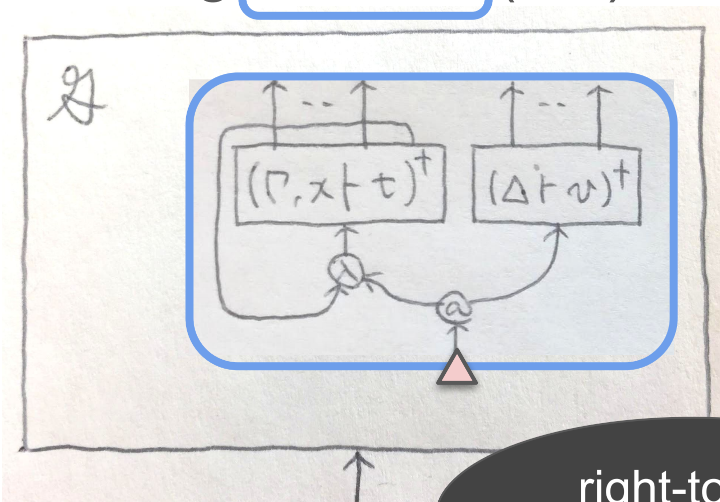
3. visiting the hole (1/1)

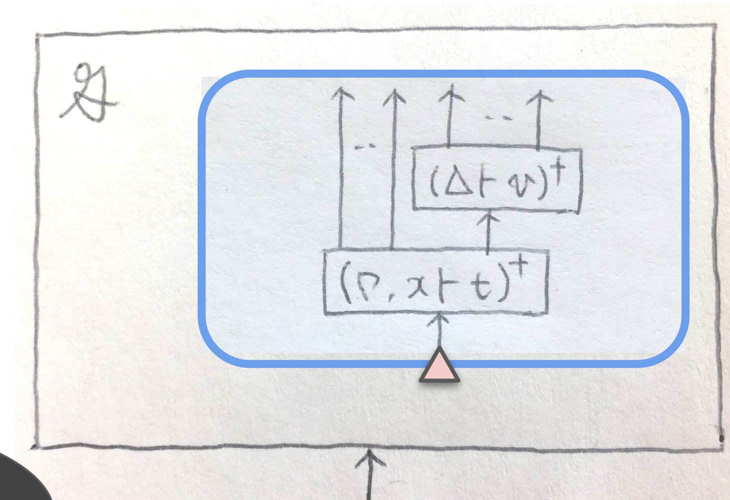




Case study: linear λ-calculus + “linear” recursion

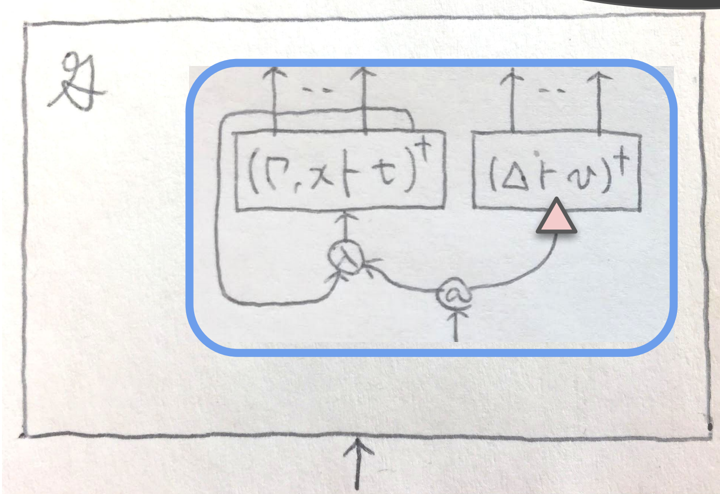
3. visiting the hole (1/1)





