λ-Circuit

An informal method to reason about λ-calculus diagrammatically

Chenchao Ding

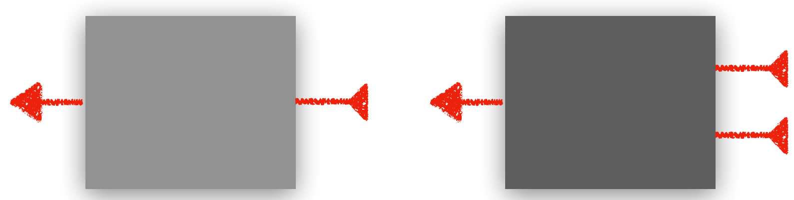
Part I: Introduction

"A program is seen as a machine.
To make sense of it, one must observe its operation."

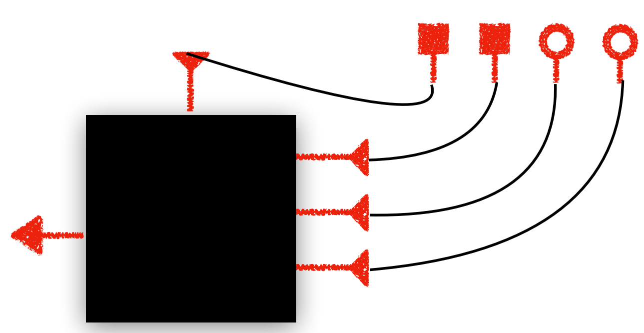
- Turchin, *The Concept of a Supercompiler*

Physics/Machine/Circuits

wires



circuit templates



wire connections
& template instantiations

~

Logic/Language/Lambdas

x, y, z ...

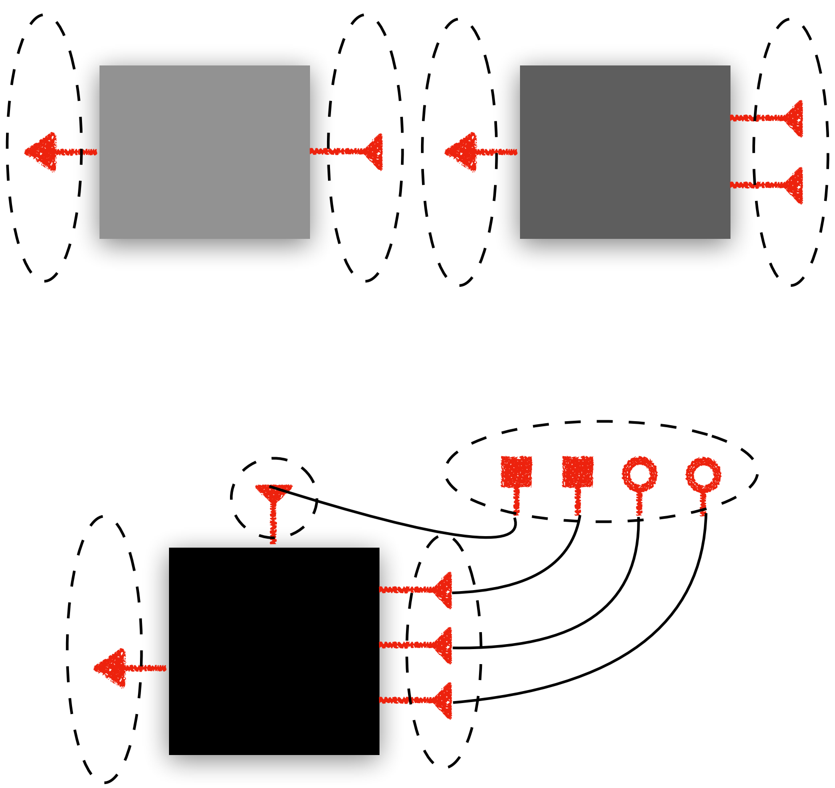
variables

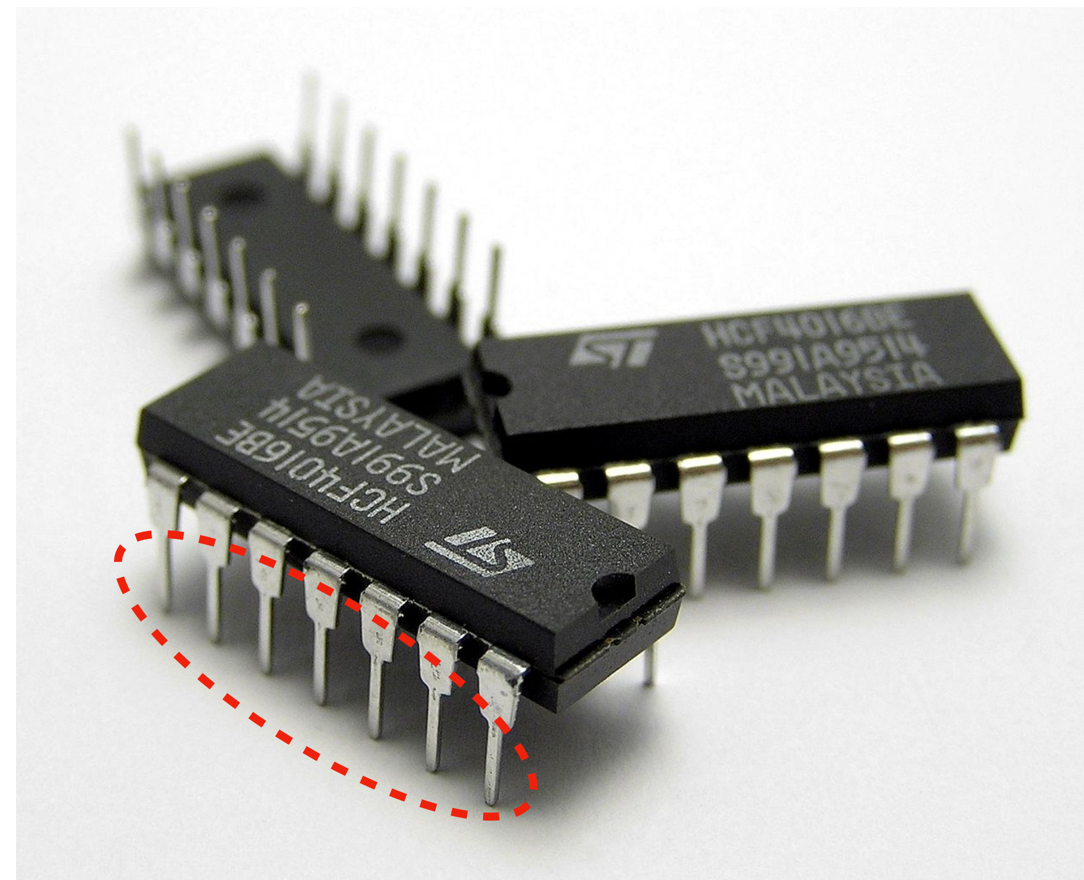
λx.E, λxy.E

λ abstractions

(λx.E a), (λxy.E a b)

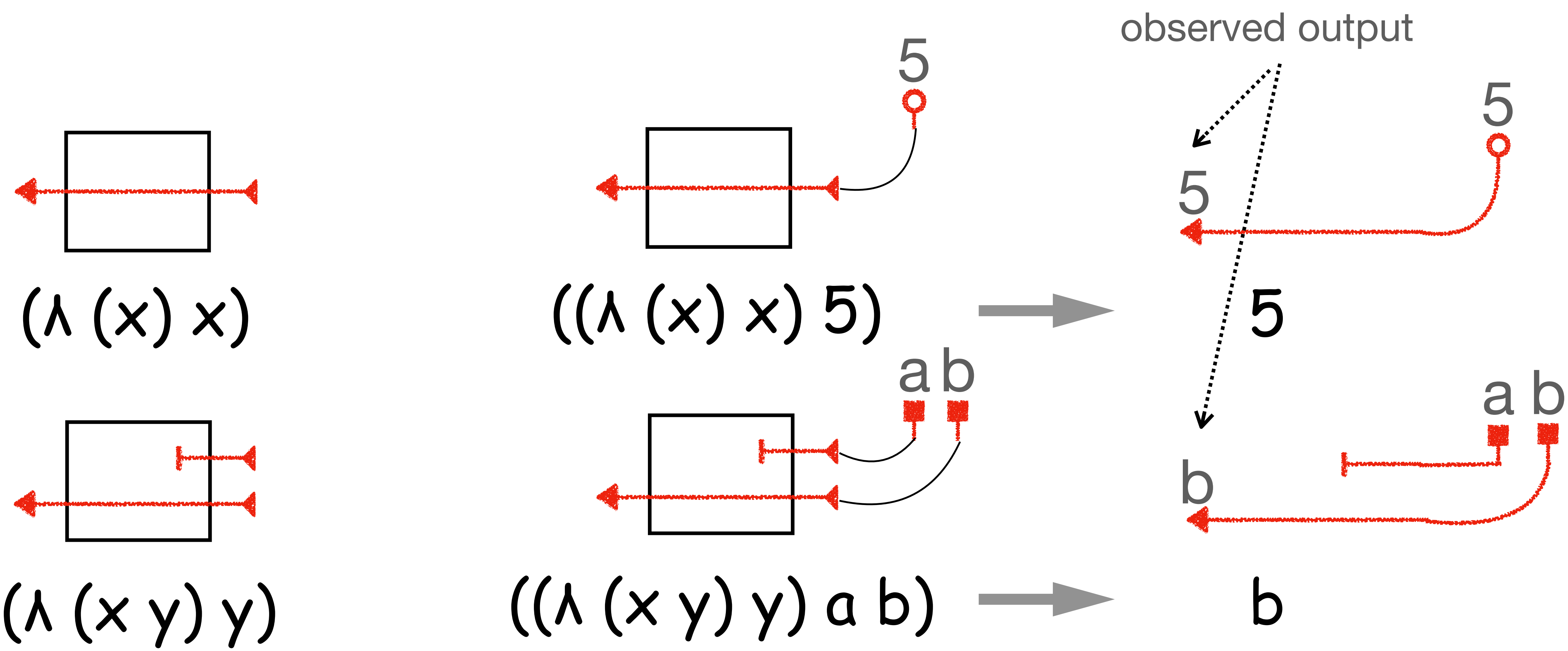
applications



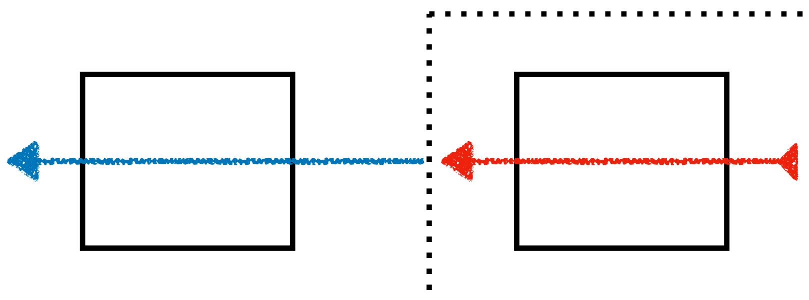


electronic pins and sockets

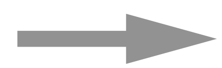
Simple Examples

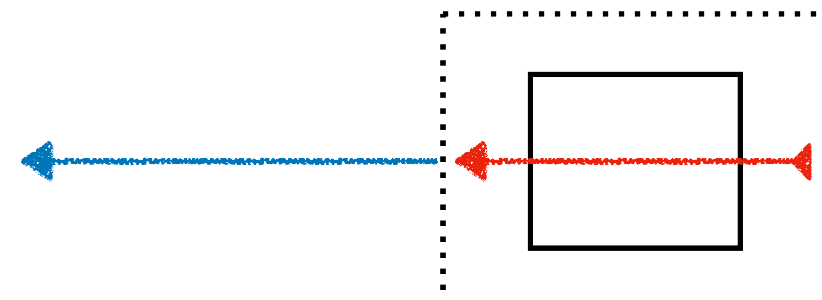


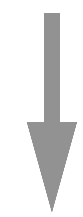
Higher-Order Functions

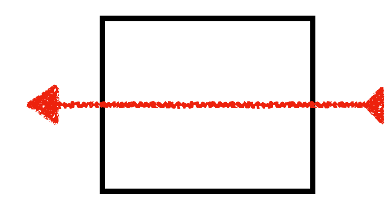


((λ (x) x) (λ (x) x))





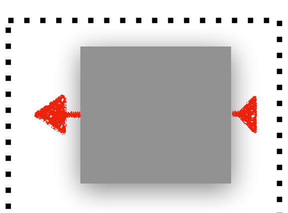
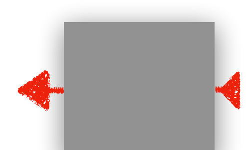




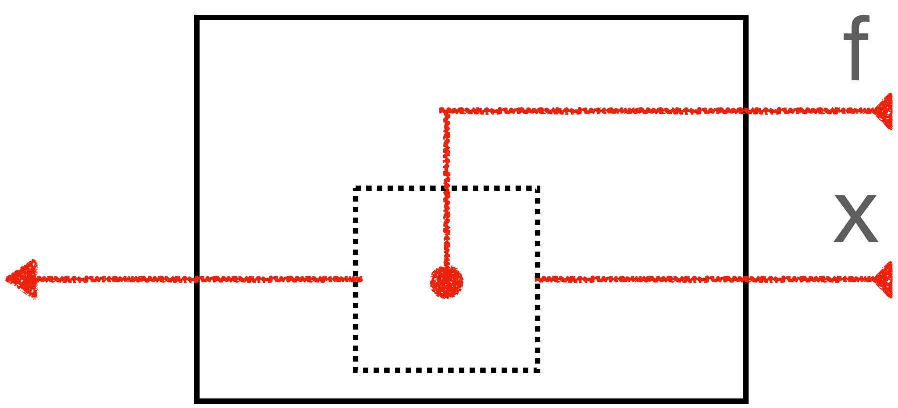
(λ (x) x)

variables can carry values of functions

wires can conduct “values” of circuits

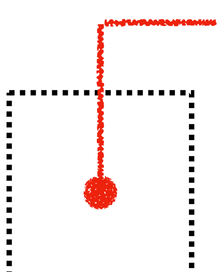
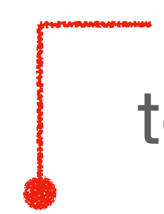
 means “value of” 

Higher-Order Functions

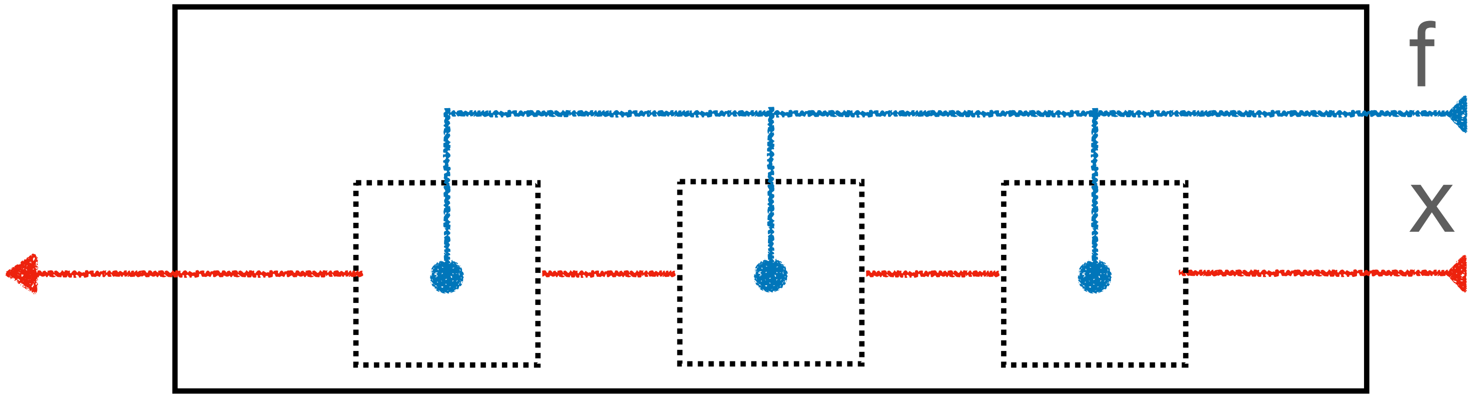


defer/suspend the evaluation of body

(λ (f x) (f x))

 is waiting for a circuit bound by  to “fill in the blank”

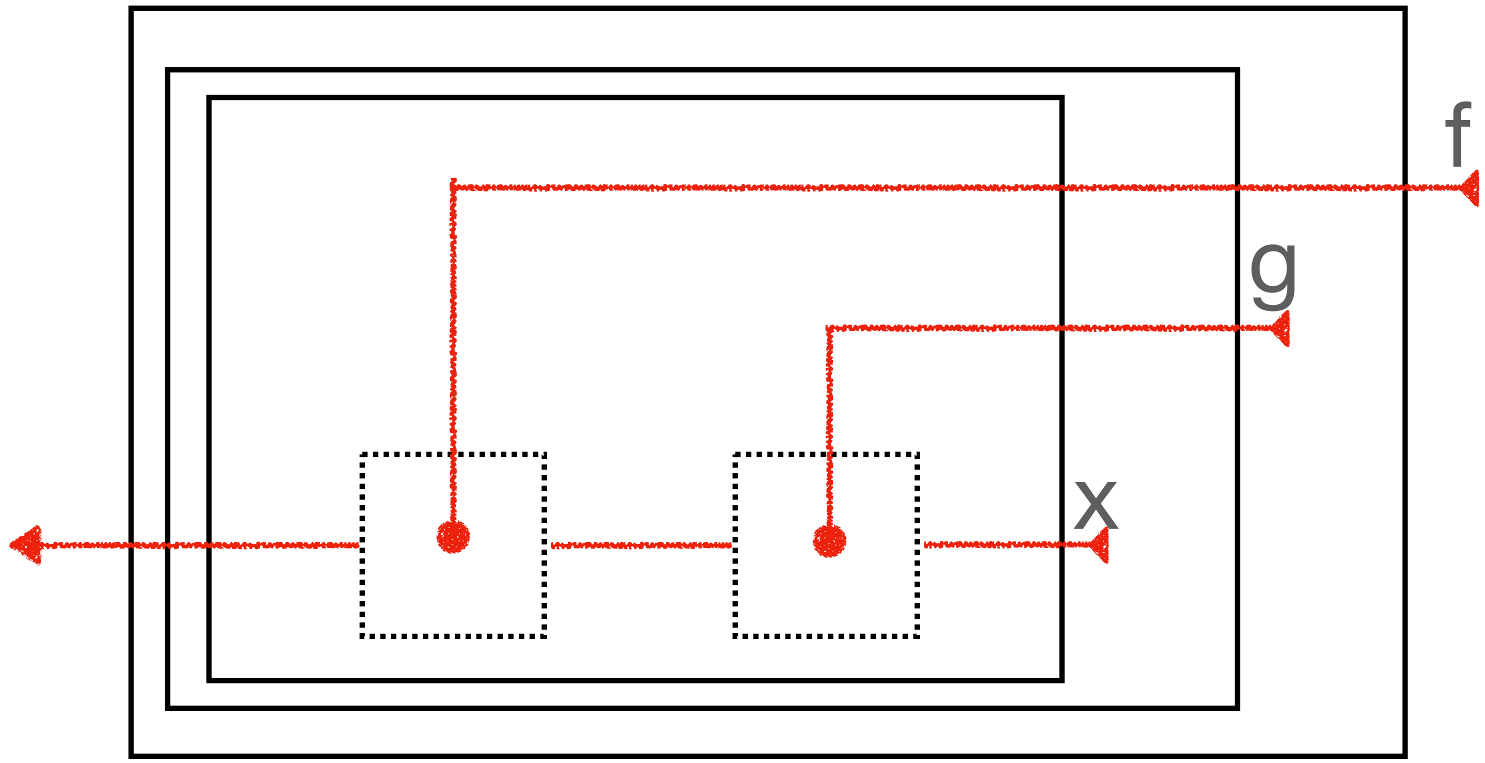
Duplicated Circuits



(λ (f x) (f (f (f x))))

