Local reasoning for robust observational equivalence

Kota Muroya
(RIMS, Kyoto U. & U. Birmingham)

Dan R. Ghica
Todd Waugh Ambridge
(U. Birmingham)

SYCO 7 & STRINGS III (Birmingham), 6 September 2019

In search of a diagrammatic language for ...

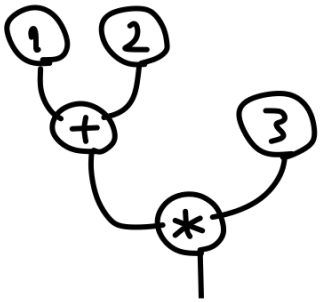
Local reasoning for robust observational equivalence

In search of a diagrammatic language for...

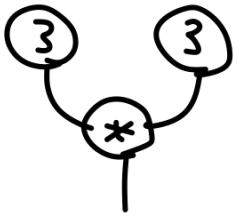
modelling program execution

2D representation of programs

(1 + 2) * 3



3 * 3



9

expected axioms

2D representation of programs

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

$=_\alpha\ (\lambda z.z)\ (\lambda z.z)$

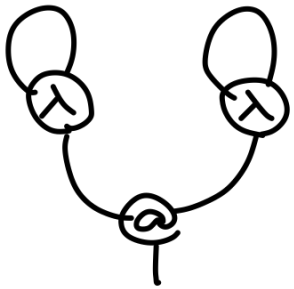
$\lambda y.y$

variables as wires

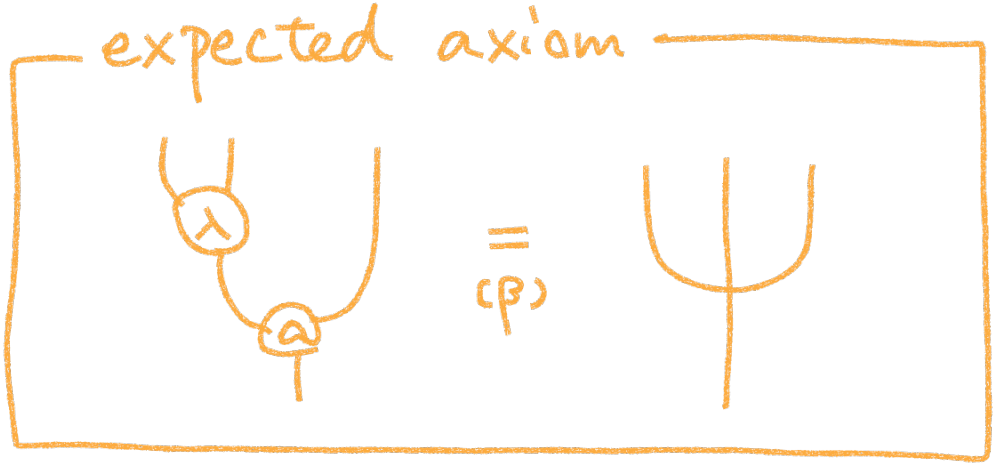
2D representation of programs

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