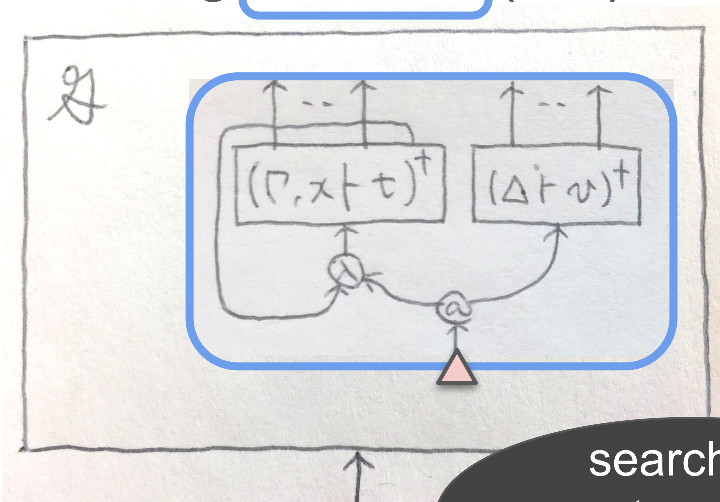
↓1

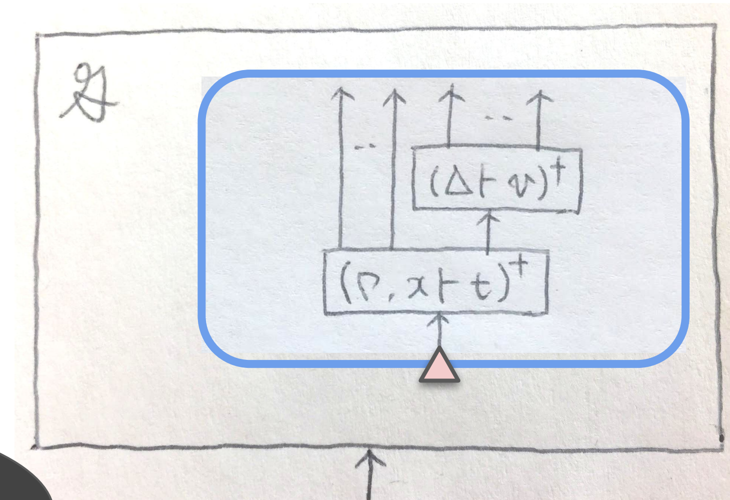
right-to-left
call-by-value



Case study: linear λ-calculus + “linear” recursion

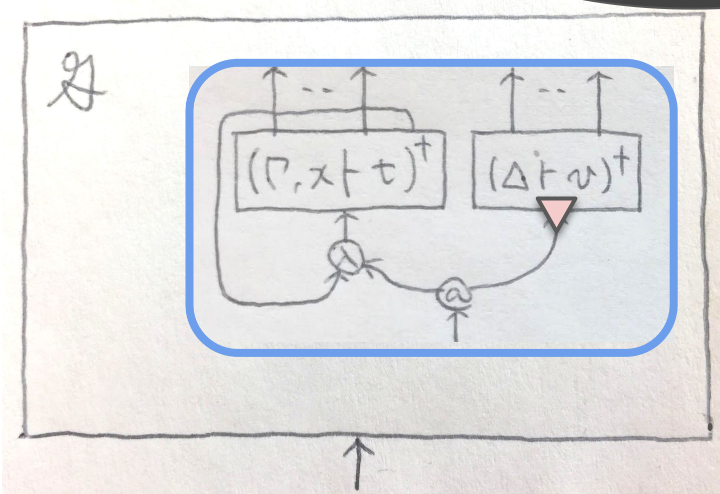
3. visiting the hole (1/1)