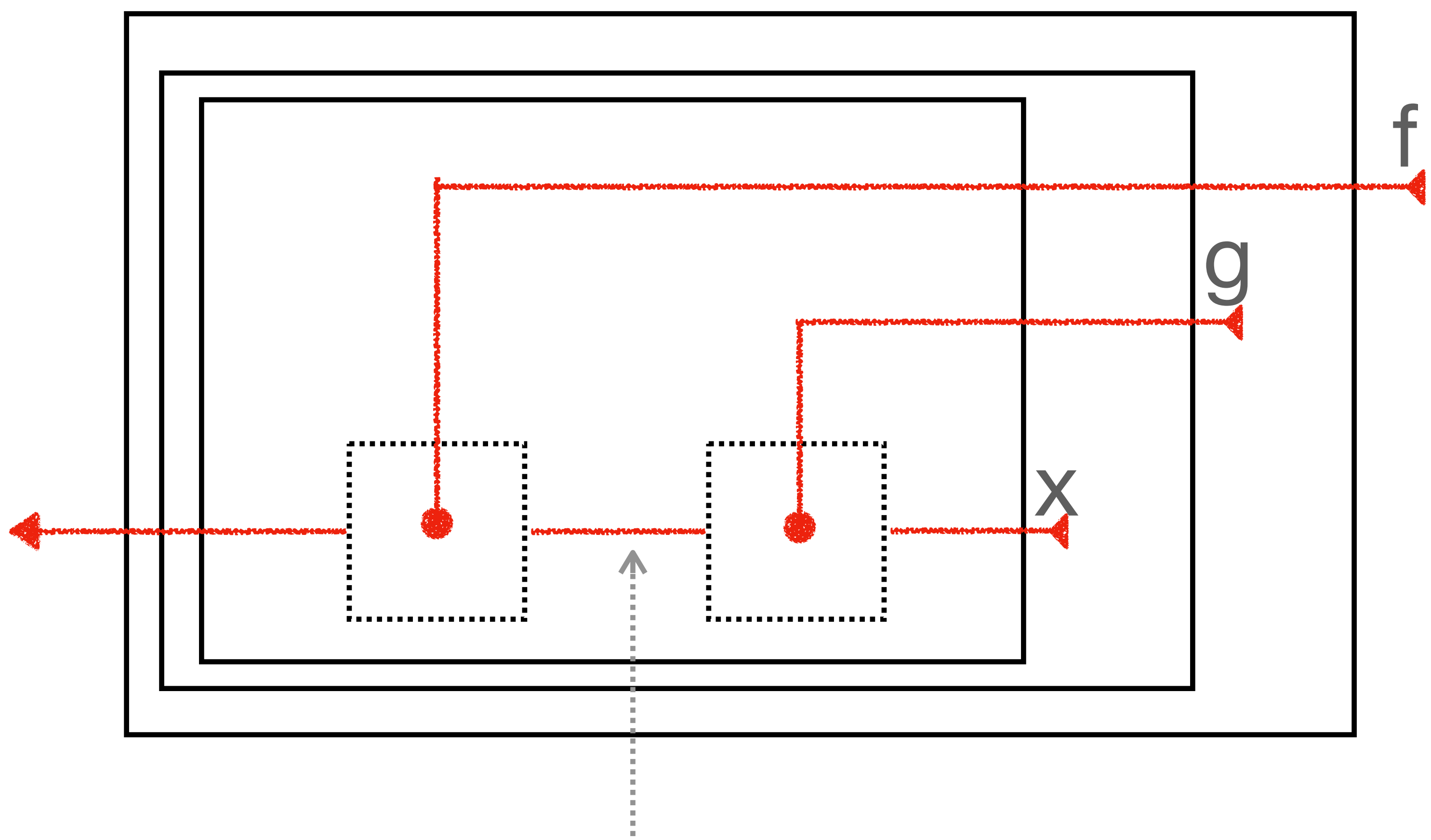
the wires in blue always carry the same “value”

A Less Simple Example: *Compose*



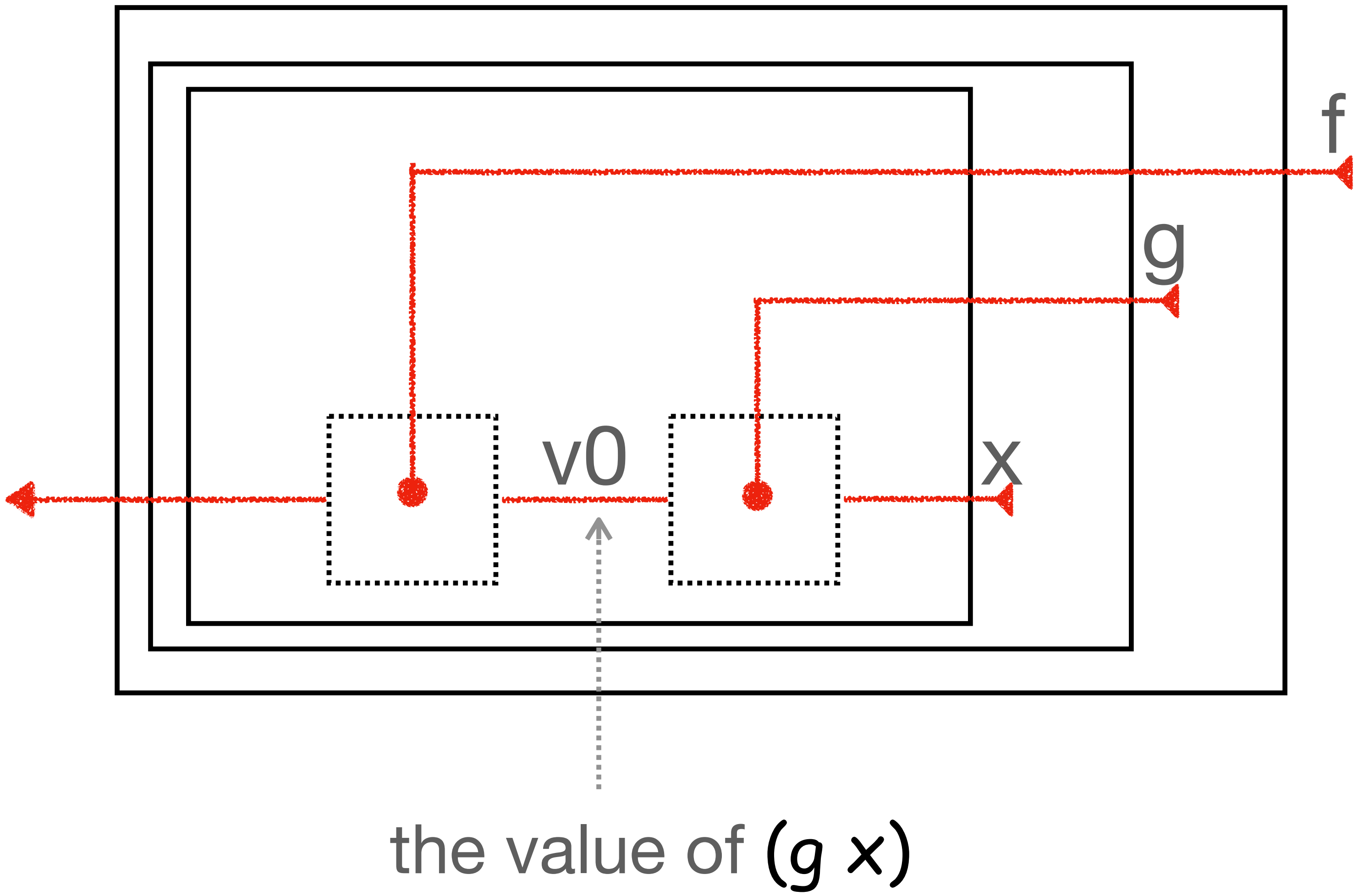
let compose = (λ (f) (λ (g) (λ (x) (f (g x)))))

A Less Simple Example: *Compose*



(λ (f)
  (λ (g)
    (λ (x) (f (g x)))))

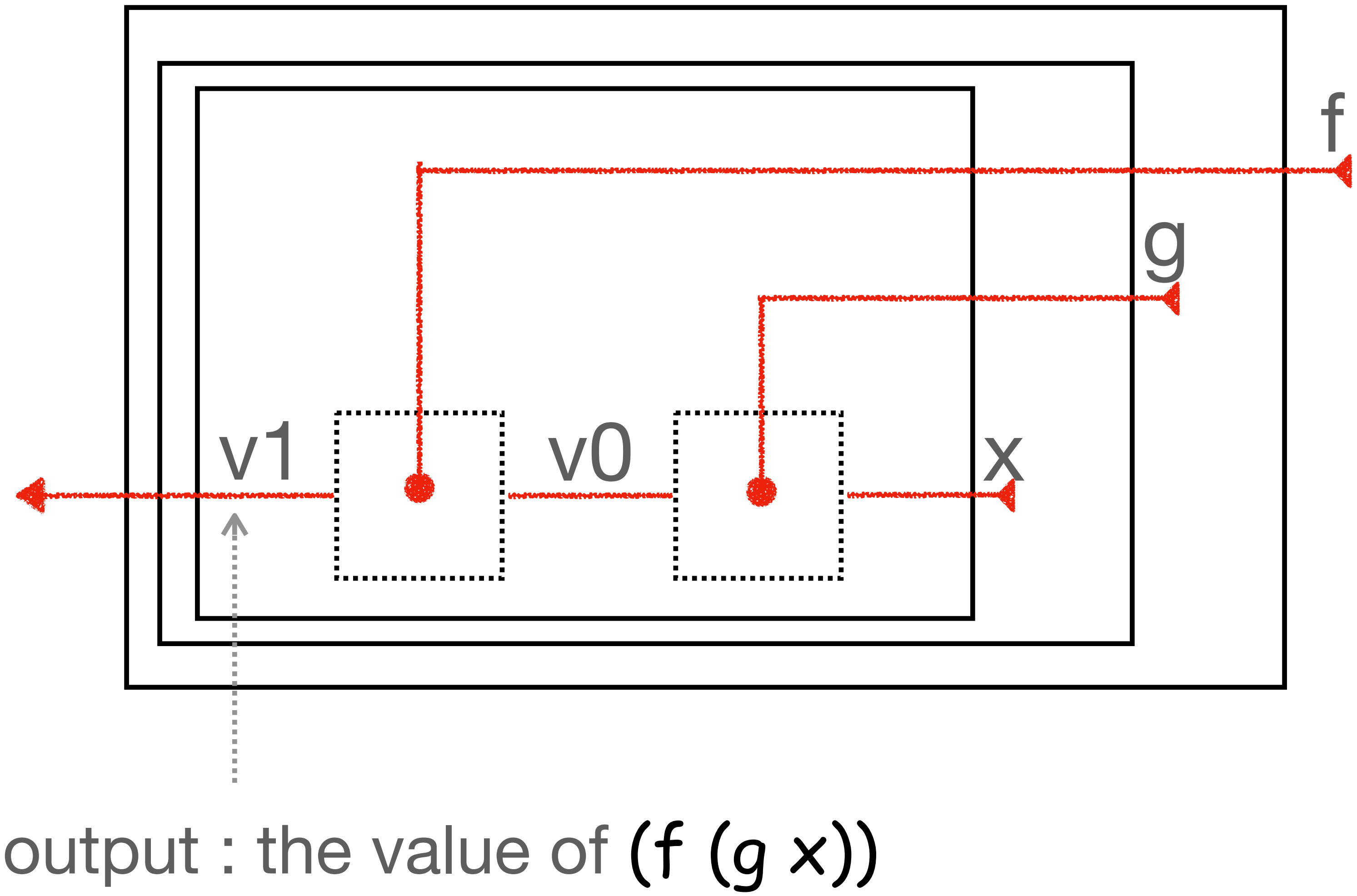
A Less Simple Example: *Compose*



(λ (f)
  (λ (g)
    (λ (x) (f (g x)))))

v0 = (g x)

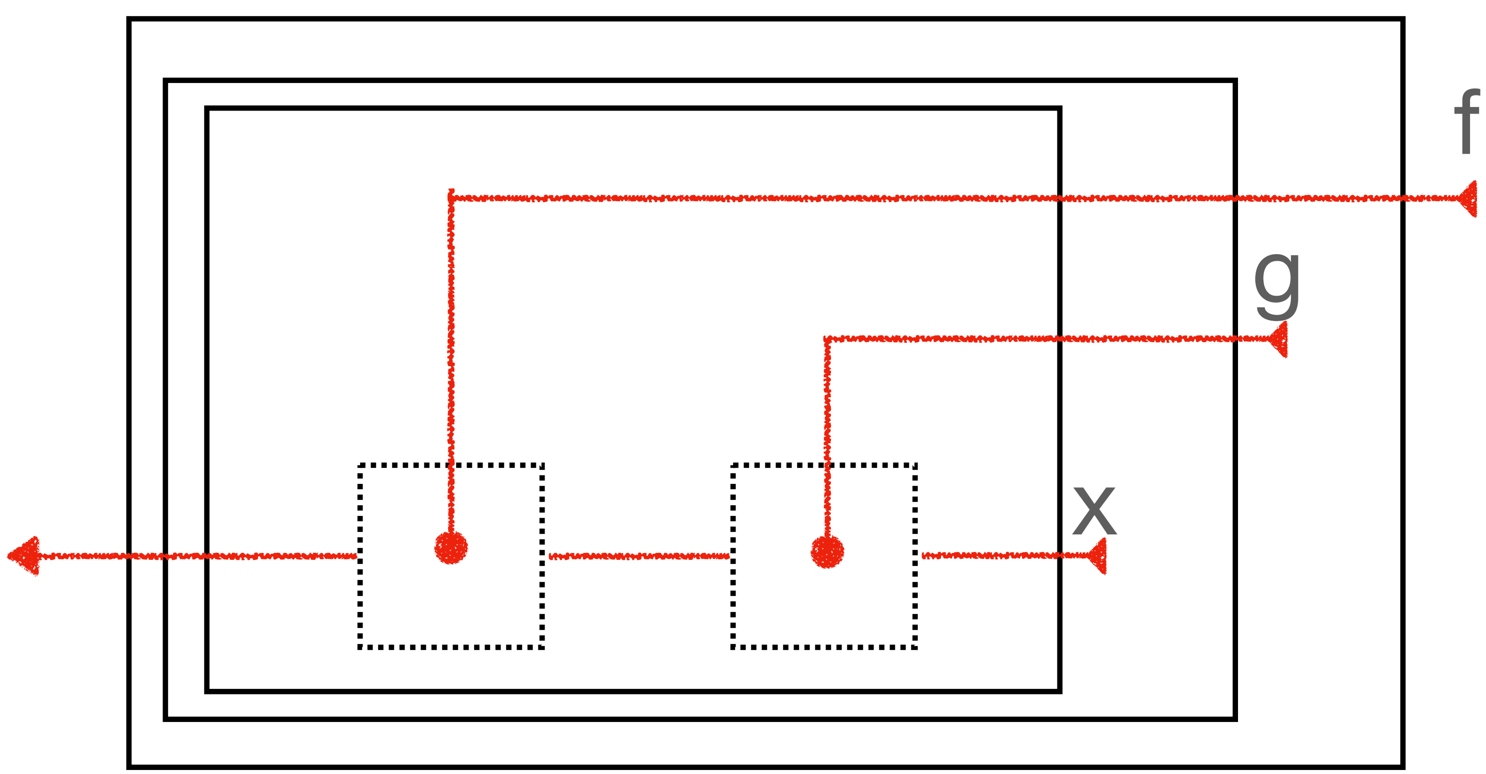
A Less Simple Example: *Compose*



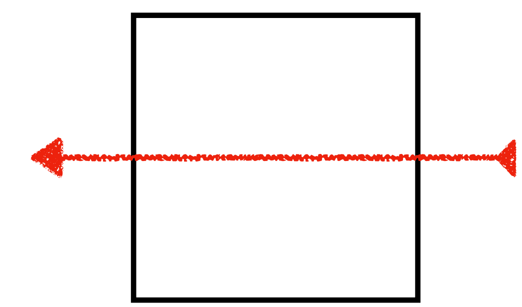
(λ (f)
  (λ (g)
    (λ (x) (f (g x)))))

v0 = (g x)
v1 = (f v0)

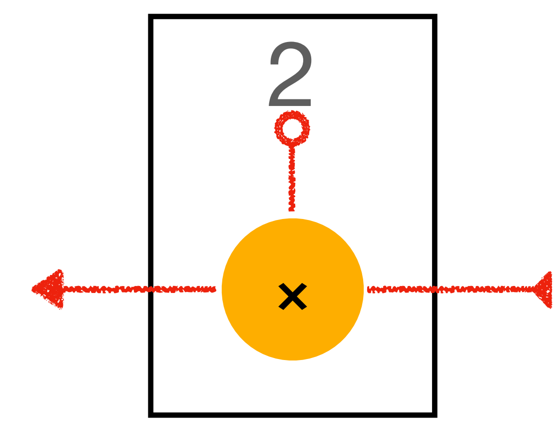
A Less Simple Example: *Compose*



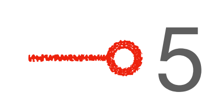
(((compose id) double) 5)