$=_\alpha\ (\lambda z.z)\ (\lambda z.z)$



$\lambda y.y$

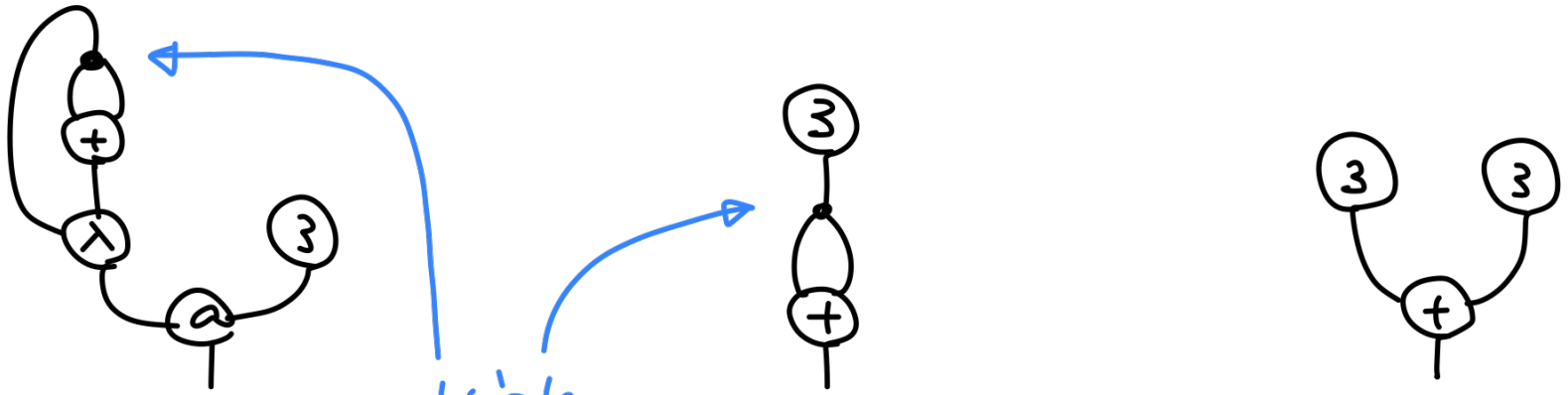


2D representation of programs

$(\lambda x.\, x + x)\ 3$

$\text{let } x = 3 \text{ in } x + x$

$3 + 3$



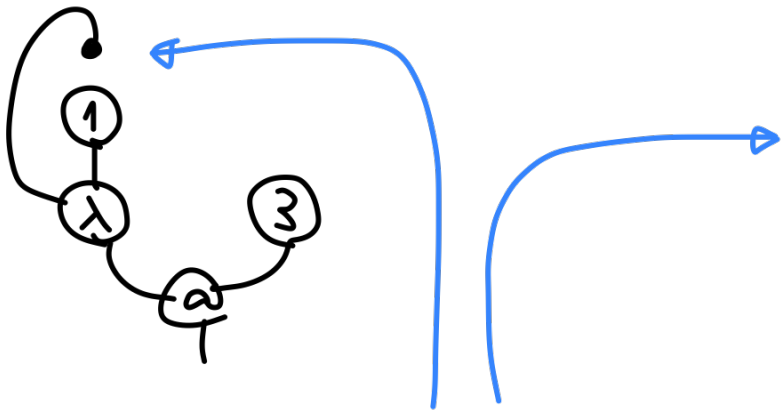
multiple occurrences of a variable

expected axioms

for copying

2D representation of programs

$(\lambda x.\ 1)\ 3$

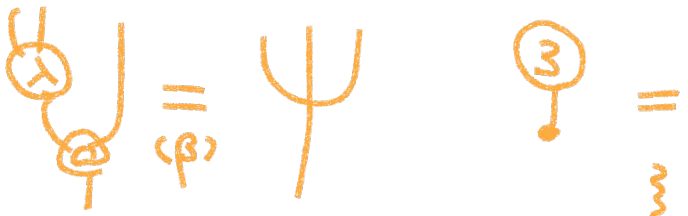


zero occurrence of a variable

$\text{let } x = 3 \text{ in } 1$

1

expected axioms



↓ for discarding

2D representation of programs

`new a = 1 in !a`

reference / location creation

dereference / read

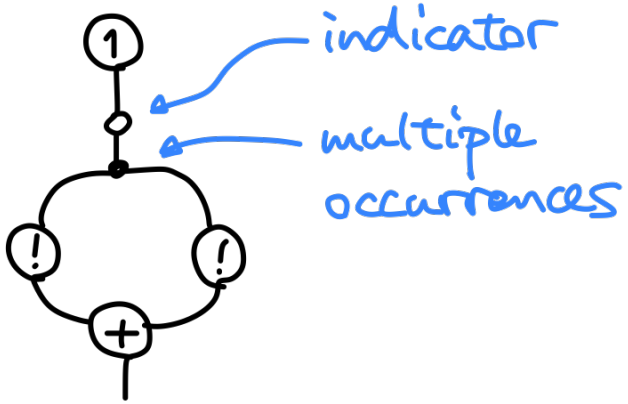
1

← reference / location indicator

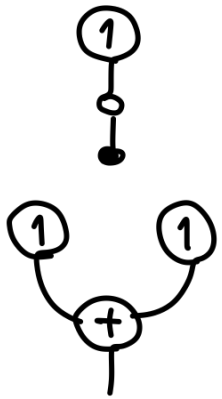
expected axiom

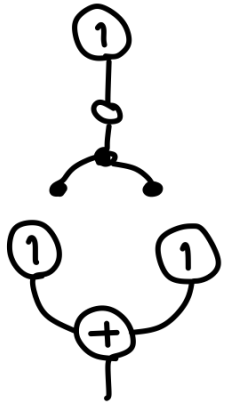
2D representation of programs

new a = 1 in !a + !a

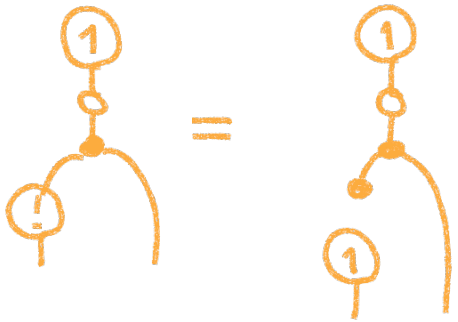