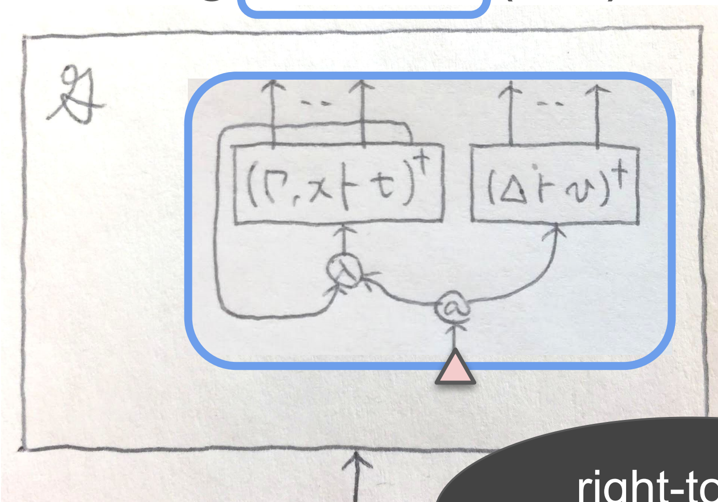
$\downarrow_2$

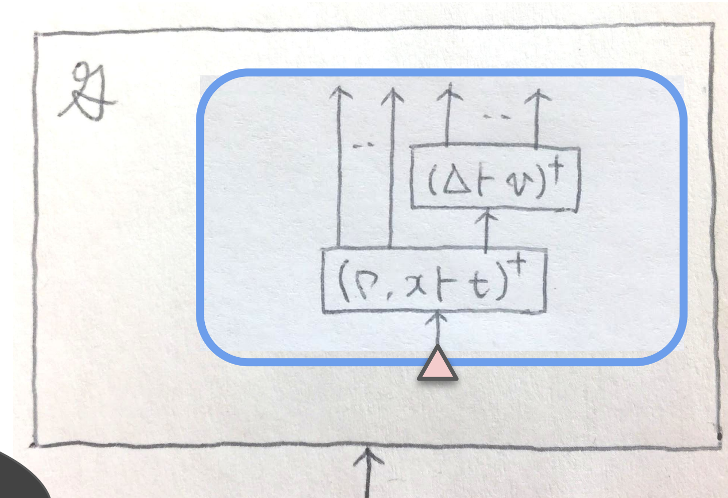
searching stopped at a value



Case study: linear λ-calculus + "linear" recursion

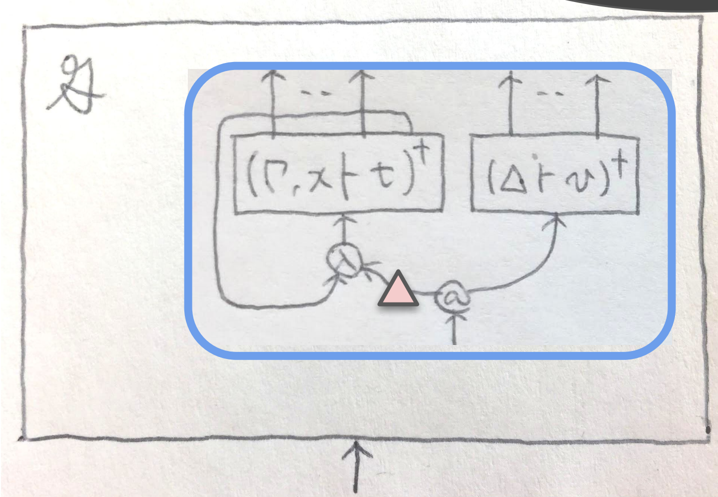
3. visiting the hole (1/1)