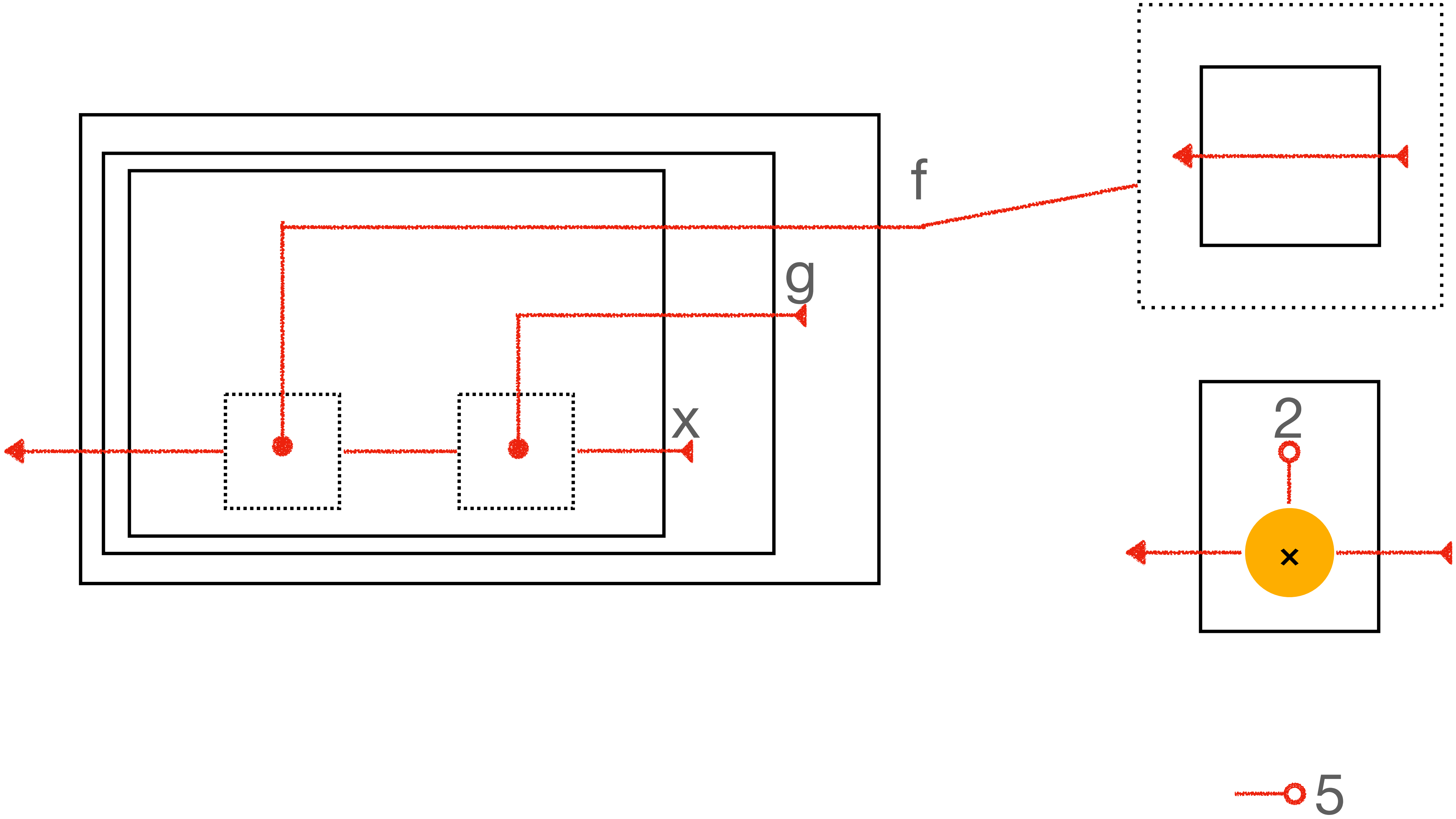
let id = (λ (x) x)

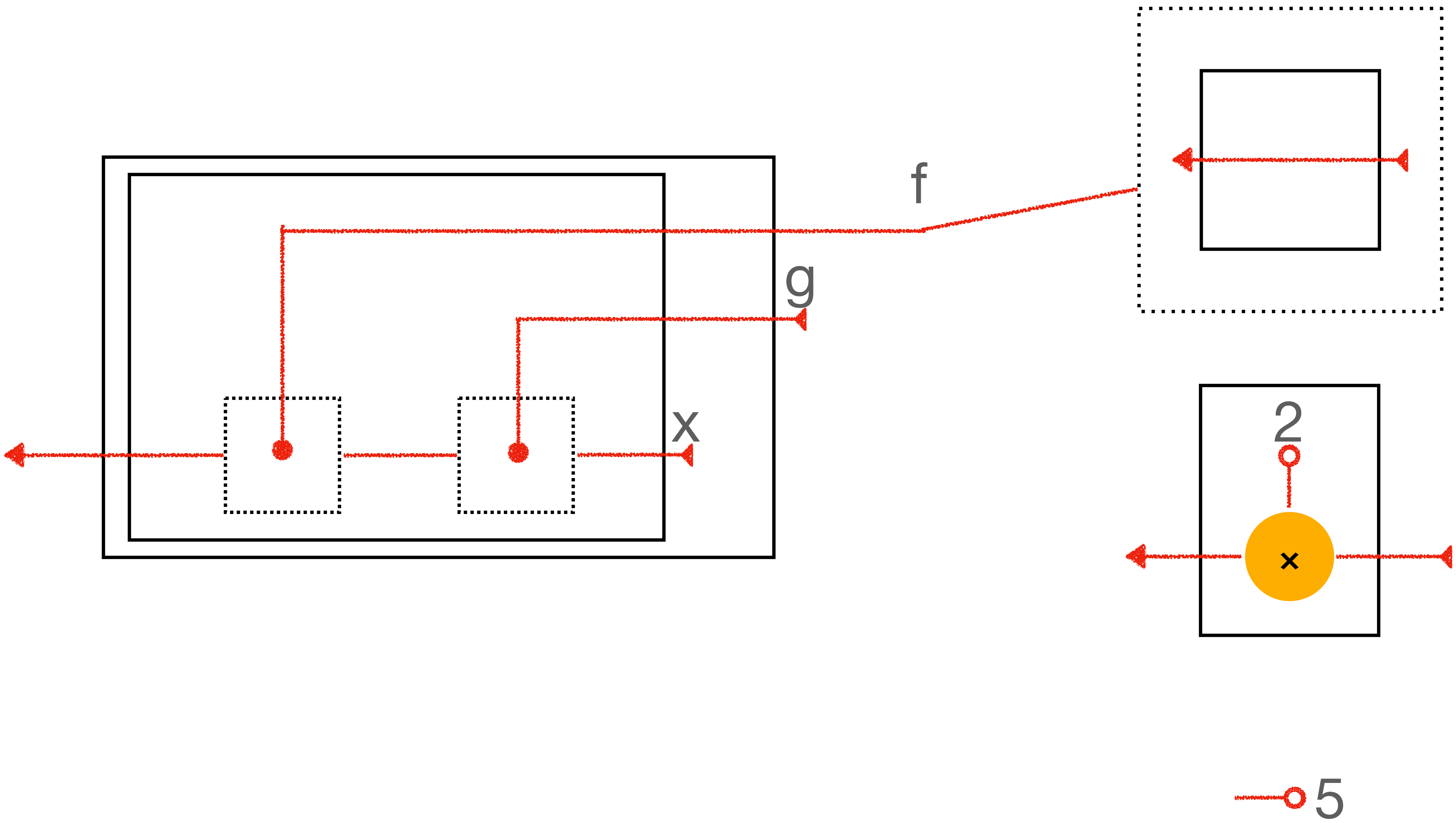


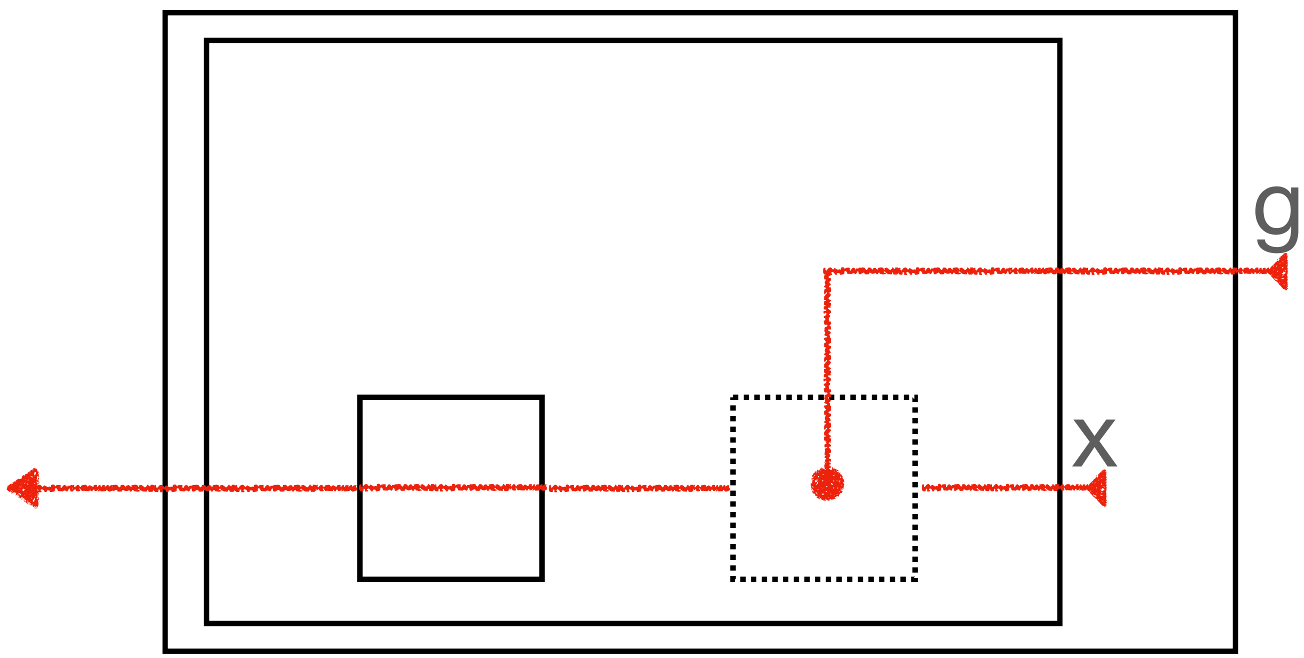
let double = (λ (x) (* 2 x))

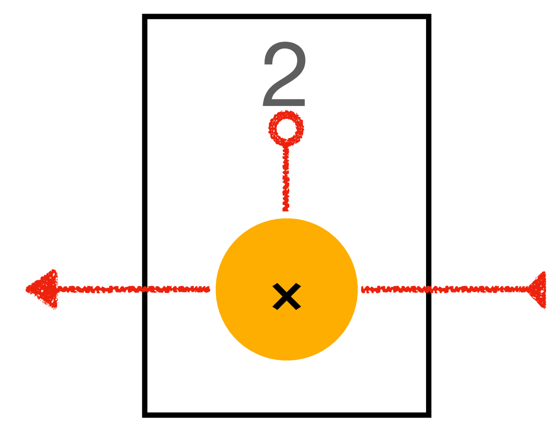


constant : 5

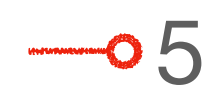

f
g
x
2
×
5

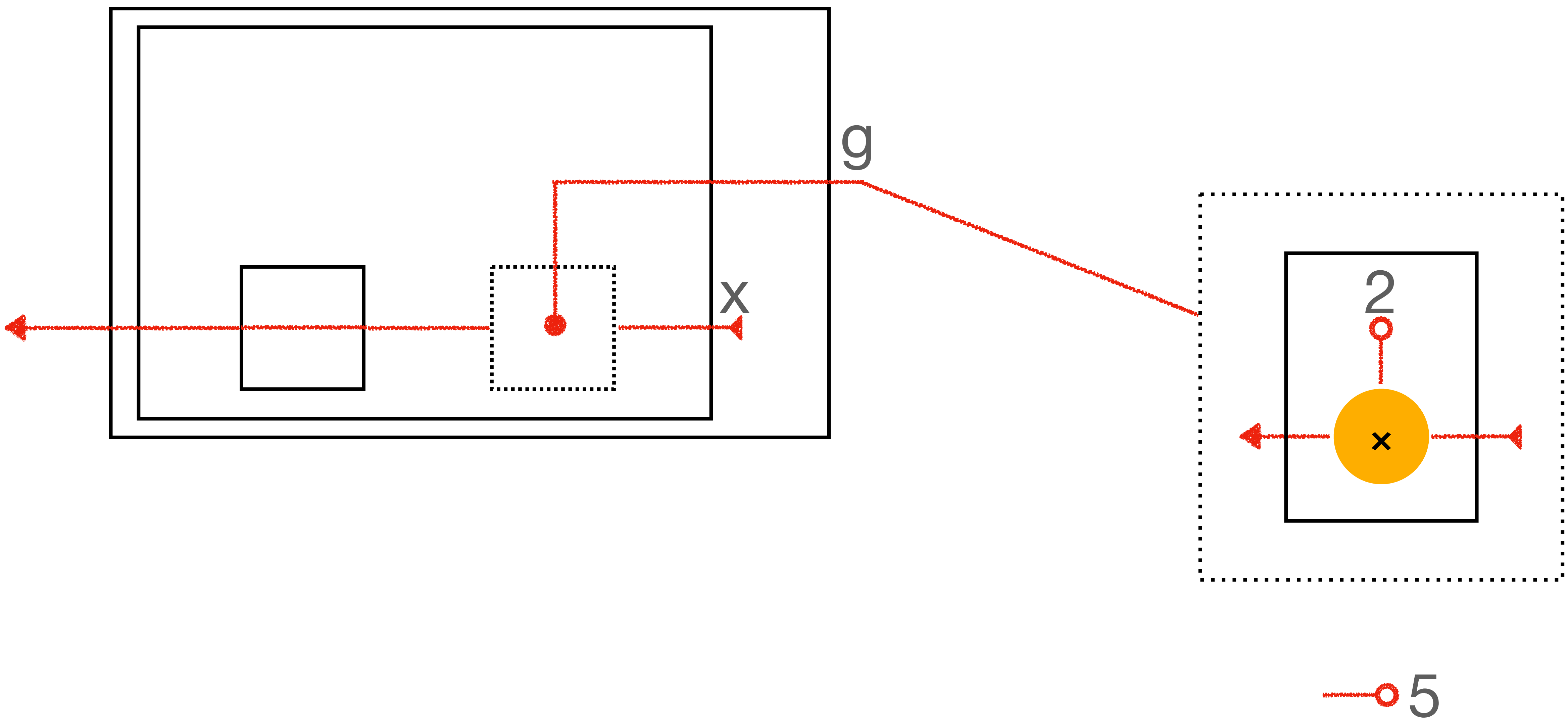

f
g
x
2
×
5



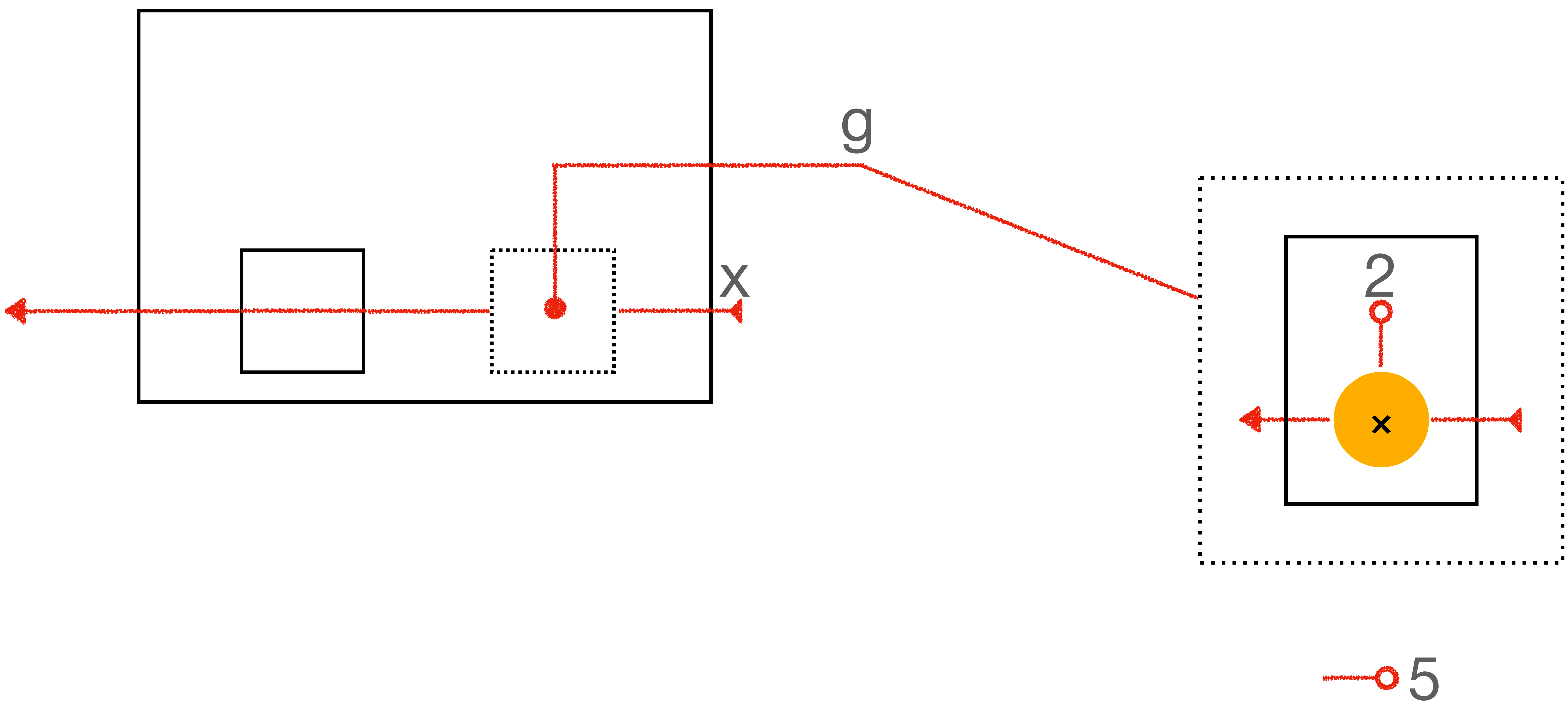


(λ (g) (λ (x) ((λ (x) x) (g x))))