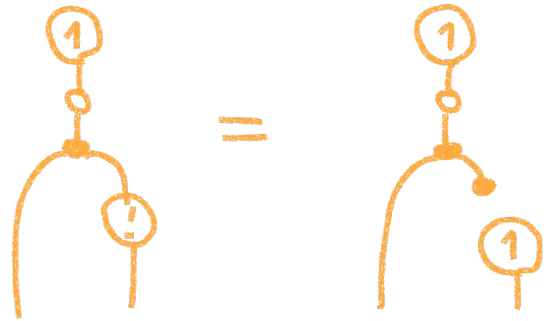
new a = 1 in 1 + 1





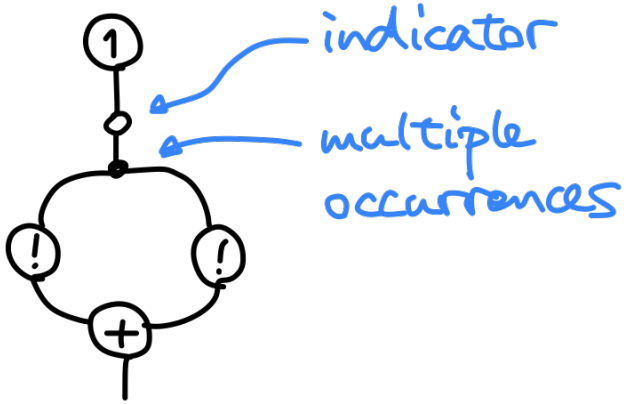
expected axioms



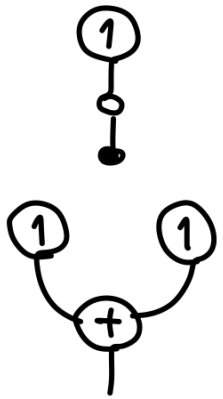


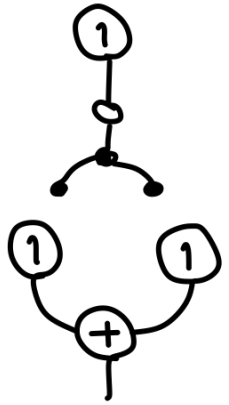
2D representation of programs

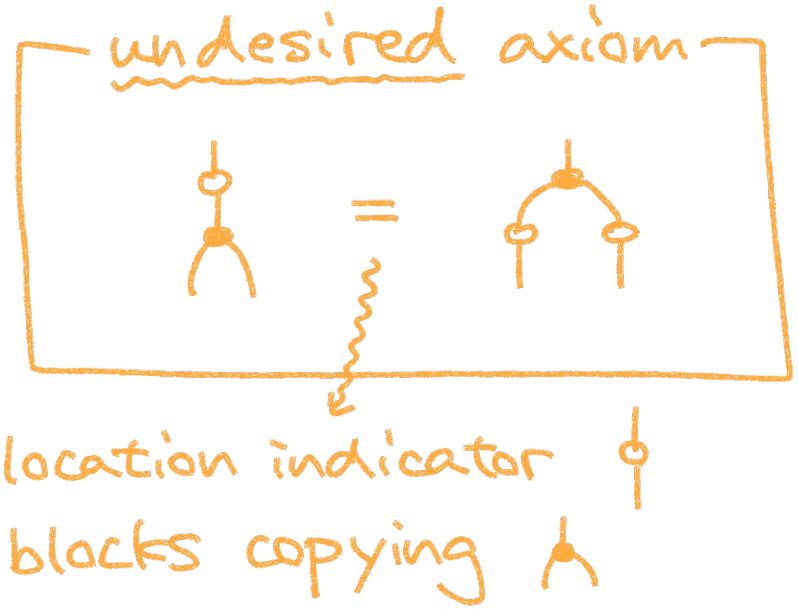
new a = 1 in !a + !a



new a = 1 in 1 + 1

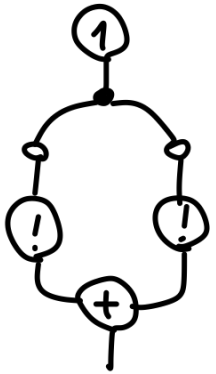






let x = 1 in

(new a = x in !a) + (new a = x in !a)



2D representation of programs

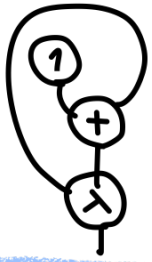
- name-free (α-equivalence built in)

$\lambda x.x$
$=_\alpha \lambda y.y$

- more refined & less structured than 1D syntax

diagrams with
no term counterpart
e.g.

$\lambda x.1+x$



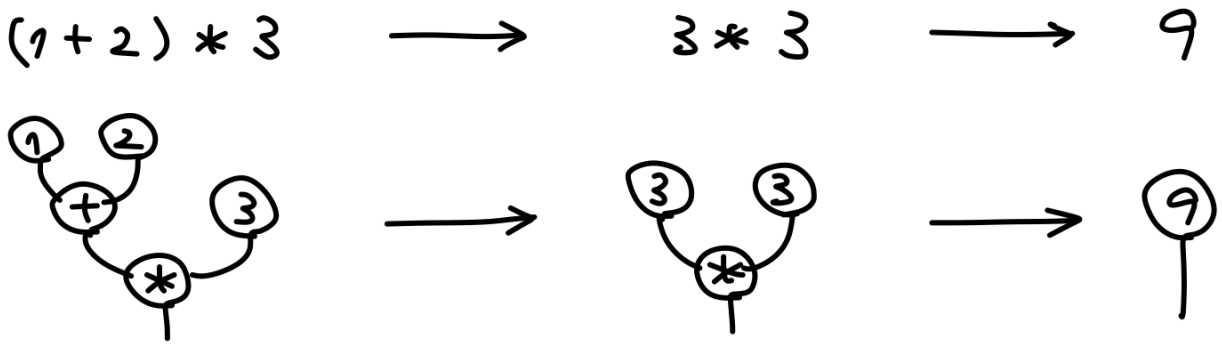
$\text{let } w=1 \text{ in } \lambda x.w+x$

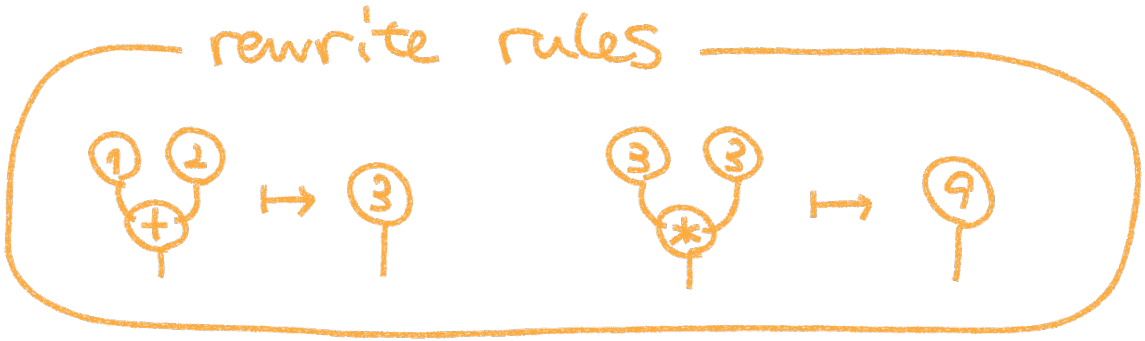
desired feature of a diagrammatic language