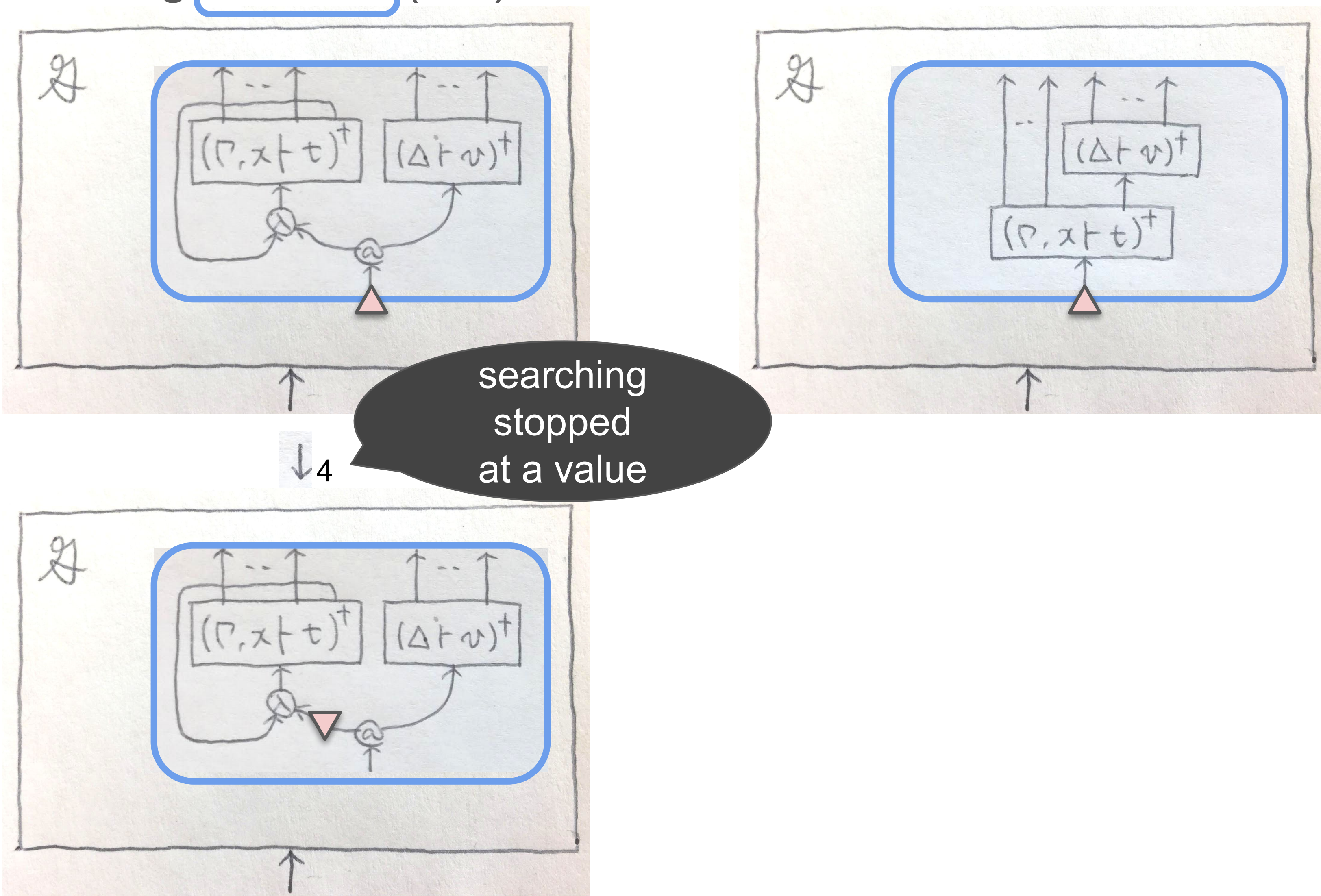
$\downarrow_3$

right-to-left
call-by-value



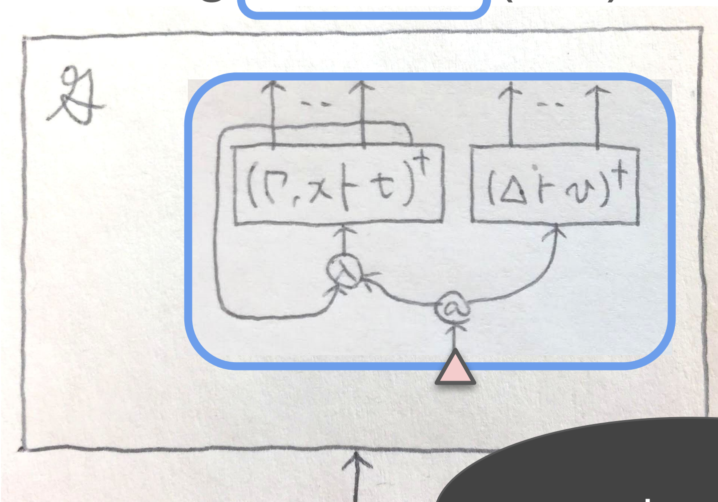
Case study: linear λ-calculus + “linear” recursion