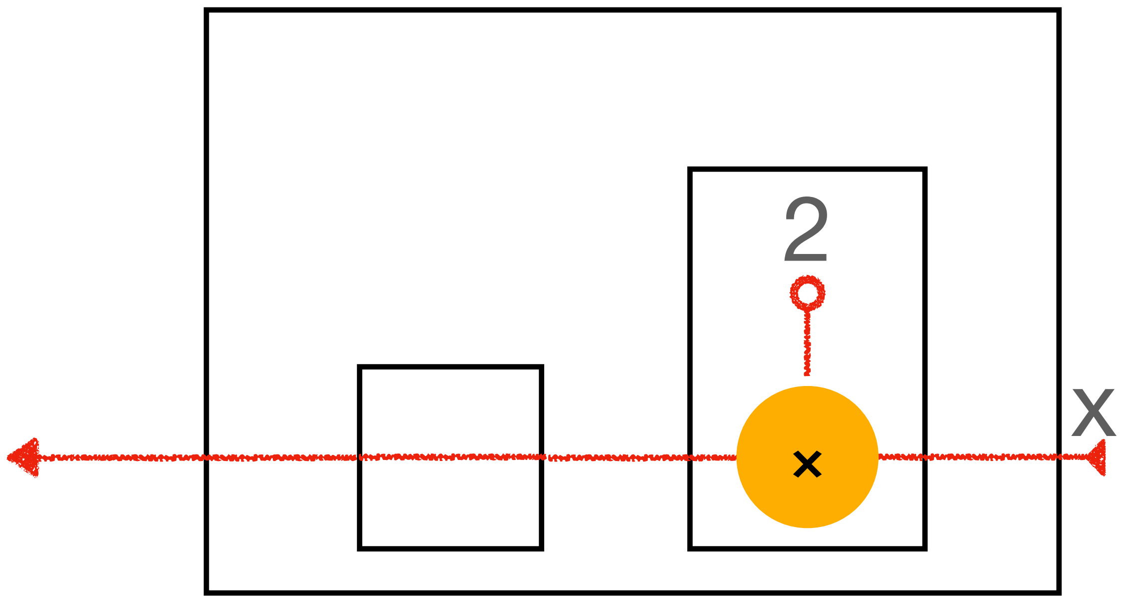




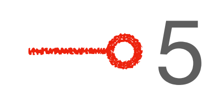
g
x
2
×
5

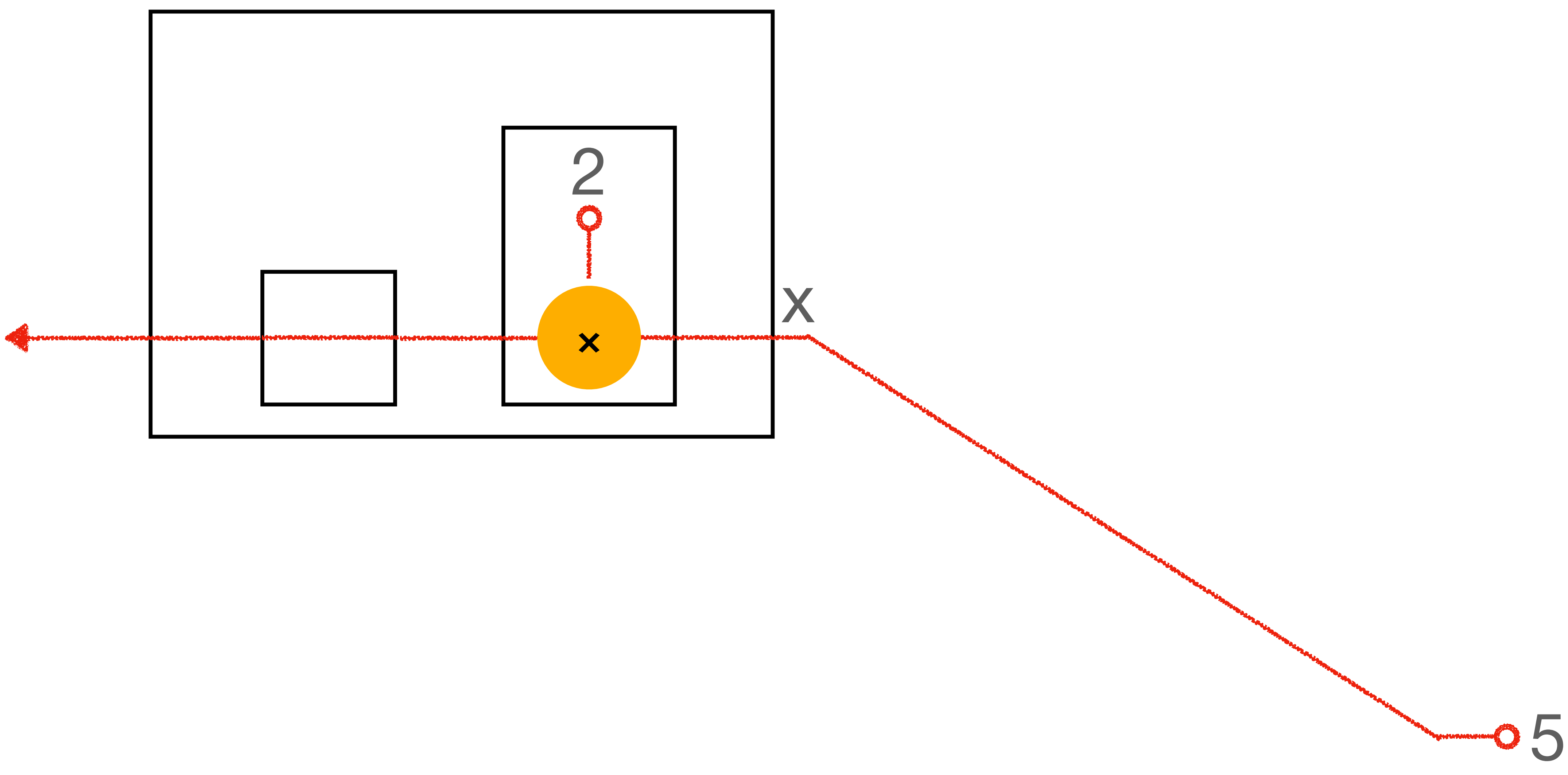


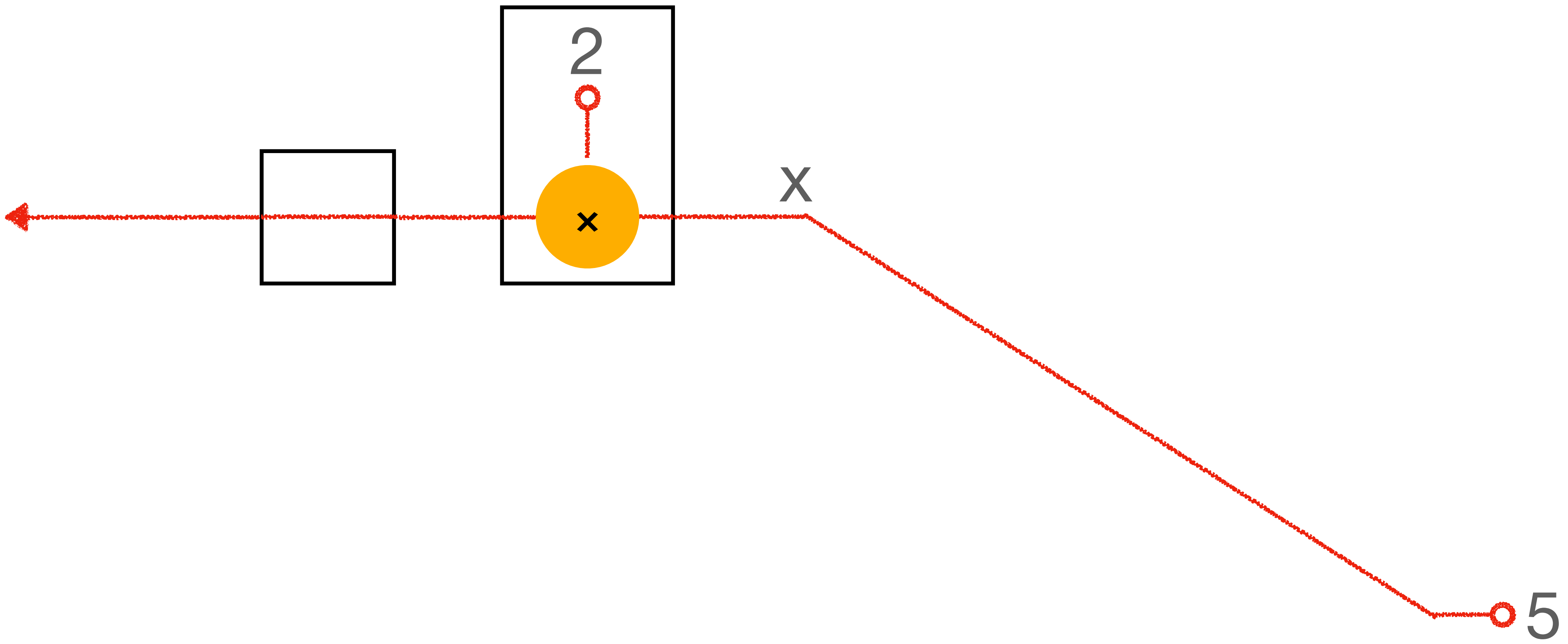
g
x
2
5

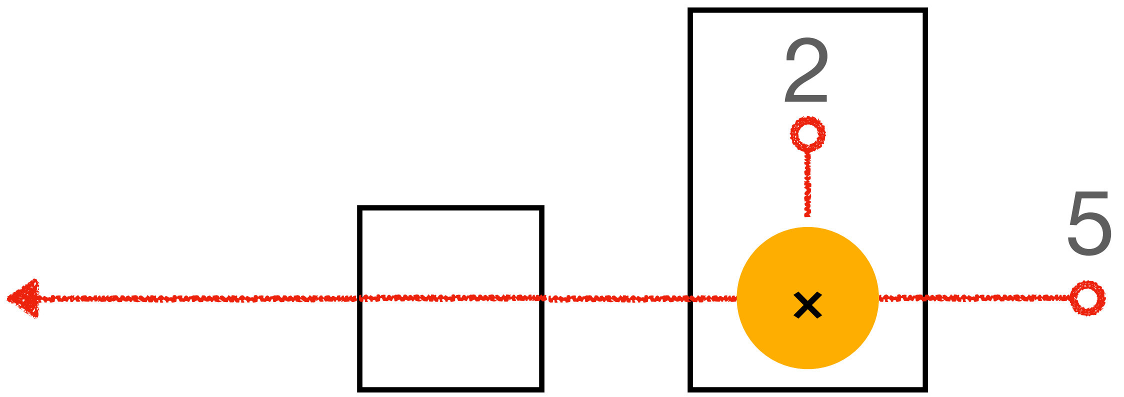


(λ (x) ((λ (x) x) ((λ (x) (* 2 x)) x)))

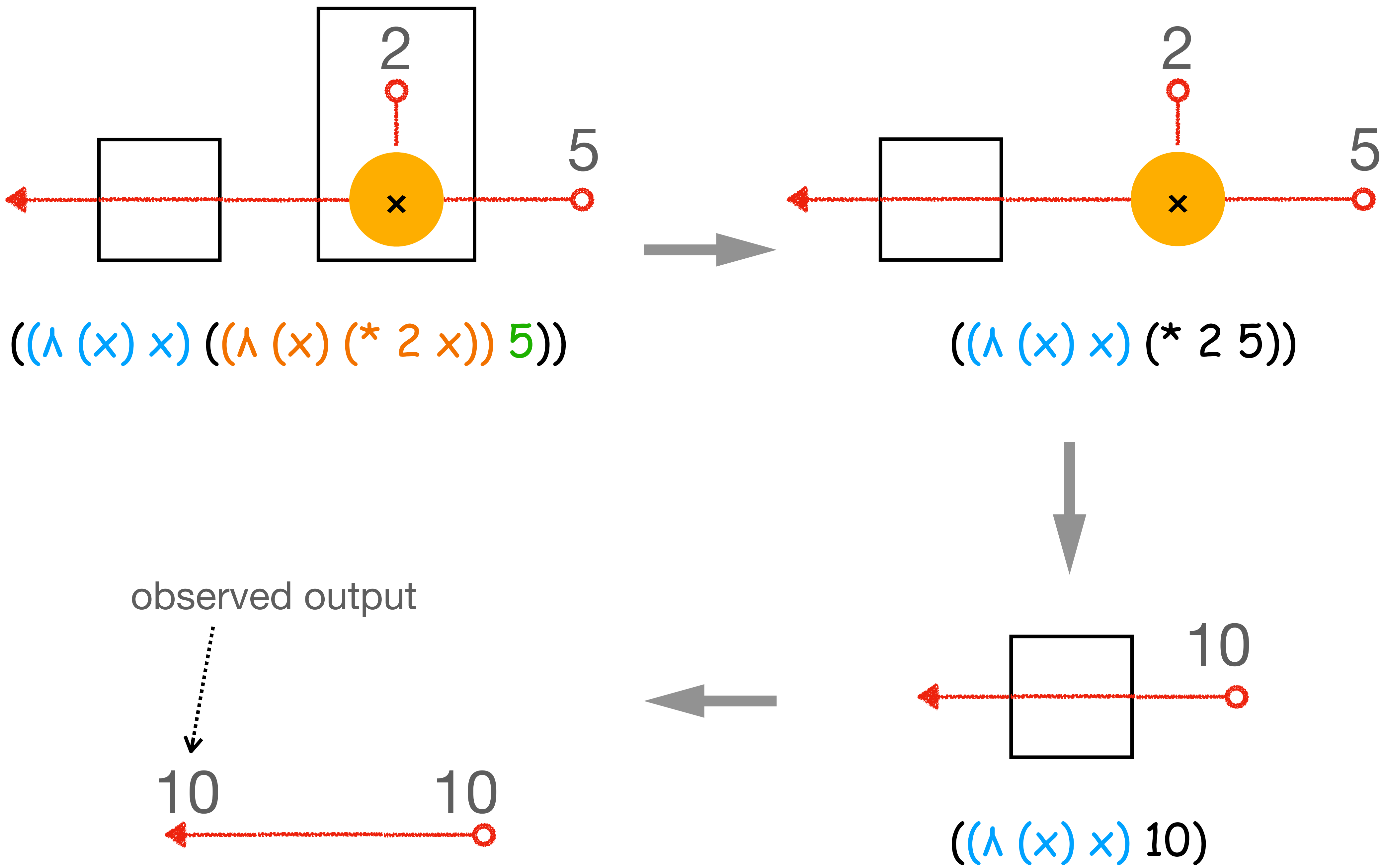



2
x
5

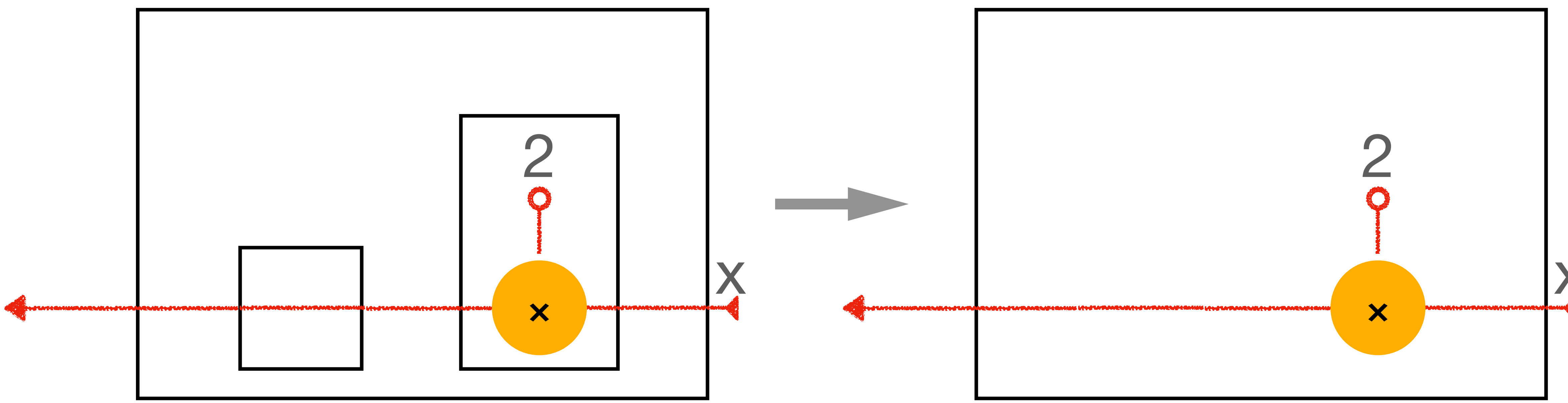

2
×
x
5



((λ (x) x) ((λ (x) (* 2 x)) 5))


2
5
((λ (x) x) ((λ (x) (* 2 x)) 5))
2
5
((λ (x) x) (* 2 5))
10
((λ (x) x) 10)
observed output
10
10

“Circuits Simplification”

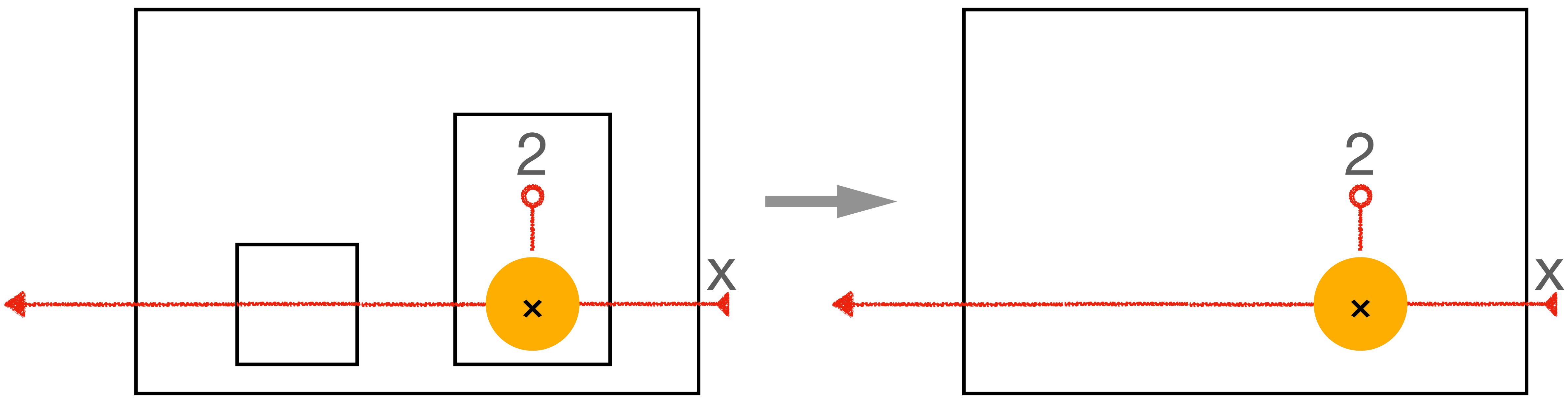


(λ (x)
((λ (x) x) ((λ (x) (* 2 x)) x)))))

→

(λ (x) (* 2 x)))))

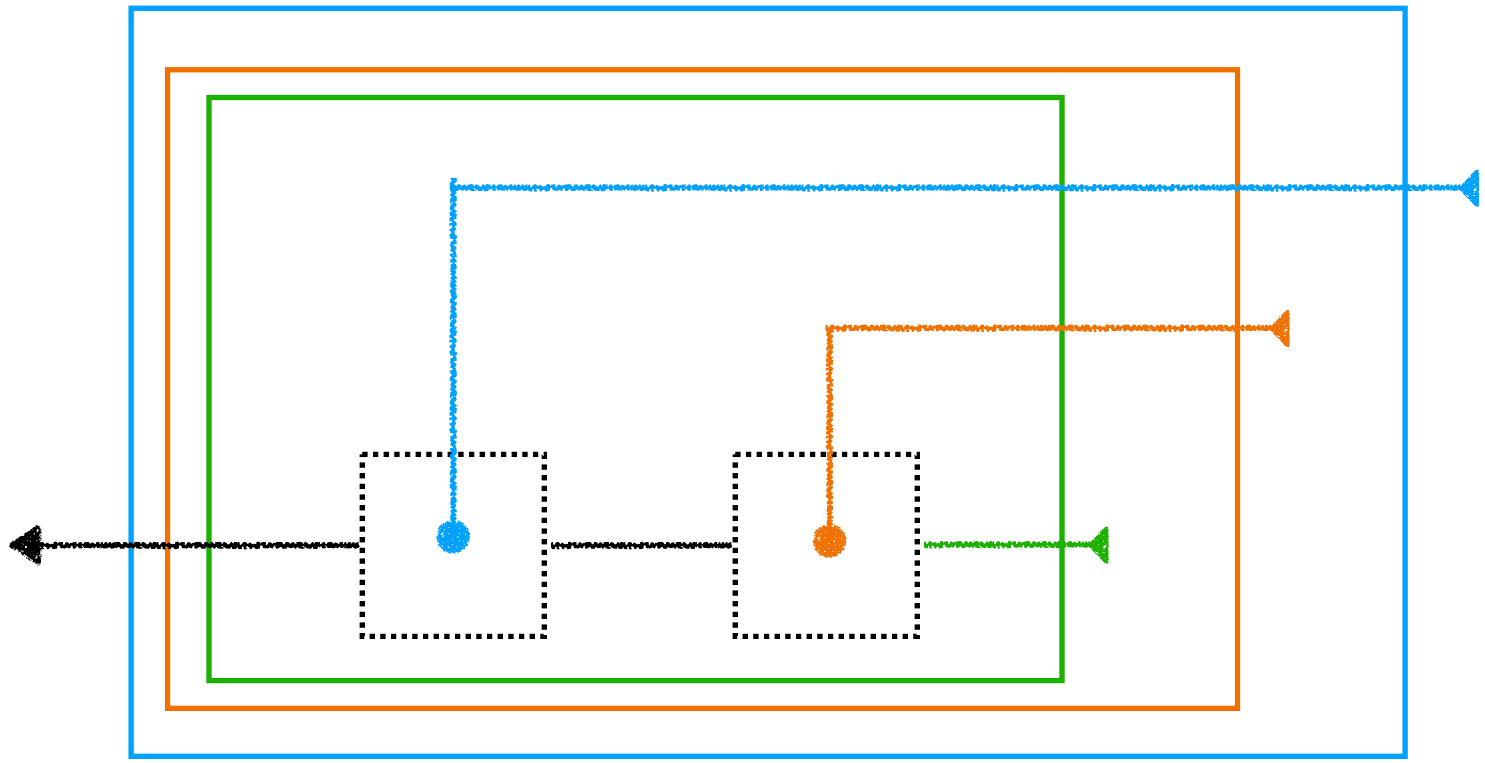
Partial Evaluation & “Supercompilation”




```
(λ (x)
  ((λ (x) x) ((λ (x) (* 2 x)) x)))))
```

```
(λ (x) (* 2 x)))))
```