- copying vs. sharing

2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

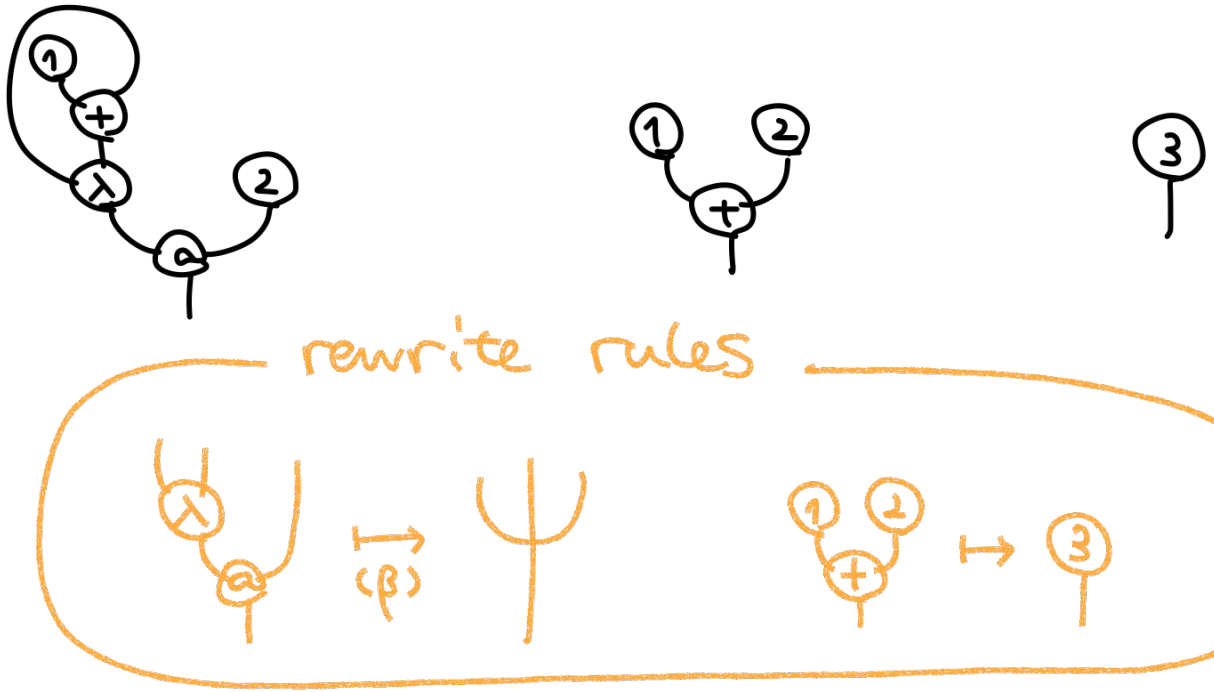




2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

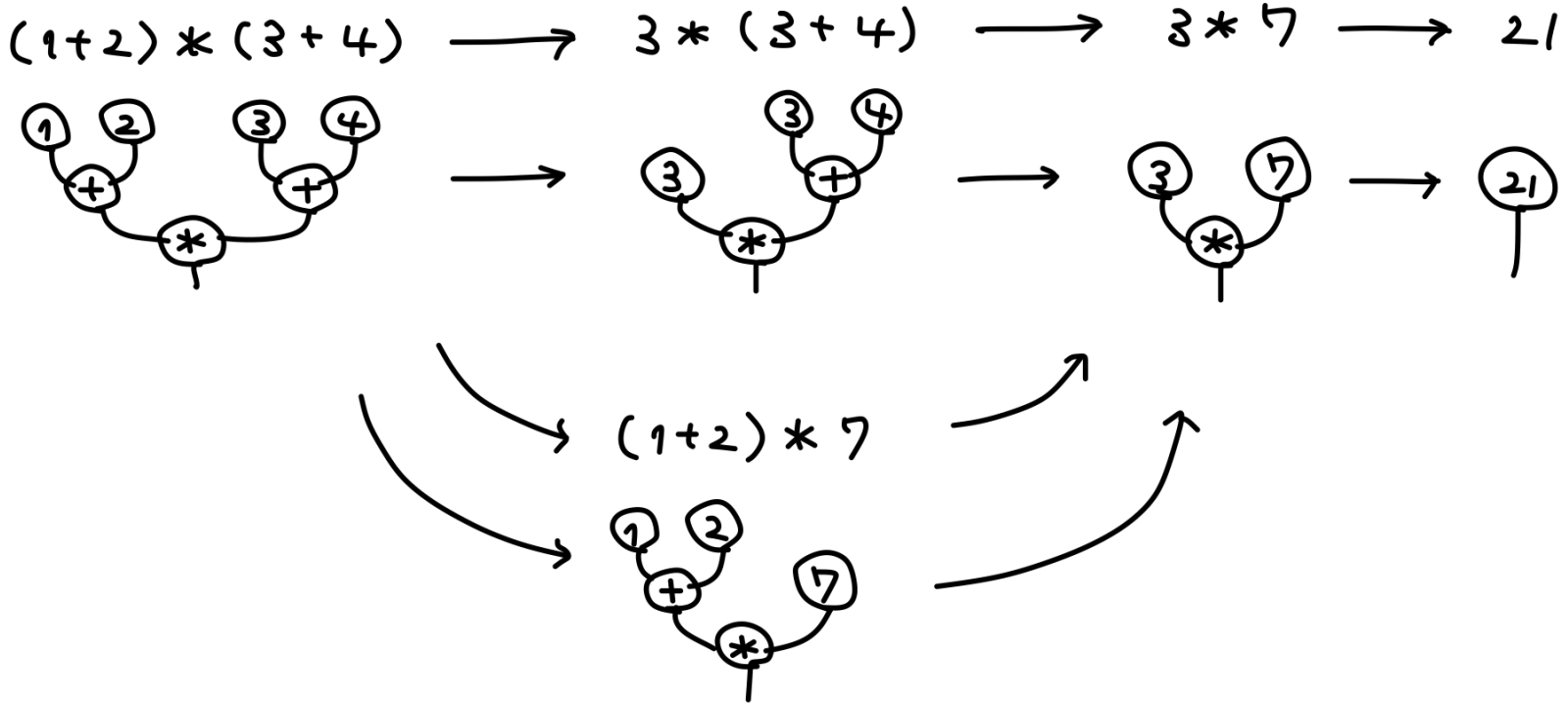
$(\lambda x.\ 1 + x)\ 2 \longrightarrow 1 + 2 \longrightarrow 3$



2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

- strategy of redex search

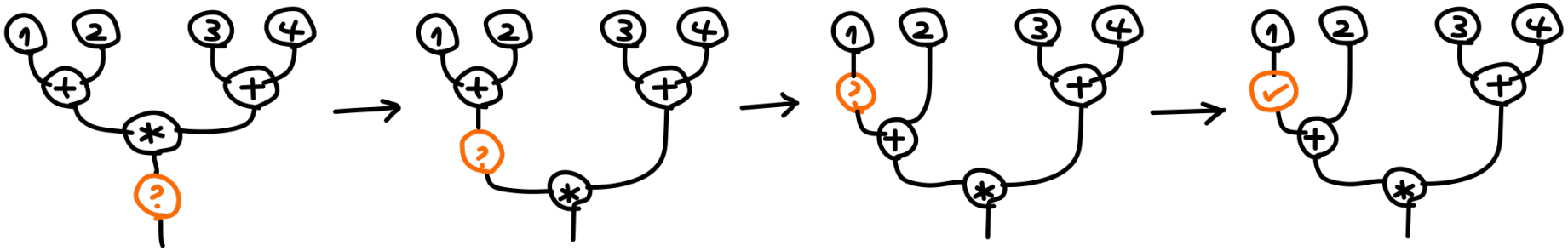


2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

▸ strategy of redex search specified by taken

$(1+2)*(3+4) \longrightarrow 3*(3+4) \longrightarrow 3*7 \longrightarrow 21$



(narrowing scope)