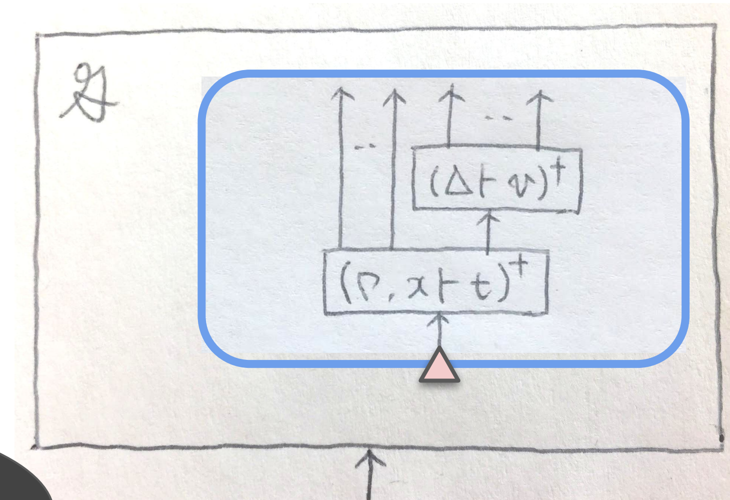
3. visiting the hole (1/1)



Case study: linear λ-calculus + “linear” recursion

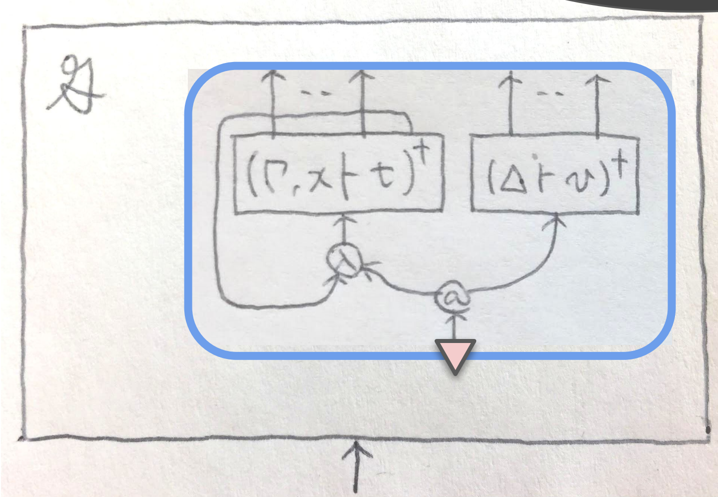
3. visiting the hole (1/1)