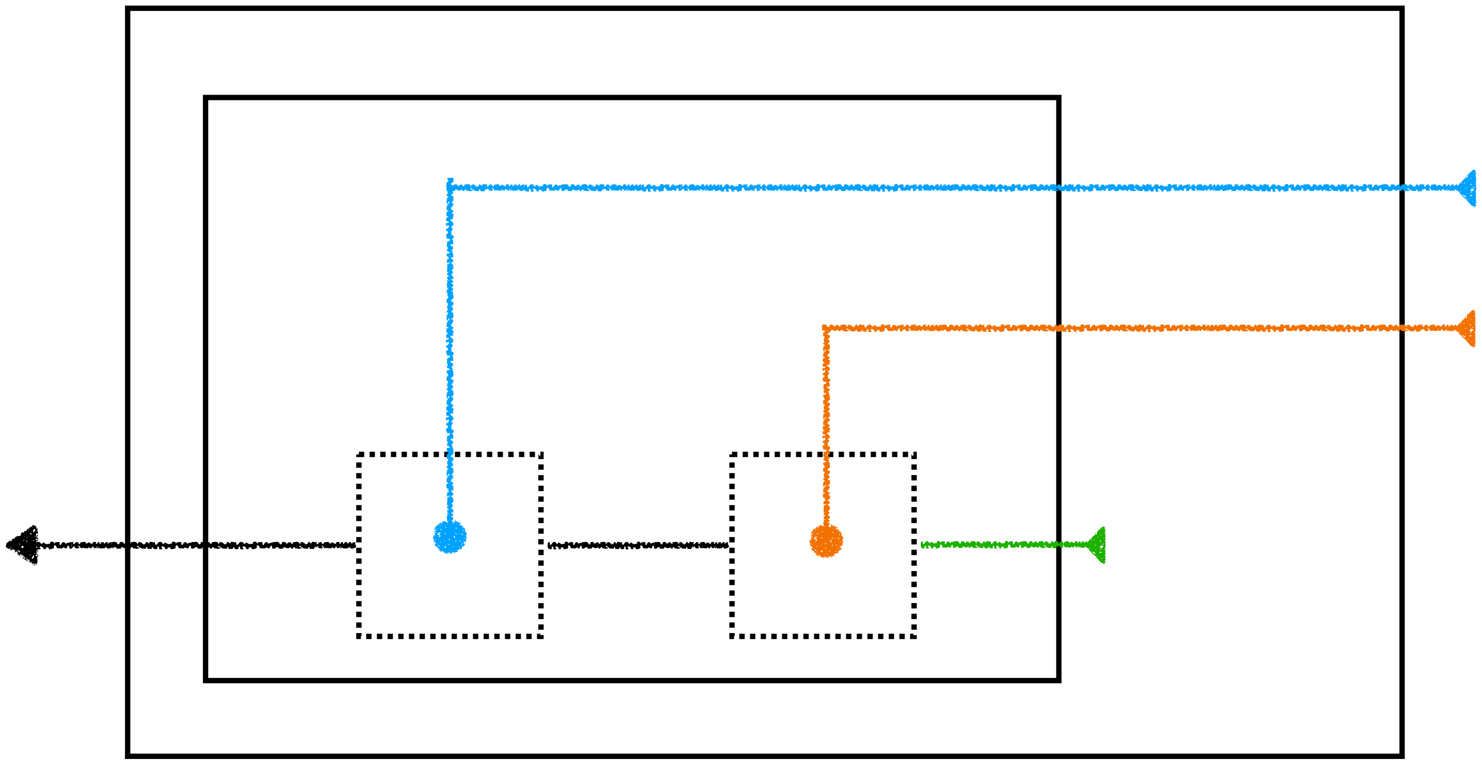
de Bruijn Numbers ~ ?



compose = (λ (λ (λ (3 (2 1)))))

implicit “**λ**”, but explicit “binding structures”

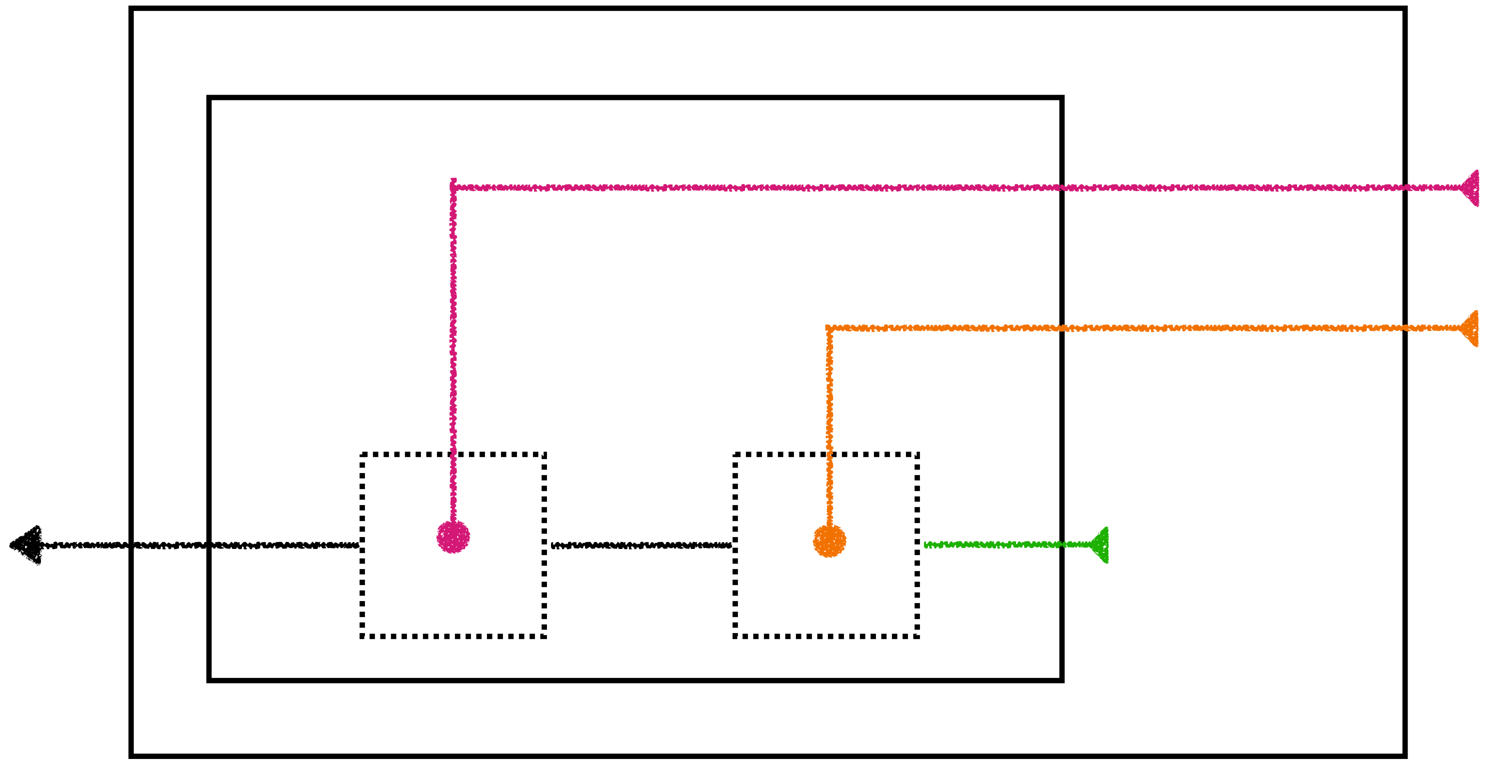
Beyond de Bruijn Numbers?



compose = (λ (f g) (λ (x) (f (g x))))

wires are perfect binders in nature!

Beyond de Bruijn Numbers?

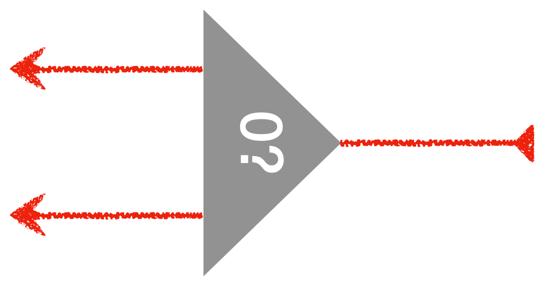


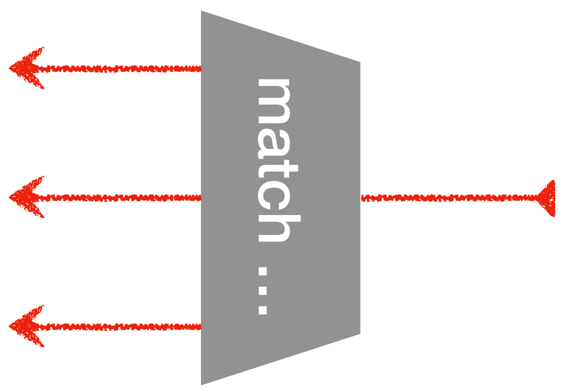
compose = (λ (h g) (λ (x) (h (g x))))

α-equivalence: the name of a wire should always be **consistent**

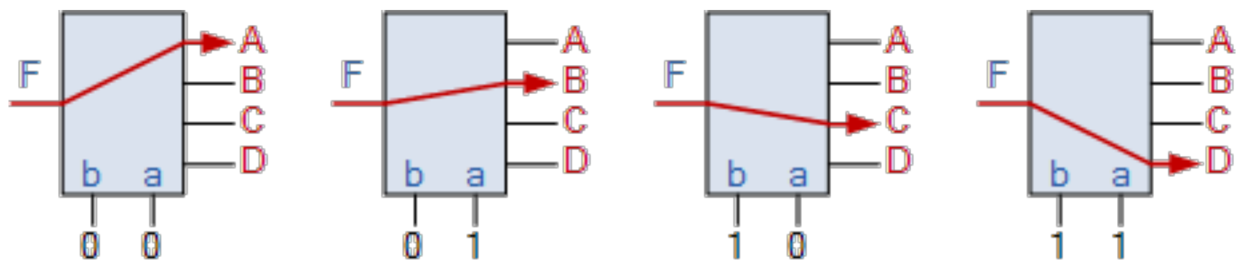
Part II: Language Structures

Conditionals ~ Demultiplexers



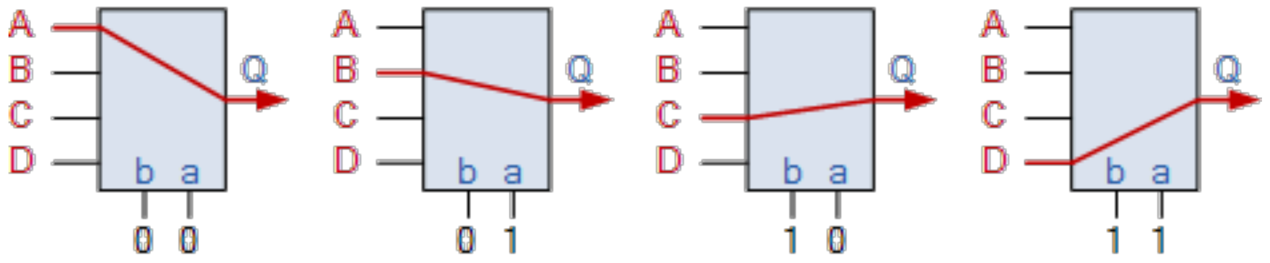


DEMUX

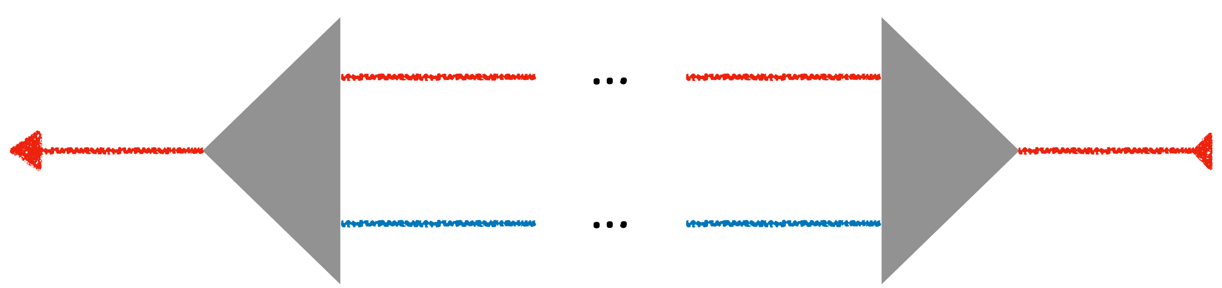


? ~ Multiplexers

Multiplexers are implicit in programming languages…

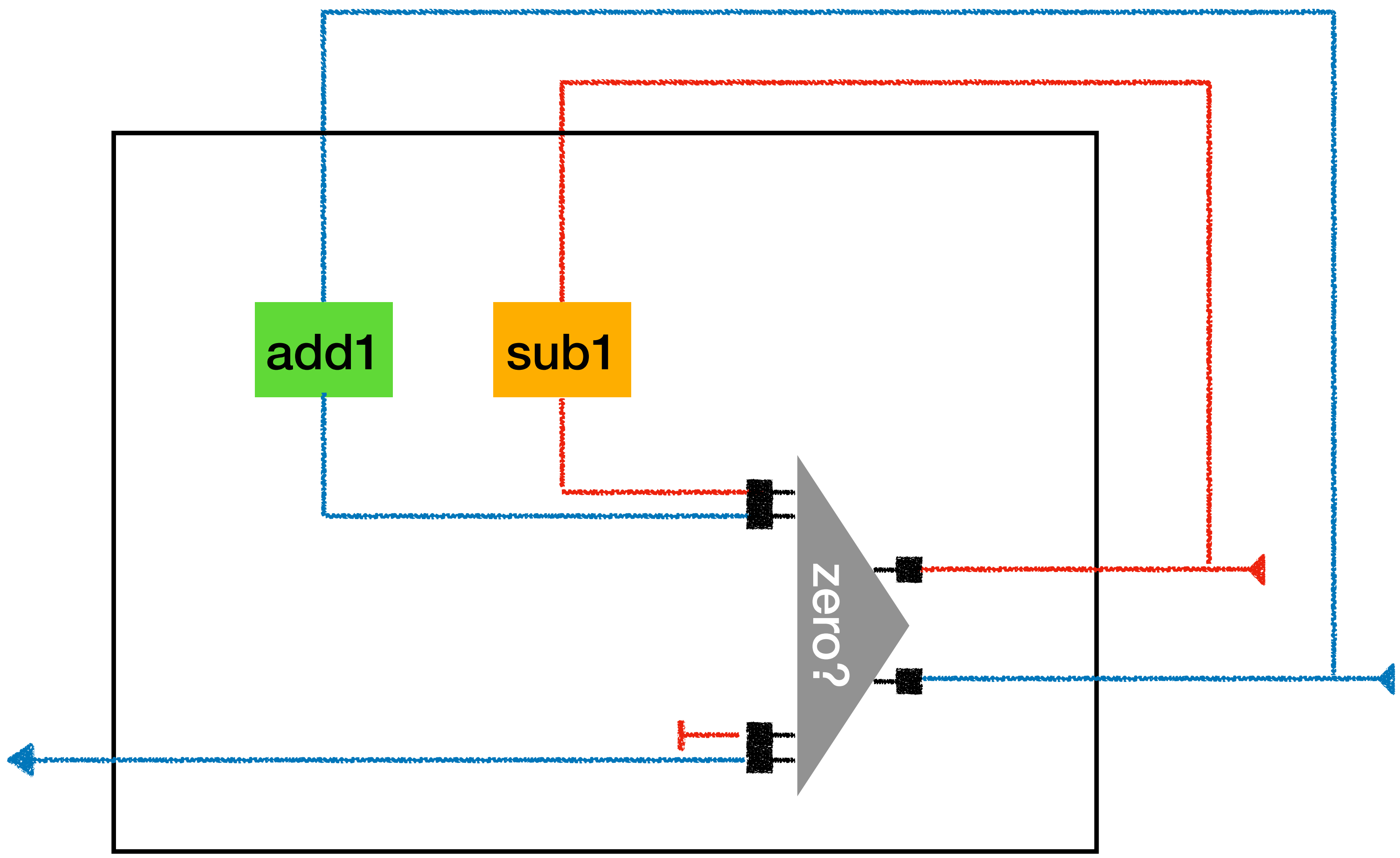



```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- 0 x)]))
```



“JOIN” in static analysis

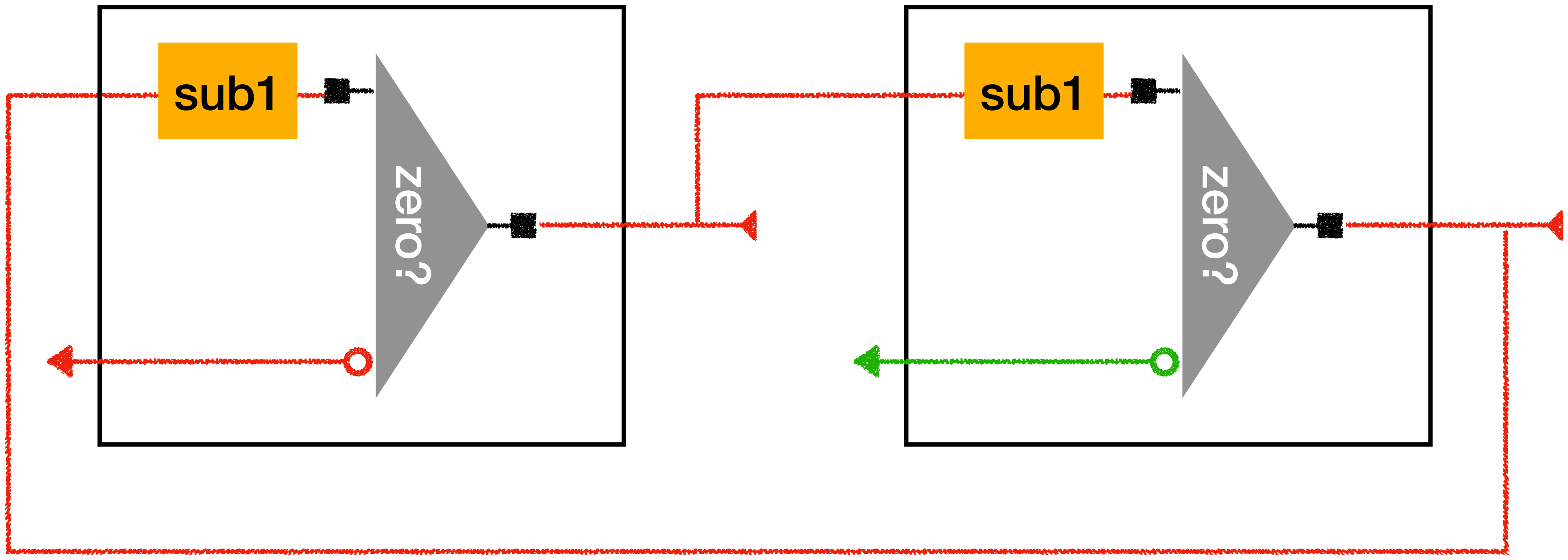
Recursive Functions ~ Dynamic Circuits




```
(define (add a b)
  (cond
    [(zero? a) b]
    [else (add (sub1 a) (add1 b))]))
```