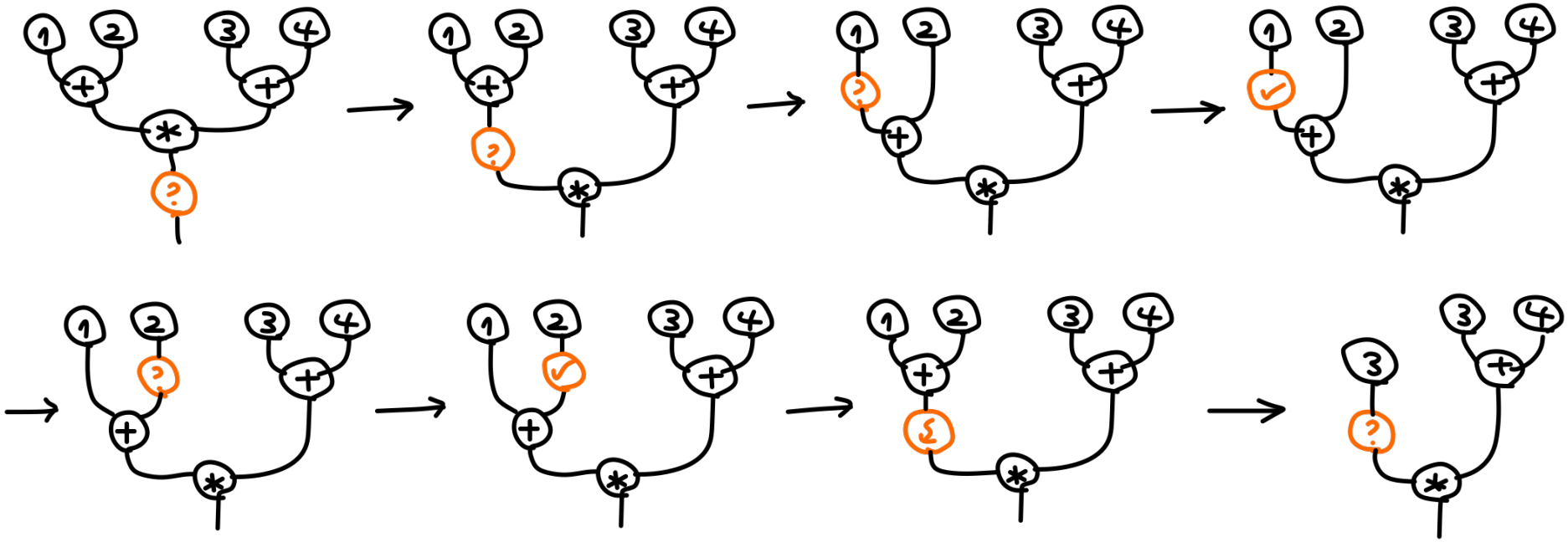
(found a value)

2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

▷ strategy of redex search specified by taken

$(1+2)*(3+4) \longrightarrow 3*(3+4) \longrightarrow 3*7 \longrightarrow 21$

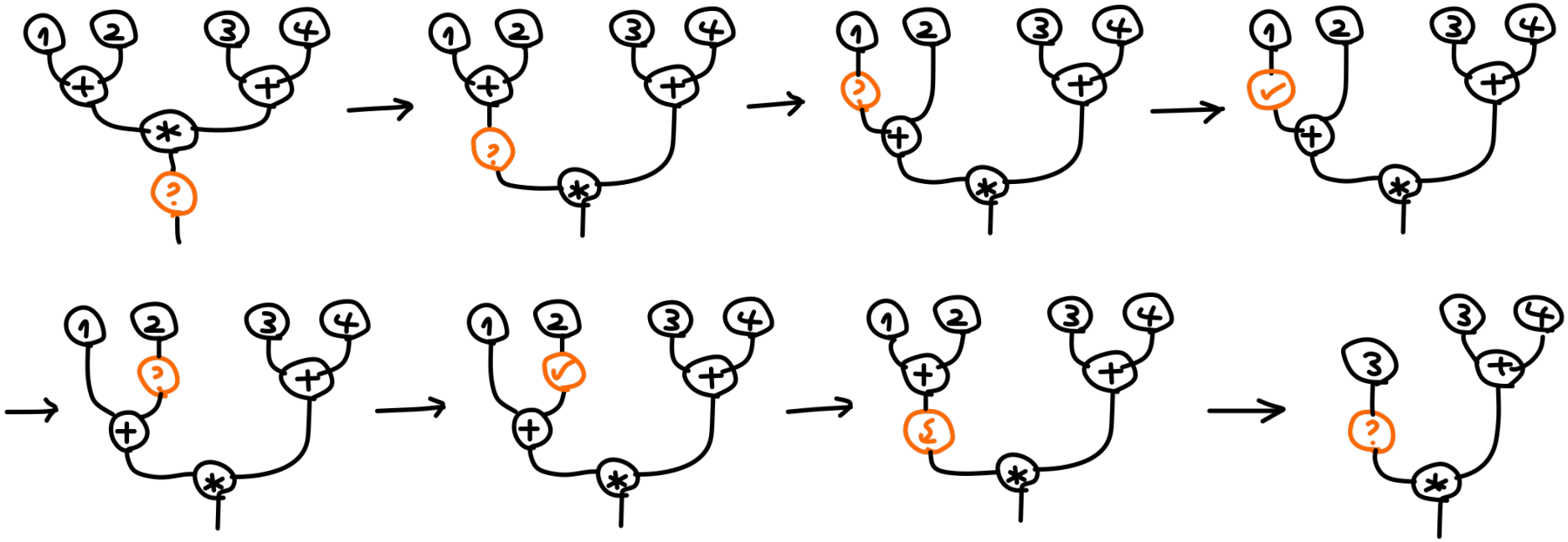


2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

- strategy of redex search specified by token

redex search is also rewriting



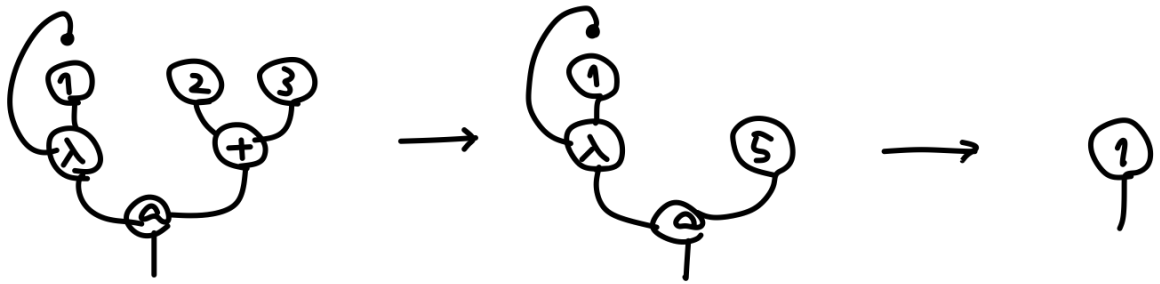
2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