↓5

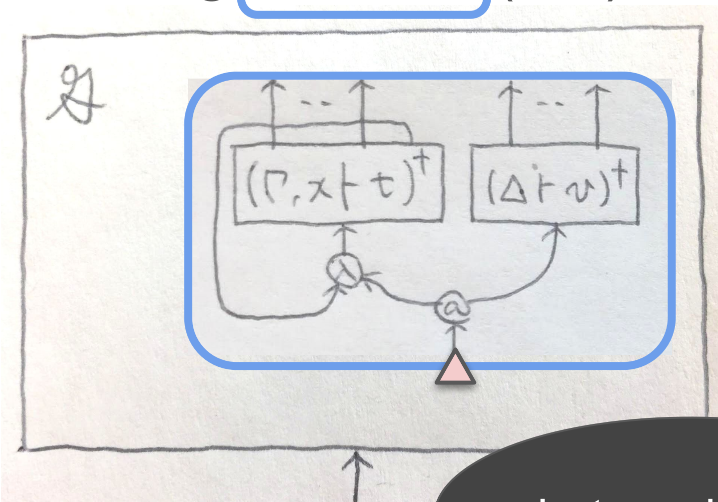
a redex found

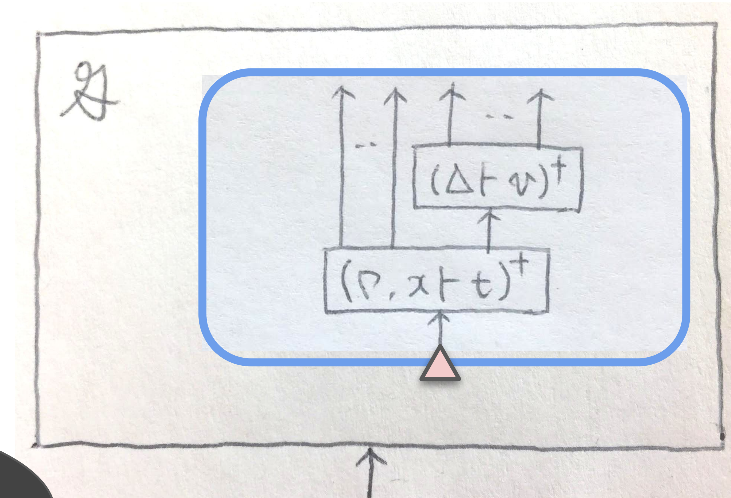


Muroya (U. B’ham. & RIMS, Kyoto U.)

Case study: linear λ-calculus + “linear” recursion

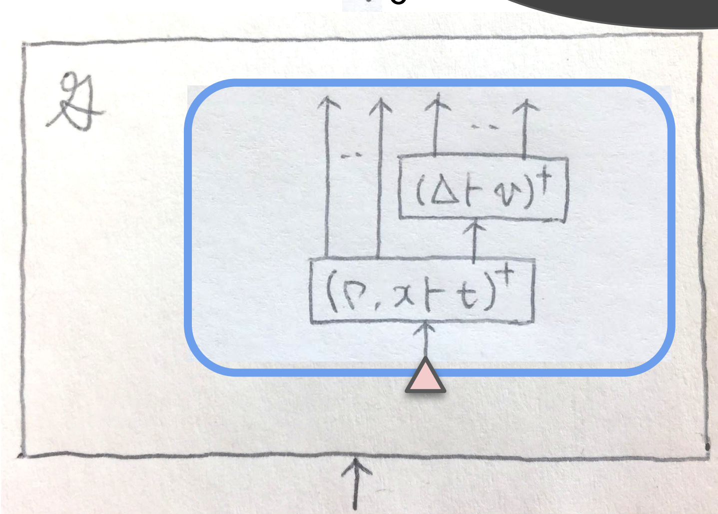
3. visiting the hole (1/1)





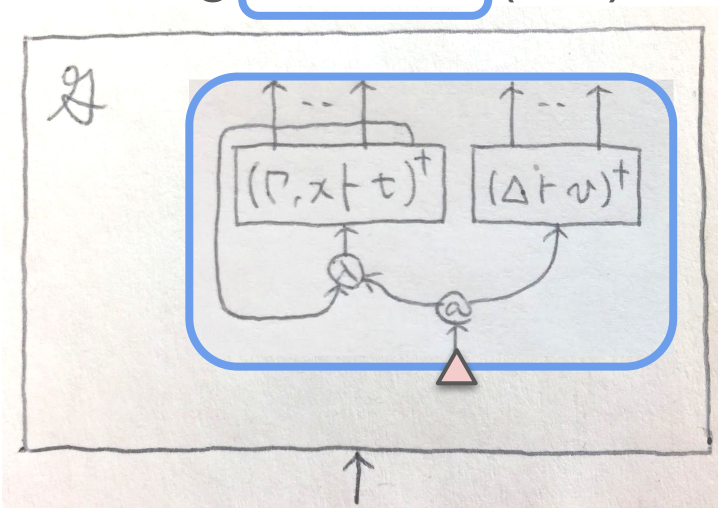
↓6

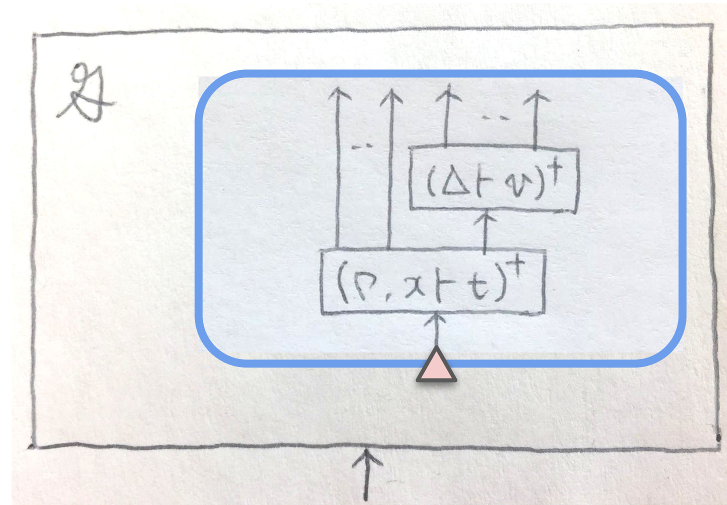
beta-reduction



Case study: linear λ-calculus + "linear" recursion

3. visiting the hole (1/1)