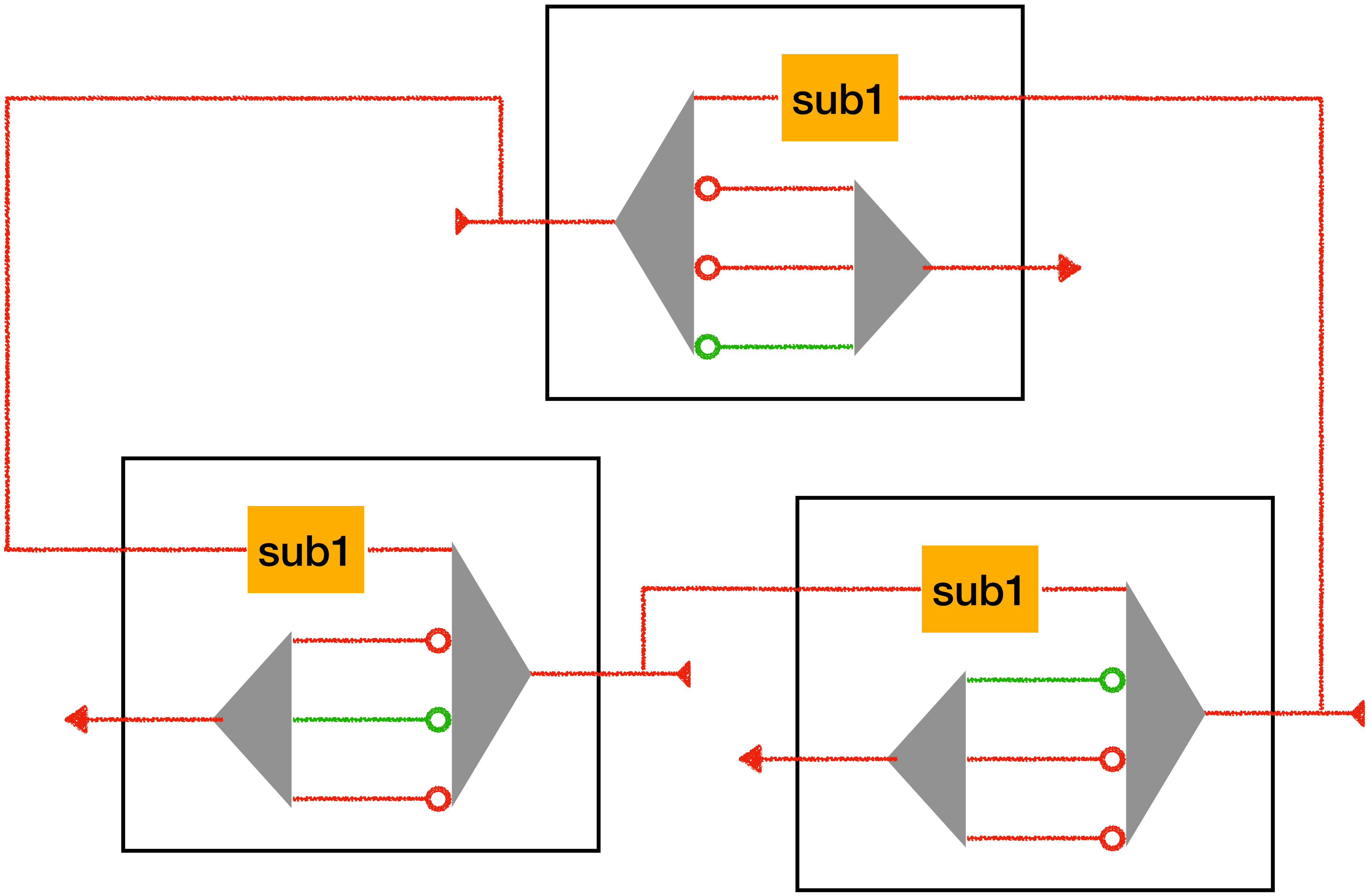
tail recursion (loop)

Recursive Functions ~ Dynamic Circuits

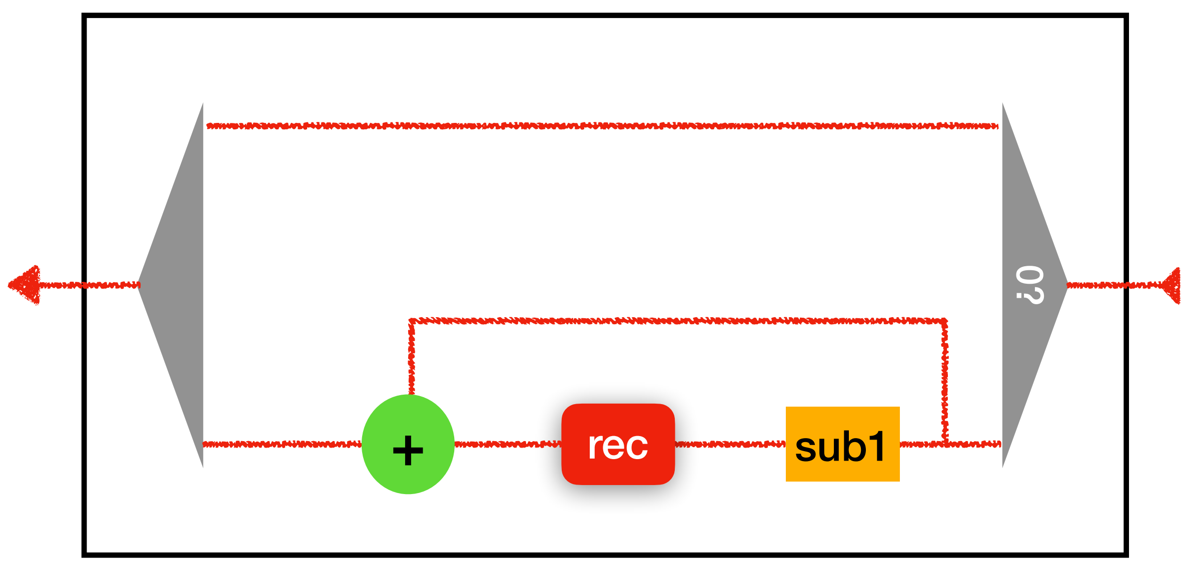


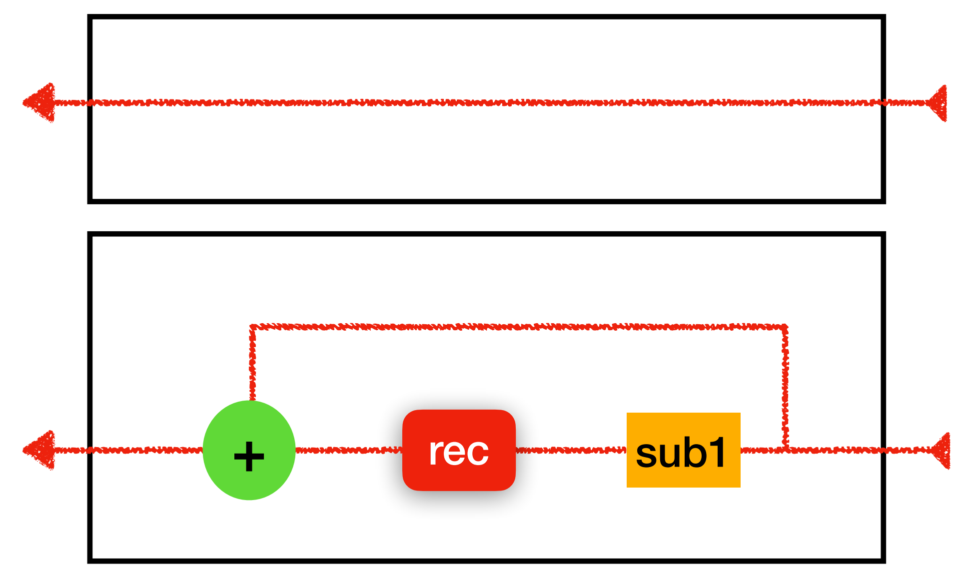
mutual recursion

Recursive Functions ~ Dynamic Circuits



Recursive Functions ~ Dynamic Circuits

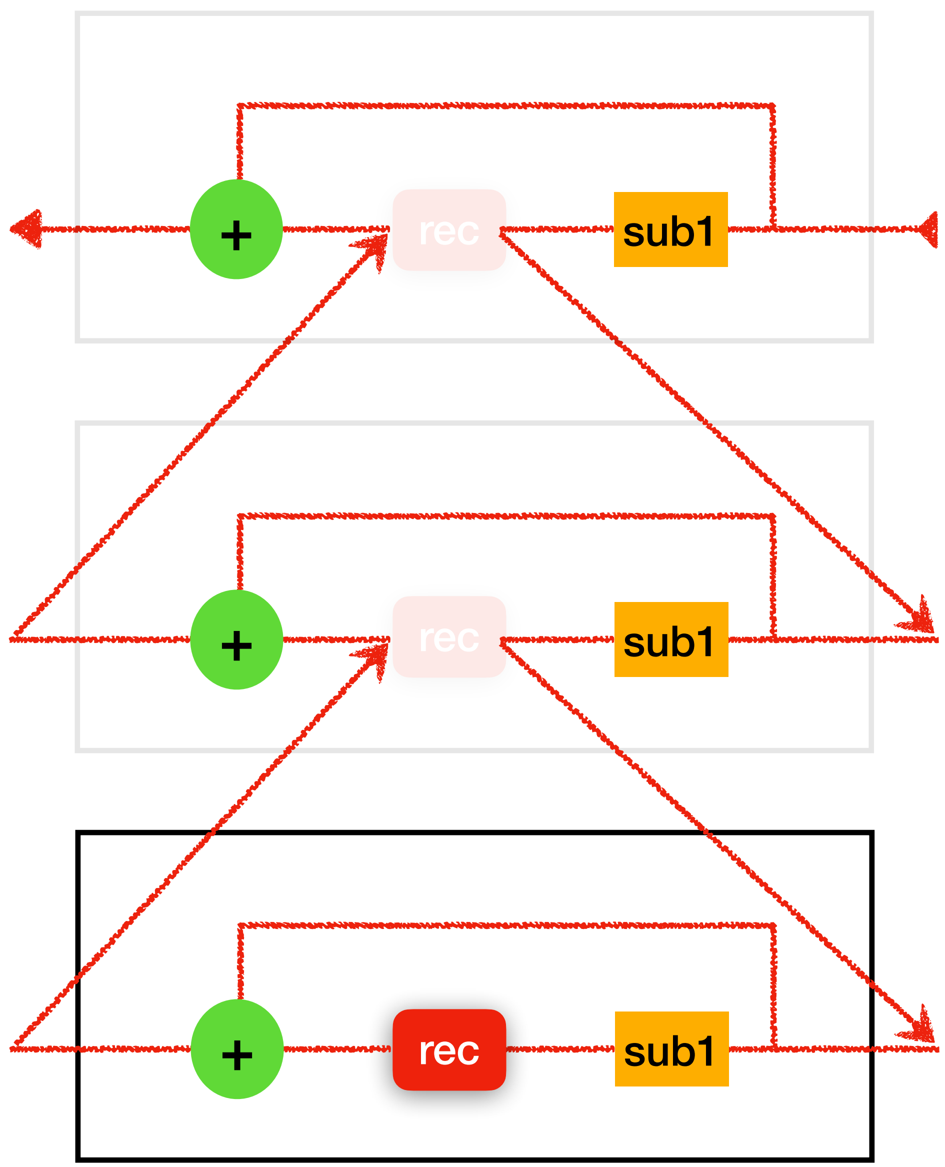


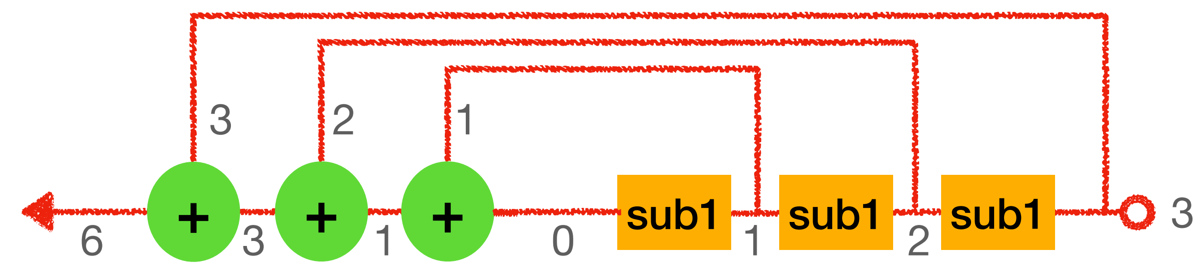



```
sum n = case n of
          0 -> 0
          suc x -> (sum x) + (suc x)
```

```
sum 0 = 0
sum (suc x) = (sum x) + (suc x)
```

Recursive Functions ~ Dynamic Circuits





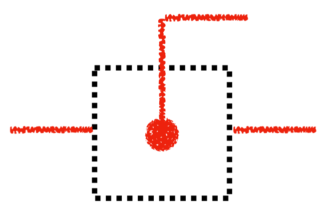
sum 3

```
sum 0 = 0
sum (suc x) = (sum x) + (suc x)
```

“dynamically unfolding”

stack (in a logical sense)

“Real Life” Circuits

Nothing like 

No “higher-order” circuits

Can’t duplicate/unfold circuits at will

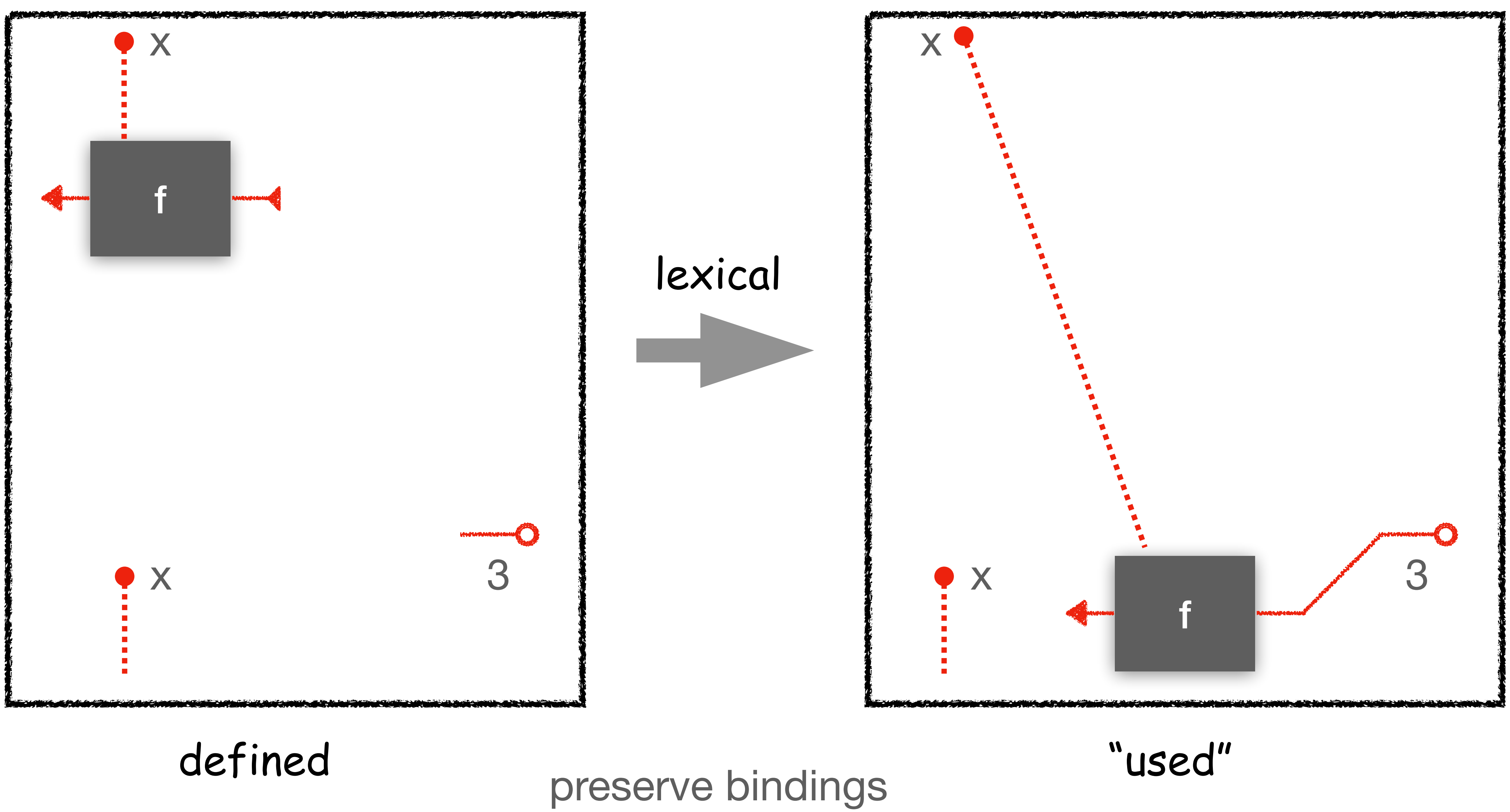
Circuits are **physical resources**
(Related: Linear Logic)

- λ-circuits are “recipes” for building more “physical” circuits
- They can be duplicated and thrown away **logically**
- Copies can be reduced, while recipes remain intact