- strategy of redex search specified by taken

redex search is also rewriting

$(\lambda x.\,1)\ (2+3) \longrightarrow (\lambda x.\,1)\ 5 \longrightarrow 1$



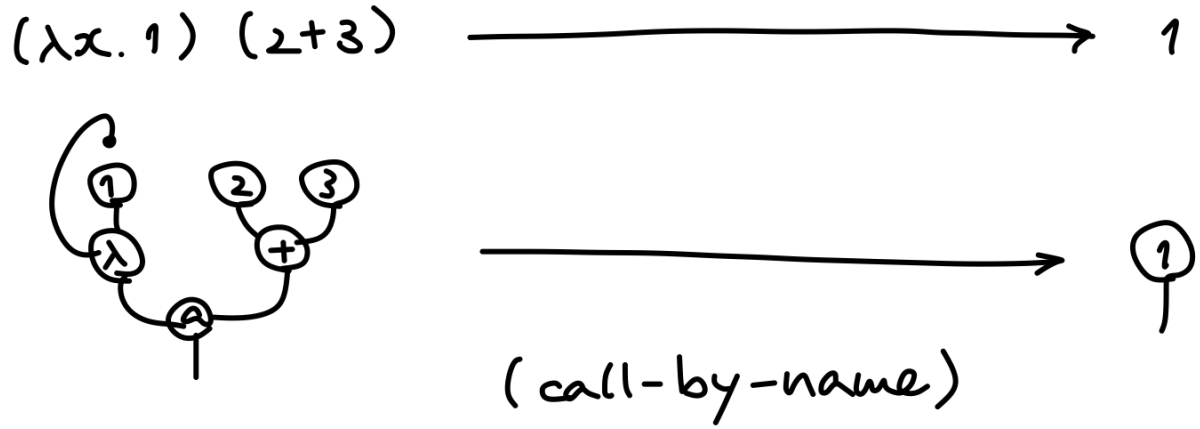
(call-by-value)

2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

- strategy of redex search specified by taken

redex search is also rewriting

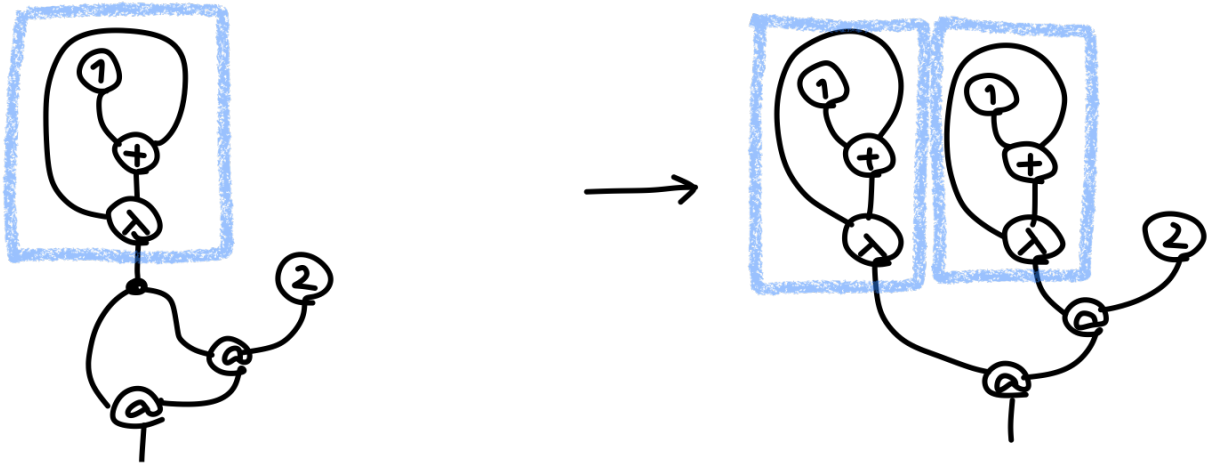


2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

▸ strategy of duplication

let $u = \lambda x.\, 1 + x$ in $u\,(u\,2) \longrightarrow (\lambda x.\, 1+x)\,((\lambda x.\, 1+x)\,2)$

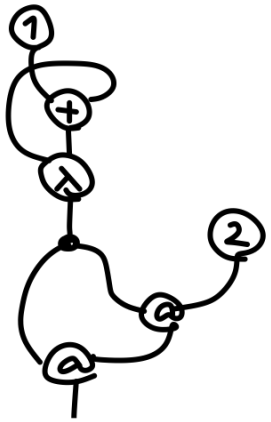


2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

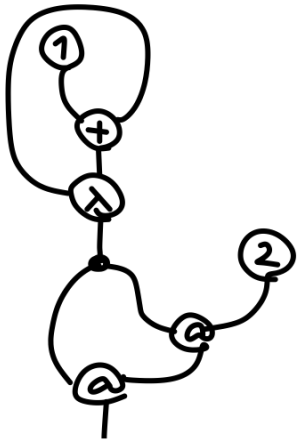
▷ strategy of duplication

let $u = (\text{let } w = 1 \text{ in } \lambda x.\, w + x)$ in $u\,(u\,2)$



[cf.