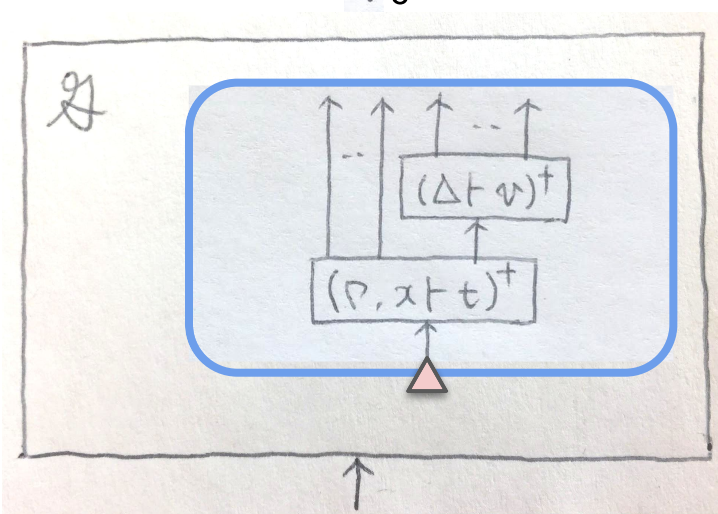




⟺

↓6

↓0



Case study: linear λ-calculus + "linear" recursion

3. visiting the hole (1 case)

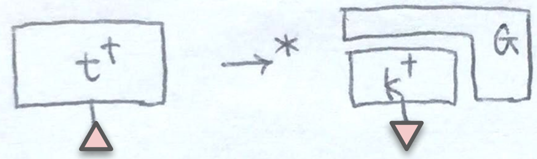
observation: the hole is reduced

Case study: linear λ-calculus + “linear” recursion

$$t ::= x \mid \lambda x.t \mid t\,t \mid k \mid \mu x.t$$
$$v ::= x \mid \lambda x.t \mid k$$

Given operational semantics:

closed term

$t \Downarrow k \overset{\Delta}{\iff}$  for some G.

define the contextual equivalence by:

$$t \simeq t' \overset{\Delta}{\iff} {}^{\forall}C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed, }$$
$$C[t] \Downarrow k \iff C[t'] \Downarrow k'$$
$$\text{Moreover, } k = k'$$

same basic constants

prove the beta-law *by step-wise reasoning*:

$$(\lambda x.t)\,v \simeq t[v/x]$$

and **observe** *some sufficient condition*.

Muroya (U. B’ham. & RIMS, Kyoto U.)

Case study: linear λ-calculus + “linear” recursion

… **prove** the beta-law *by step-wise reasoning*,

and **observe** *that*:

1. *redex searching* only inspects one node at each step
2. *rewriting* preserves, duplicates or simply reduces a beta-redex.
3. *rewriting* is “history-free”.

Case studies so far

… **prove** the beta-law *by step-wise reasoning*,

and **observe** *that*:

1. *redex searching* only inspects one node at each step
2. *rewriting* preserves, duplicates or simply reduces a beta-redex.
3. *rewriting* is “history-free”.

- ✓ untyped pure λ-calculus
- ✓ basic operations, recursion, if-statement
- ✓ control operators: call/cc, shift/reset
- ● algebraic effects & handlers

method needs to be slightly adjusted

Question

Given an extension of untyped λ-calculus,

what *operational*-semantic property of the extension

validates the call-by-value beta-law?

Answer?

A formal answer is yet to be stated…

But a graph-rewriting perspective provides:

- a useful & robust method
- key observations

Muroya (U. B'ham. & RIMS, Kyoto U.)