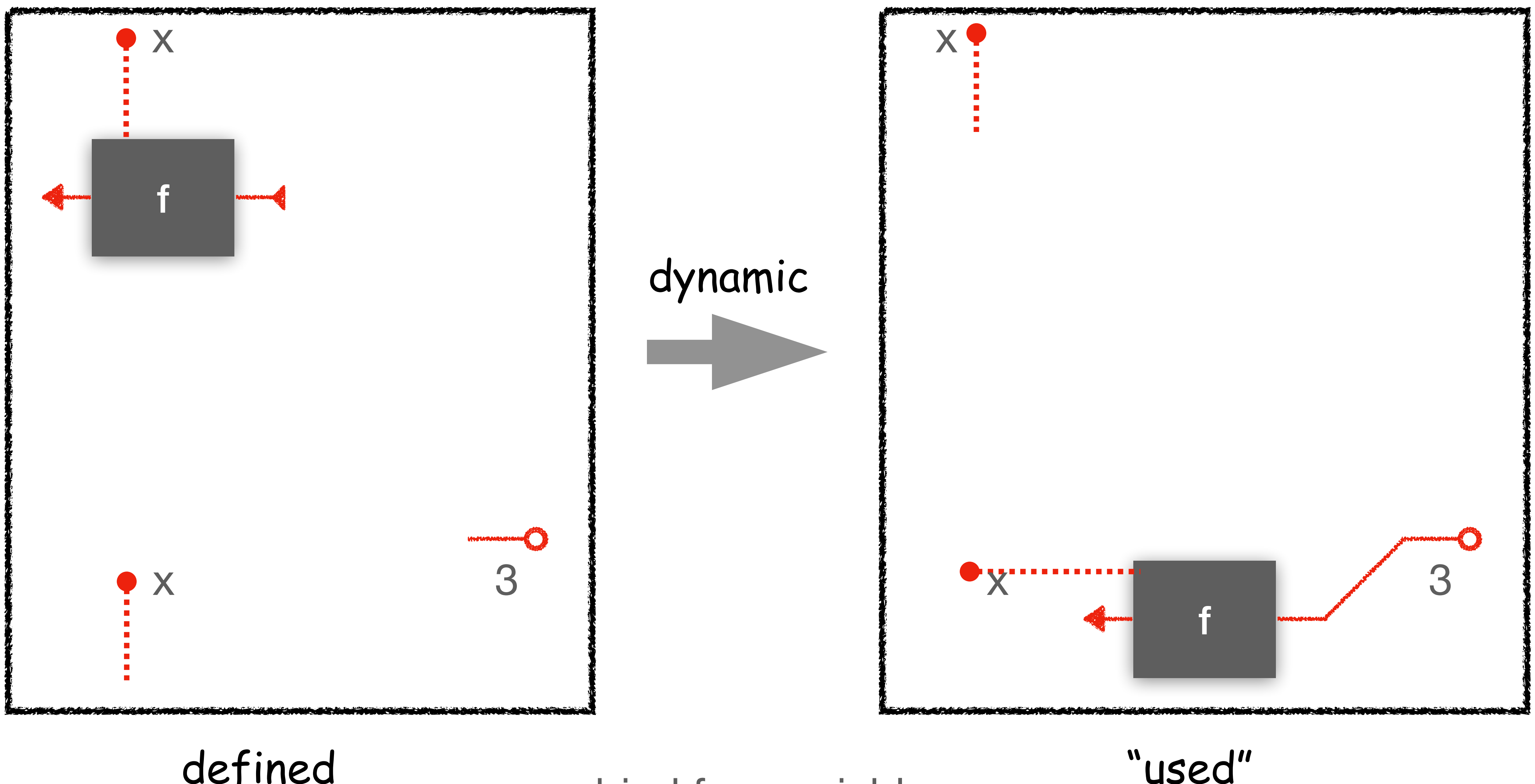
Dynamic Scoping vs. Lexical Scoping

```
(let ([x 2])
  (let ([f (λ (y) (* y x))])     ;; where f is defined
    (let ([x 4])
      (f 3))))                    ;; where f is "used"
```

Dynamic Scoping vs. Lexical Scoping

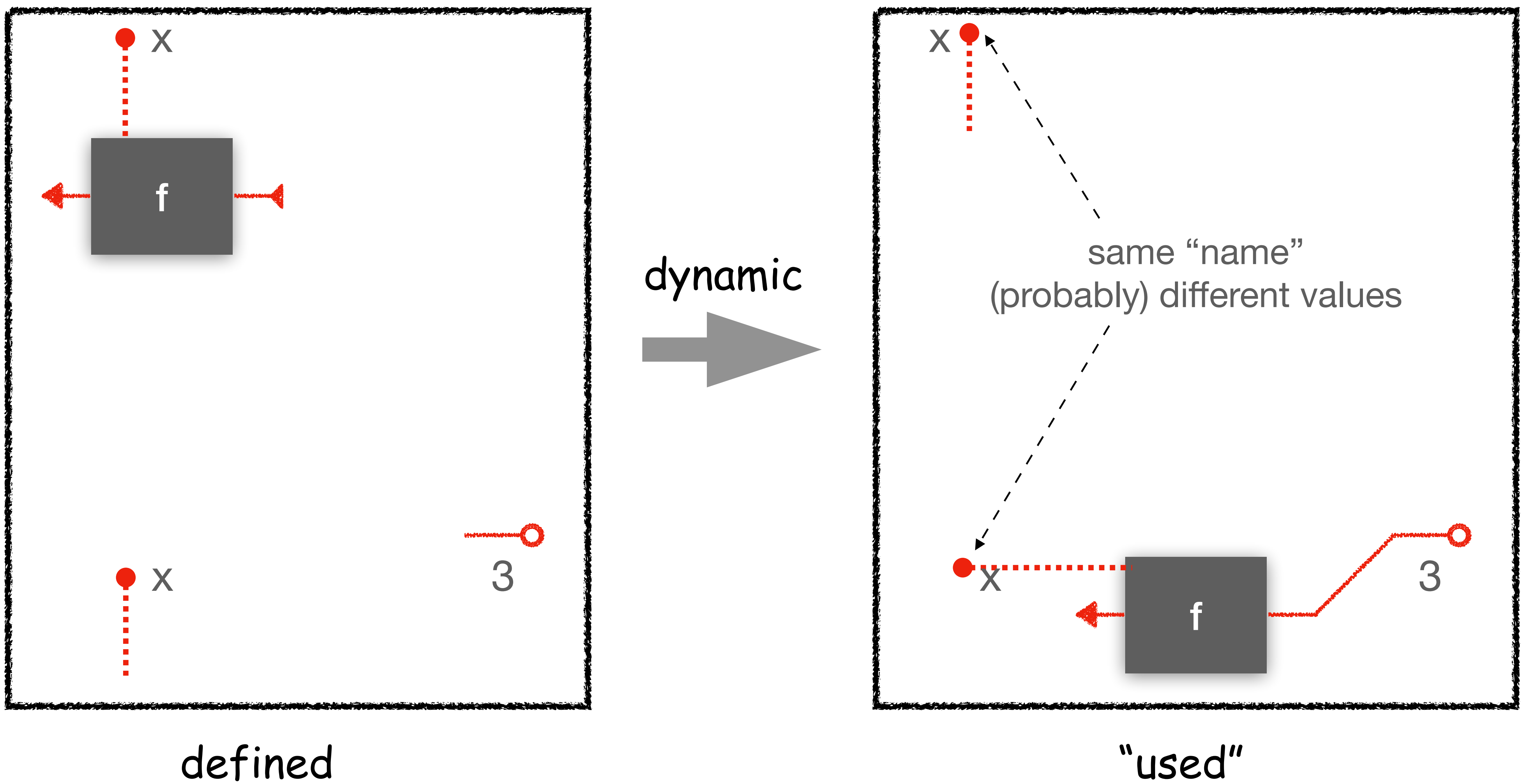


Dynamic Scoping vs. Lexical Scoping



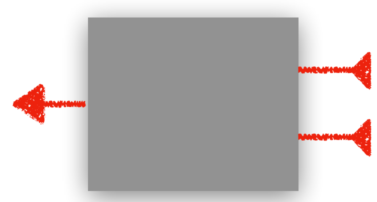
rebind free variables

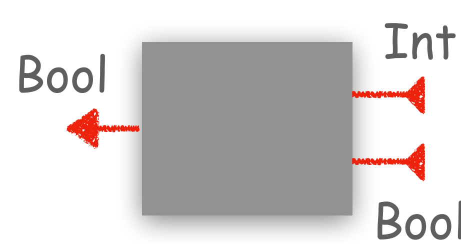
Dynamic Scoping vs. Lexical Scoping



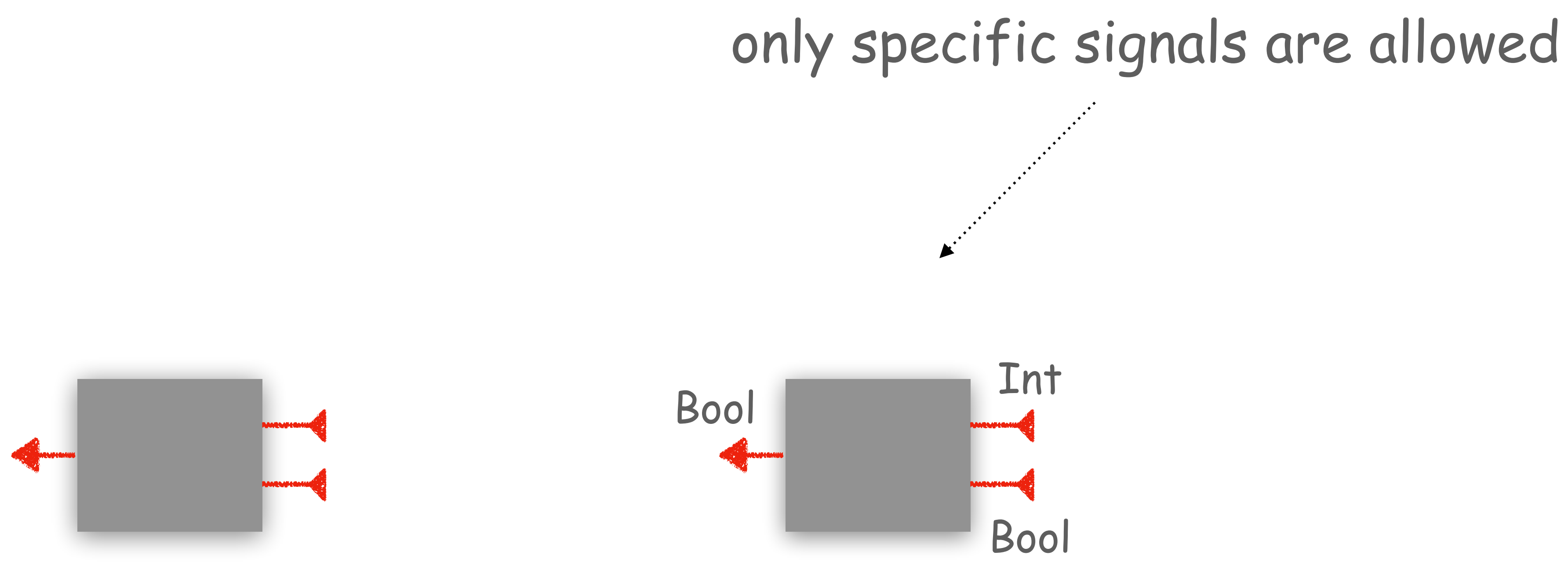
Part III: Types & Type Inference

What are types?

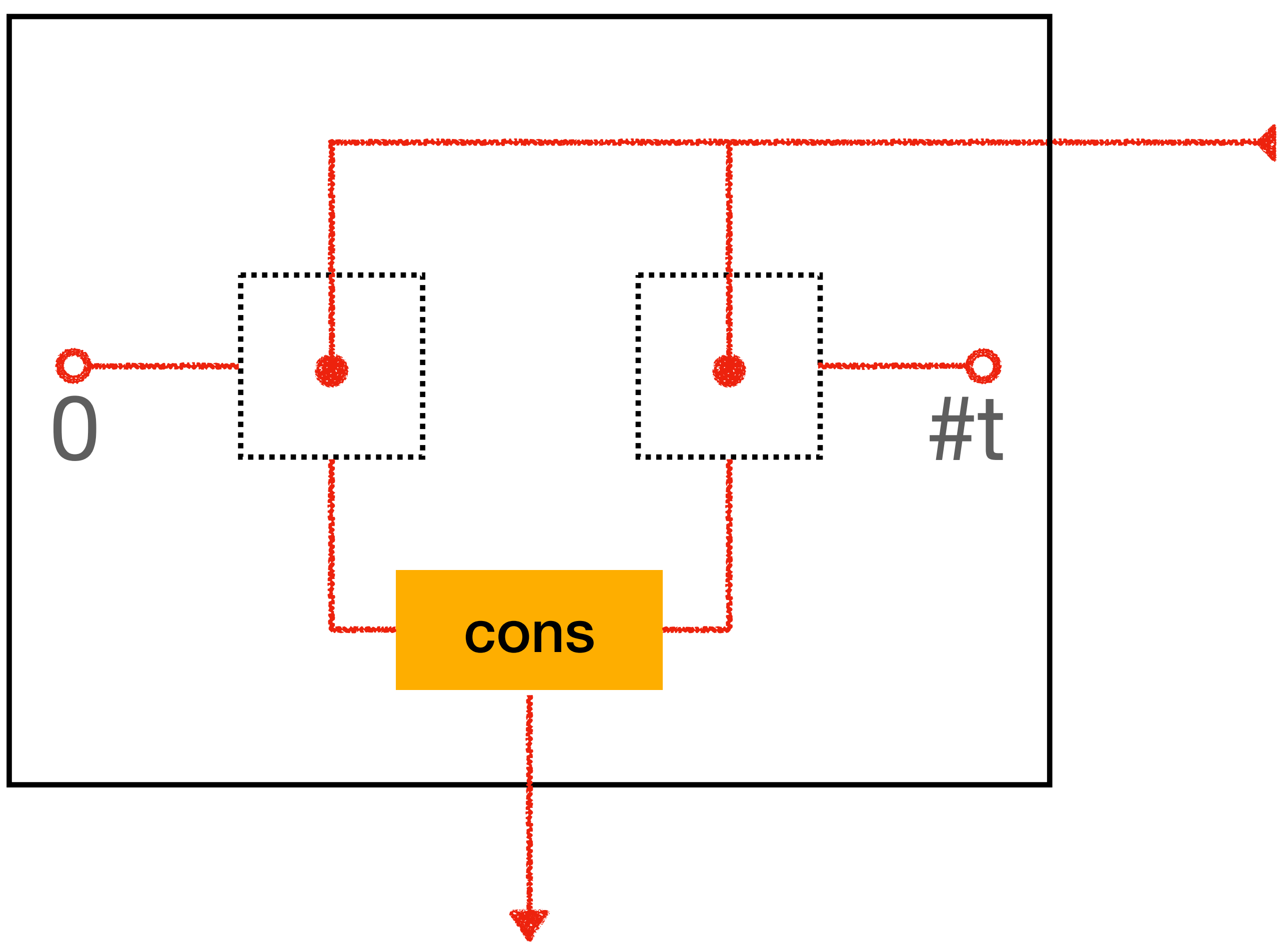




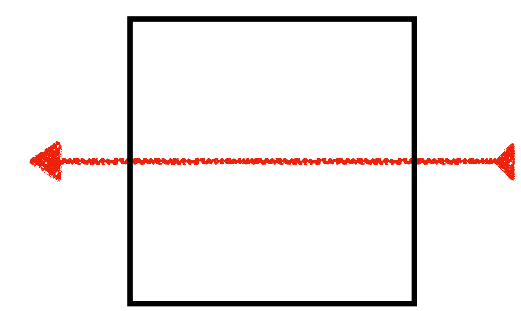
Types ~ "Interface Specifications"



Type Inference Example



foo = (λ (f) (cons (f 0) (f #t)))



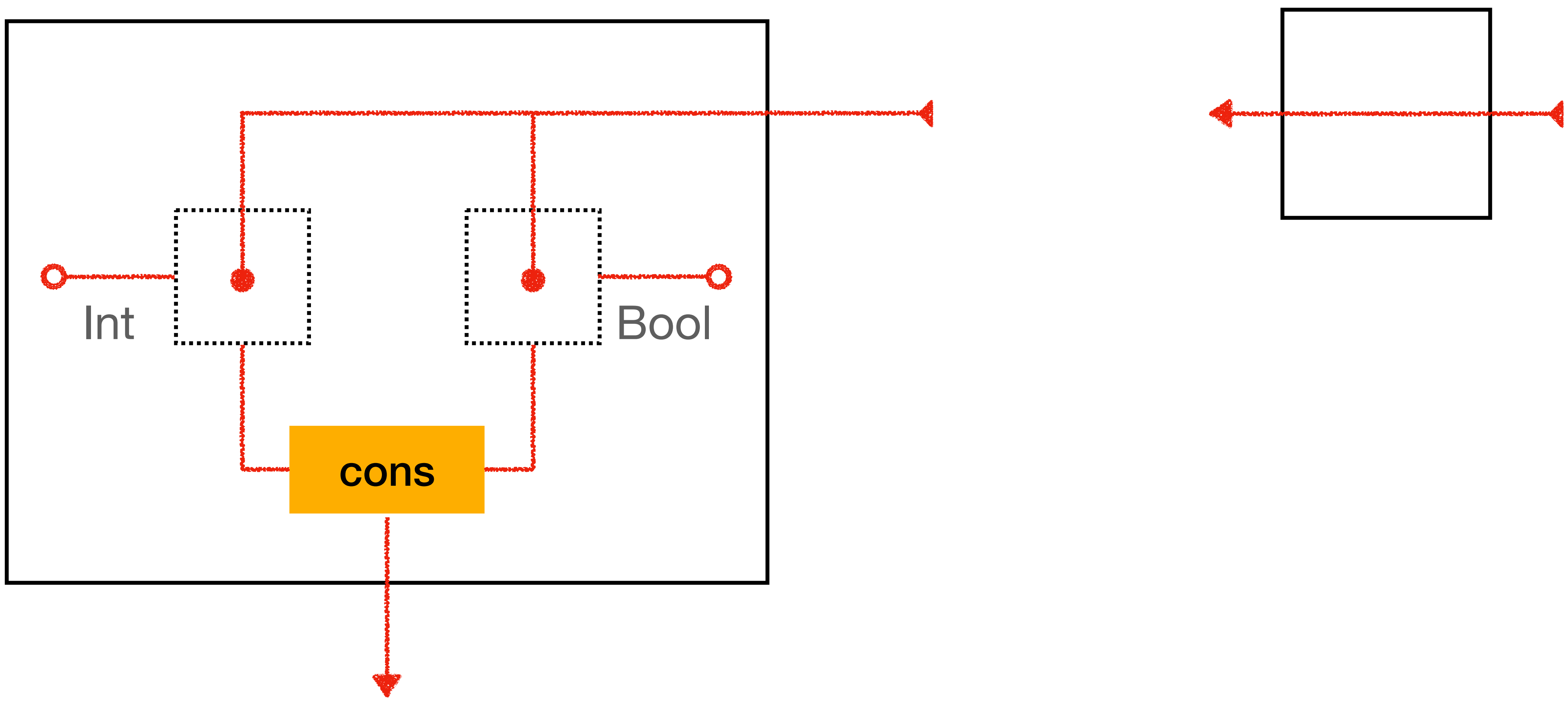
id = (λ (x) x)

(foo id)

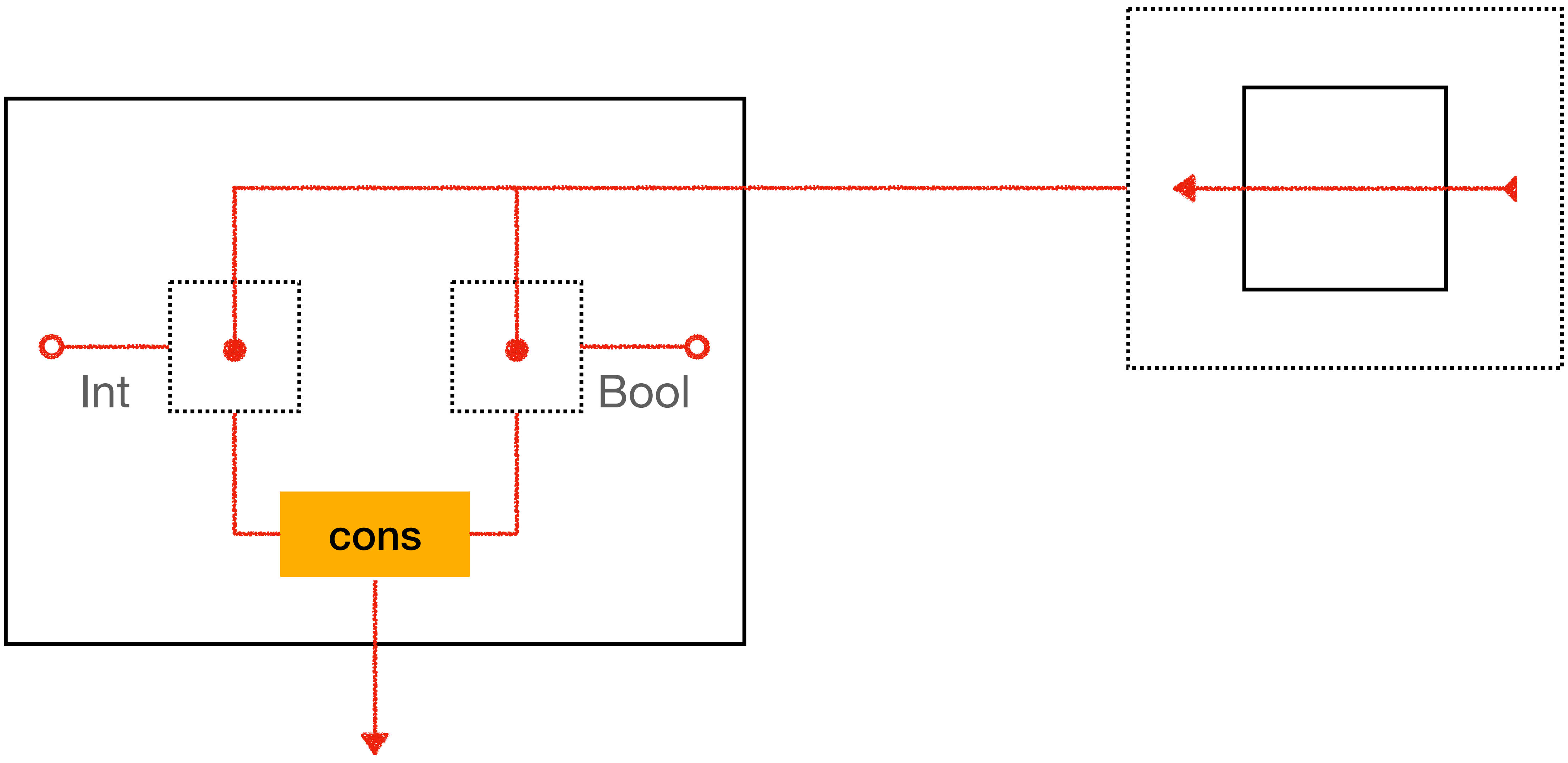
type(foo) = ?

type(id) = ?