let $u = \lambda x.\, 1 + x$ in $u\,(u\,2)$



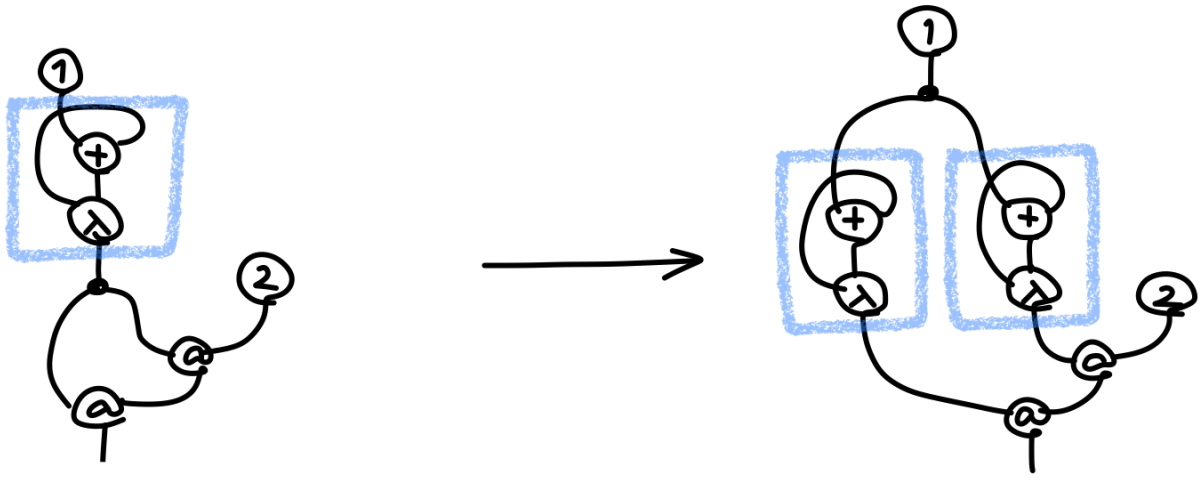
]

2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

▸ strategy of duplication

let $u = (\text{let } w = 1 \text{ in } \lambda x.\, w + x)$ in $u\ (u\ 2)$



let $w = 1$ in
let $u = \lambda x.\, w + x$ in $u\ (u\ 2)$

→

let $w = 1$ in
$(\lambda x.\, w + x)\ ((\lambda x.\, w + x)\ 2)$

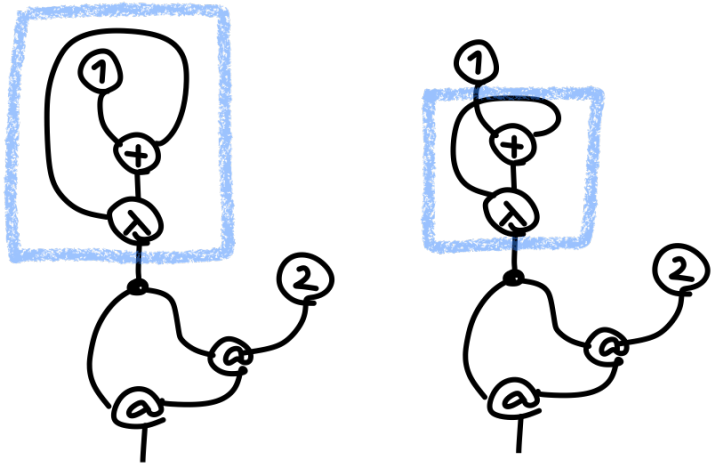
2D modelling of program execution

modelling dynamic (operational) behaviour
with strategical diagram-rewriting

- strategy of duplication
  specified by unit blocks of duplication

equip diagrams with
a black / box structure

(graph-theoretically :
nodes labelled with
a graph)

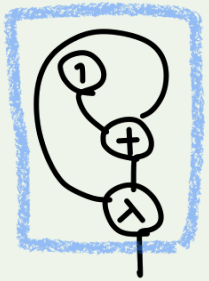


2D modelling of program execution

modelling dynamic (operational) behaviour
with strategical diagram-rewriting

- strategy of duplication
  specified by unit blocks of deferral

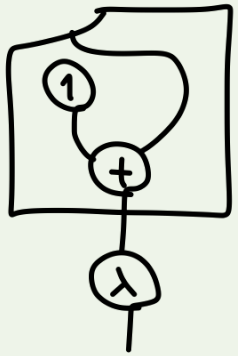
unit of duplication



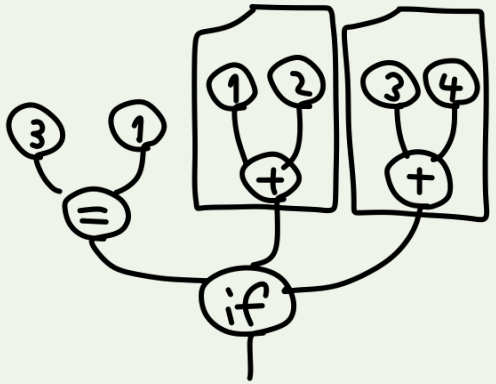
refinement

unit of deferral

$\lambda x.\ 1 + x$



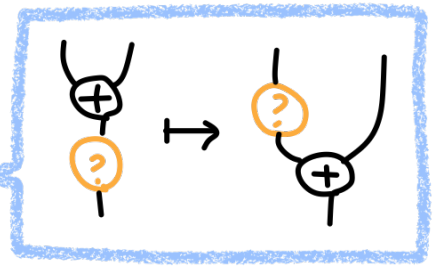
if 3 = 1 then 1+2 else 3+4



2D modelling of program execution

modelling dynamic (operational) behaviour
with strategical diagram-rewriting

- strategy of redex search:
  specified by rewriting with token



- strategy of duplication:
  specified by unit blocks of duplication/deferral

desired feature of a diagrammatic language

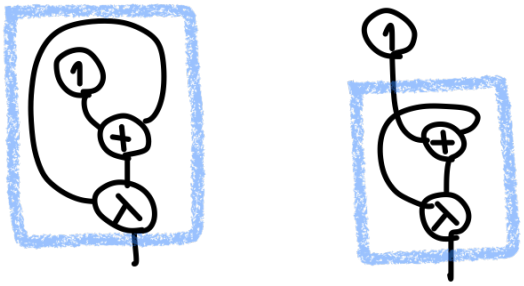
- block/box structure

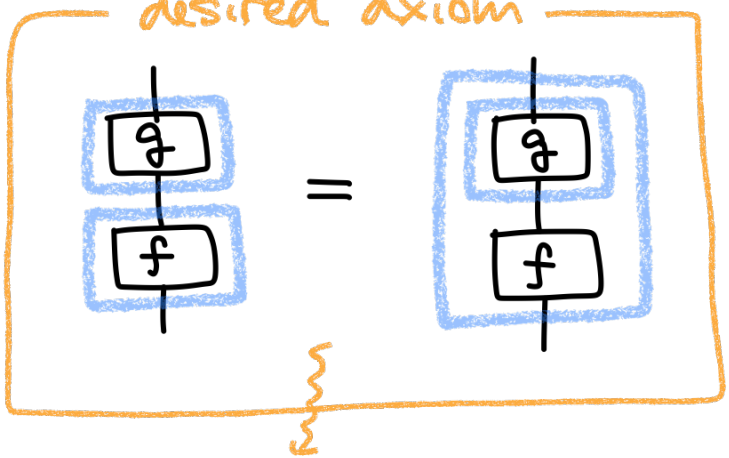
2D modelling of program execution

modelling dynamic (operational) behaviour with strategical diagram-rewriting

desired feature of a diagrammatic language

- block/box structure





not a functorial box [Mellies]

2D modelling of program execution

modelling dynamic (operational) behaviour
with strategical diagram-rewriting

... but, modelling for what ?

an answer: proving that two program fragments
have the same behaviour

2D modelling of program execution

exercise prove that 'new a=1 in λx. !a' and 'λx. 1' have the same (dynamic) behaviour in any possible programs

tryal with terms

let u = (new a=1 in λx. !a) in (u 0) + (u 0)

→ new a=1 in ((λx. !a) 0) + ((λx. !a) 0)

let u = λx. 1 in (u 0) + (u 0)

→ ((λx. 1) 0) + ((λx. 1) 0)

tracing non sub-terms

2D modelling of program execution

exercise prove that 'new a=1 in λx. !a' and 'λx. 1' have the same (dynamic) behaviour in any possible programs

tryal with diagrams

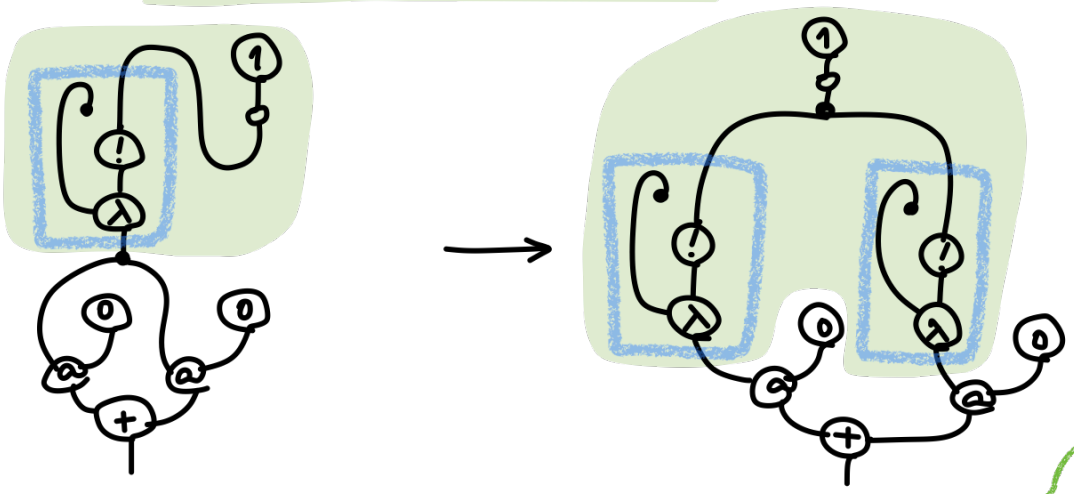
let u = (new a=1 in λx. !a) in (u 0) + (u 0)

let u = λx. 1 in (u 0) + (u 0)

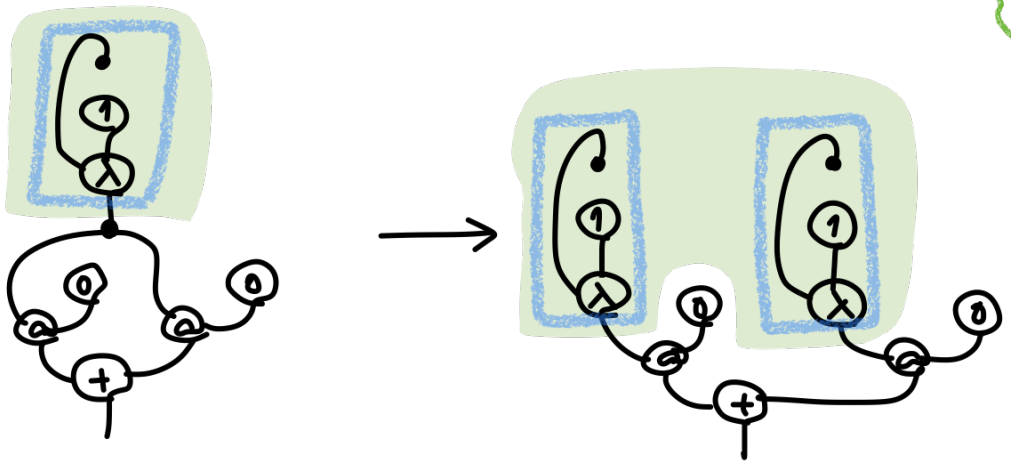
2D modelling of program execution

tryal with diagrams

let u = (new a=1 in λx.!a) in (u 0) + (u 0)



let u = λx.1 in (u 0) + (u 0)



tracing sub-diagrams

2D modelling of program execution

modelling dynamic (operational) behaviour
with strategical diagram-rewriting

... but, modelling for what ?

an answer: proving that two program fragments
have the same behaviour

⇨ proof possible by tracing sub-diagrams

2D modelling of program execution

modelling dynamic (operational) behaviour
with strategical diagram-rewriting

> 2D language is ...
>
> more refined & less structured than 1D syntax

- We can analyse dynamic behaviour
  by tracing sub-diagrams that may not be sub-terms.
- We need to keep some useful aspects of
  the (inductive) structure of 1D syntax.

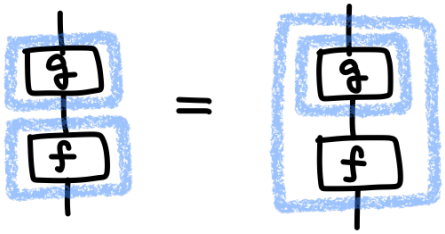
2D modelling of program execution

modelling dynamic (operational) behaviour
with strategical diagram-rewriting

> what is the right 2D, diagrammatical, language?

- copying vs. sharing

- black/box structure



- tracing sub-diagrams
  (zoom-in / zoom-out ability?)

We exploit locality more than compositionality.