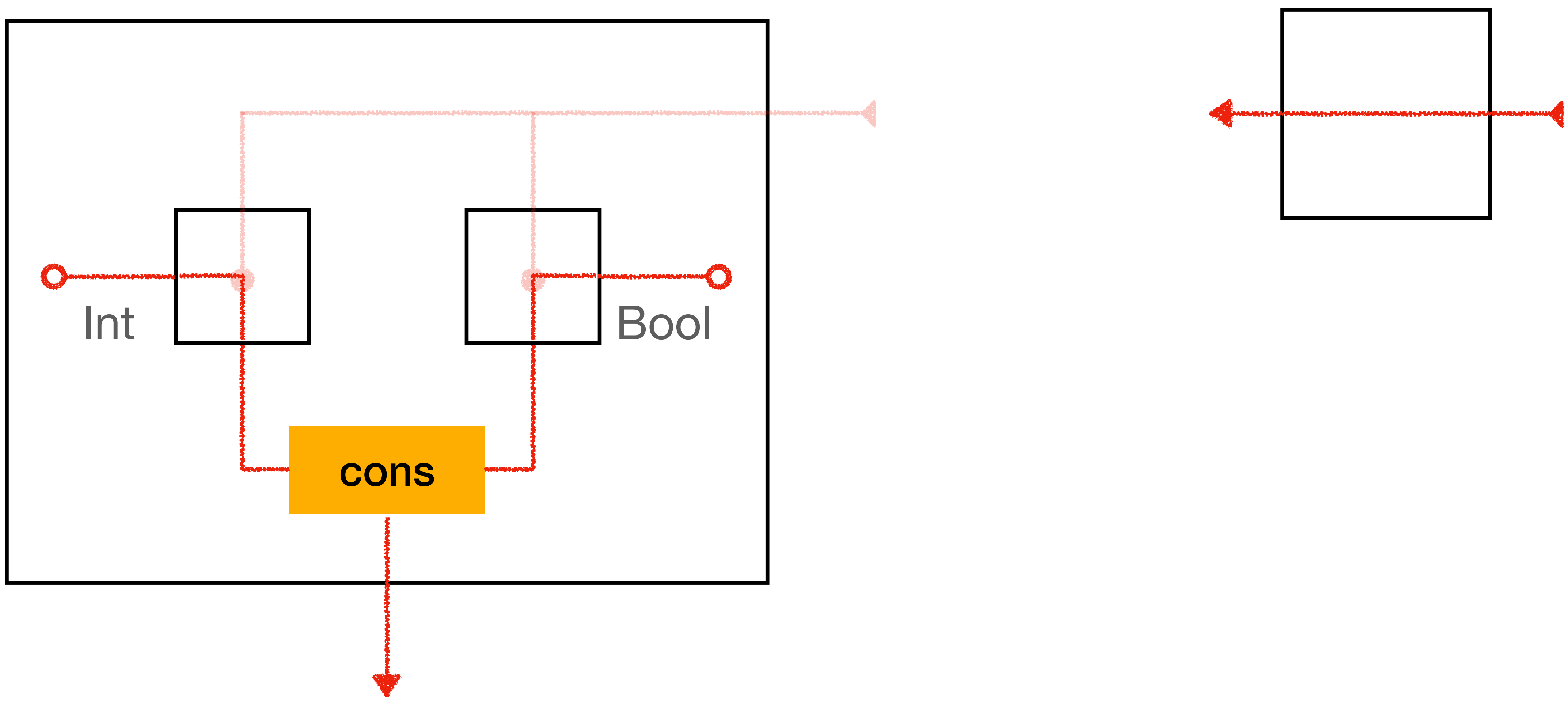
Type Inference Example



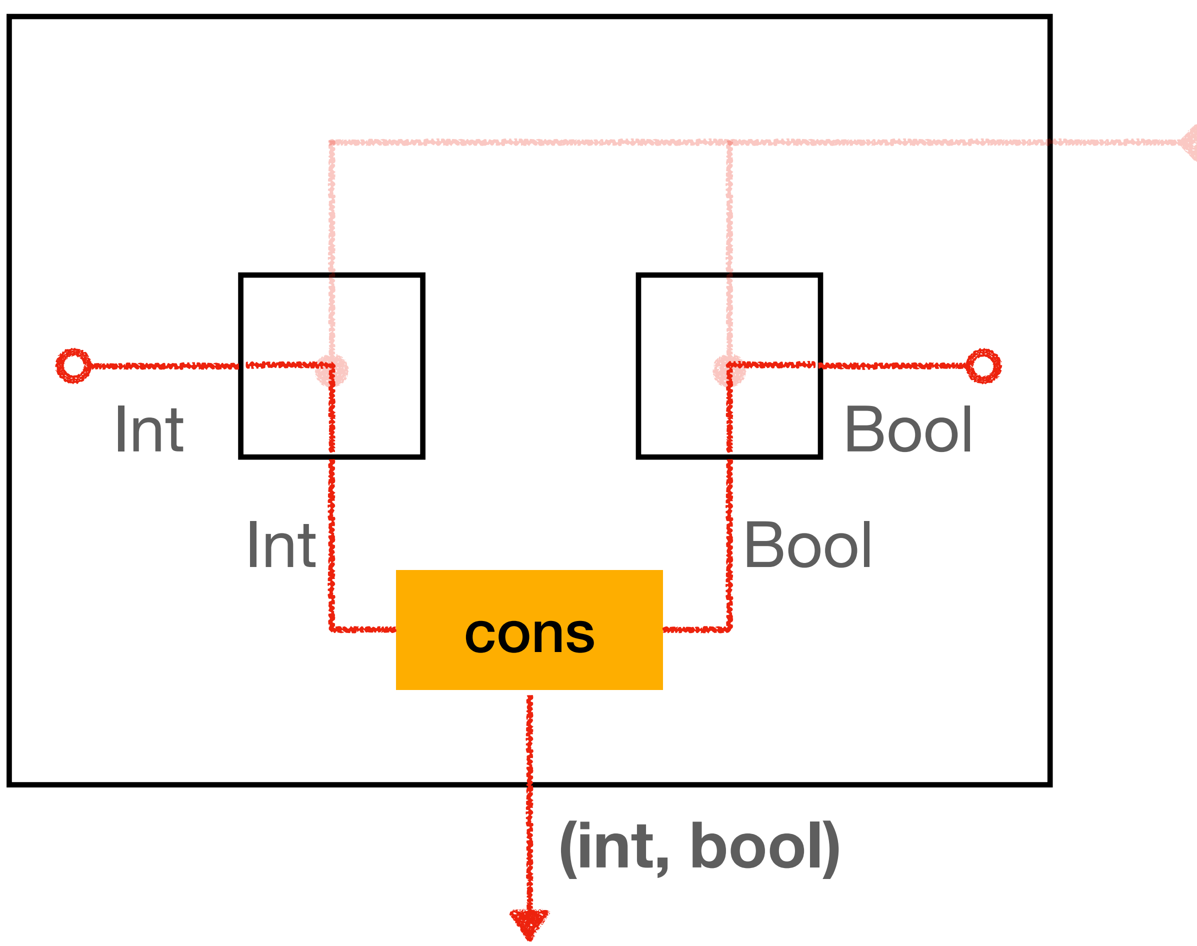
Type Inference Example

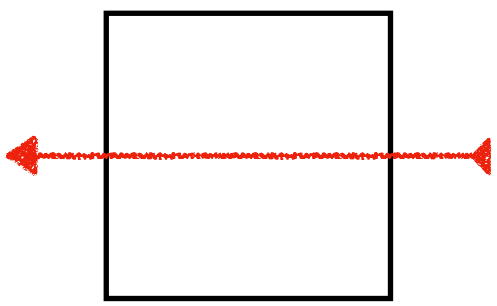


Type Inference Example

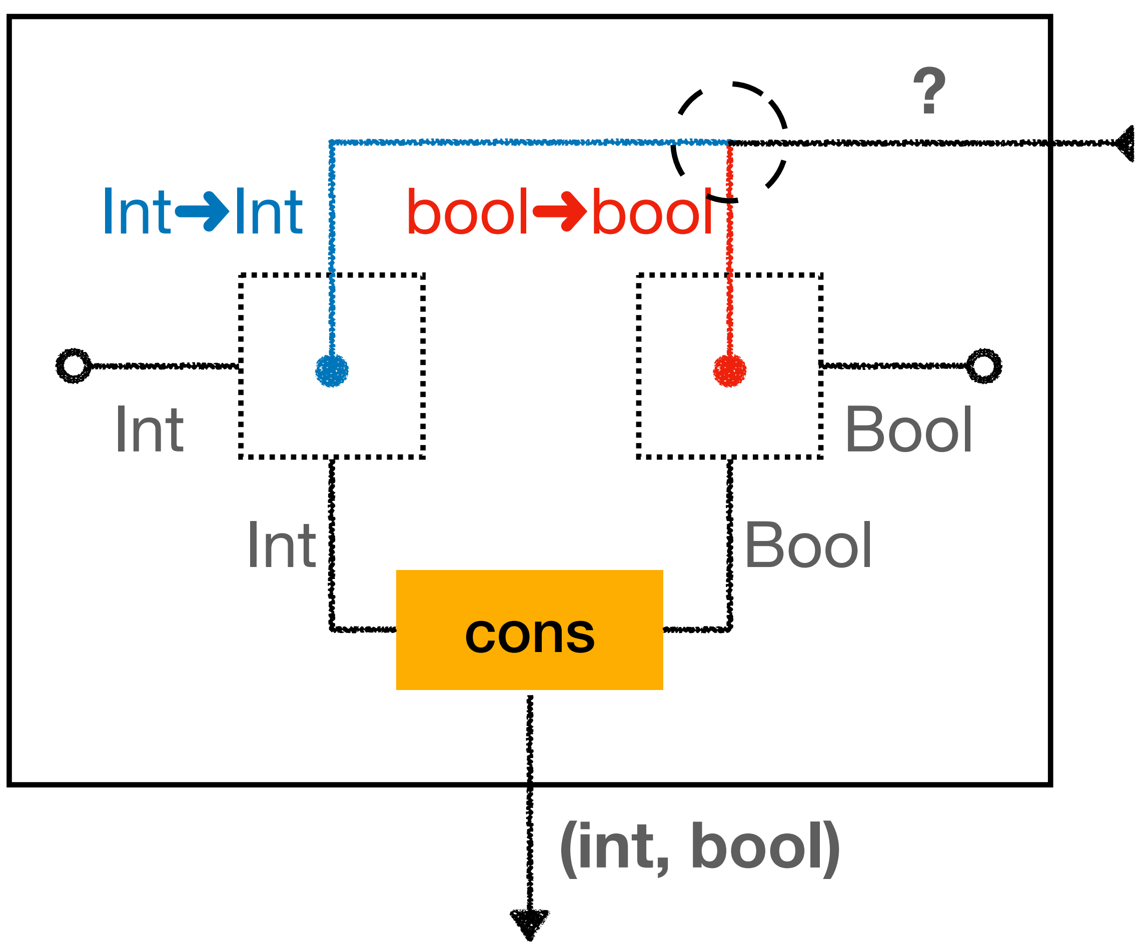


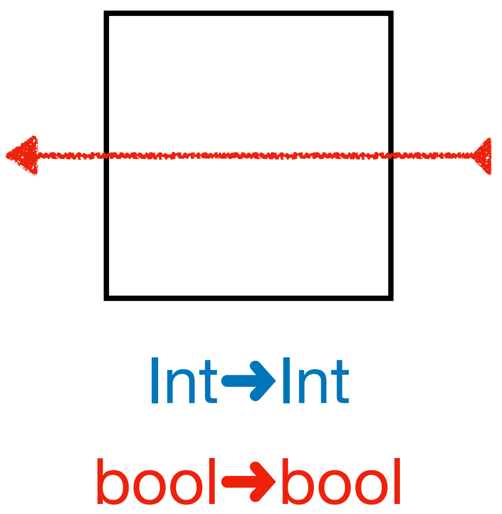
Type Inference Example



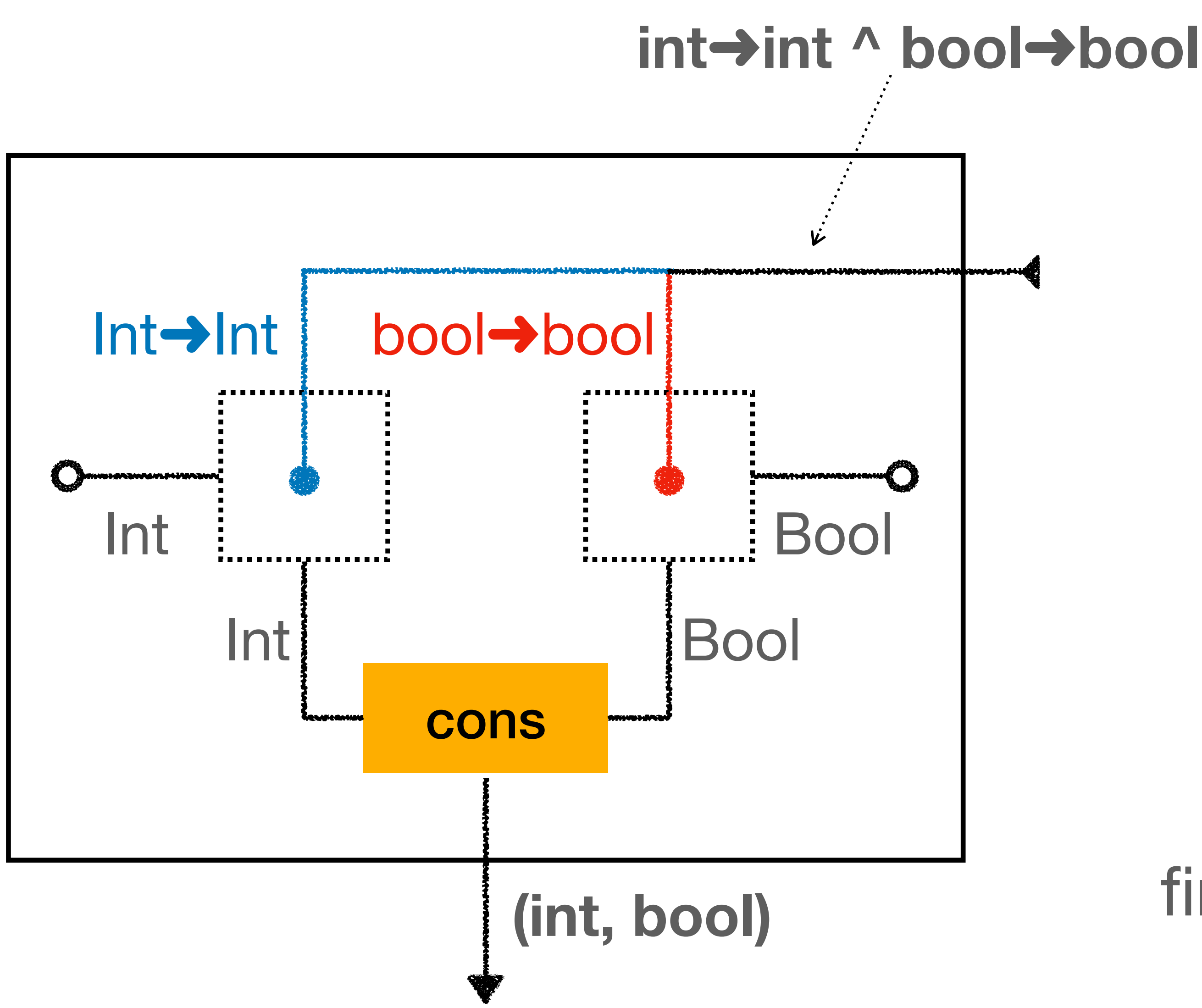


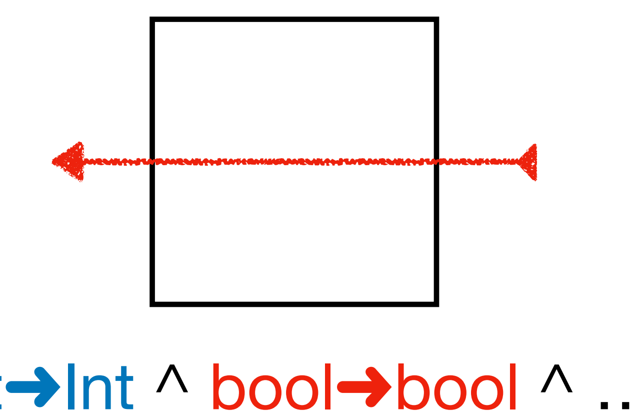
Type Inference Example





Type Inference Example

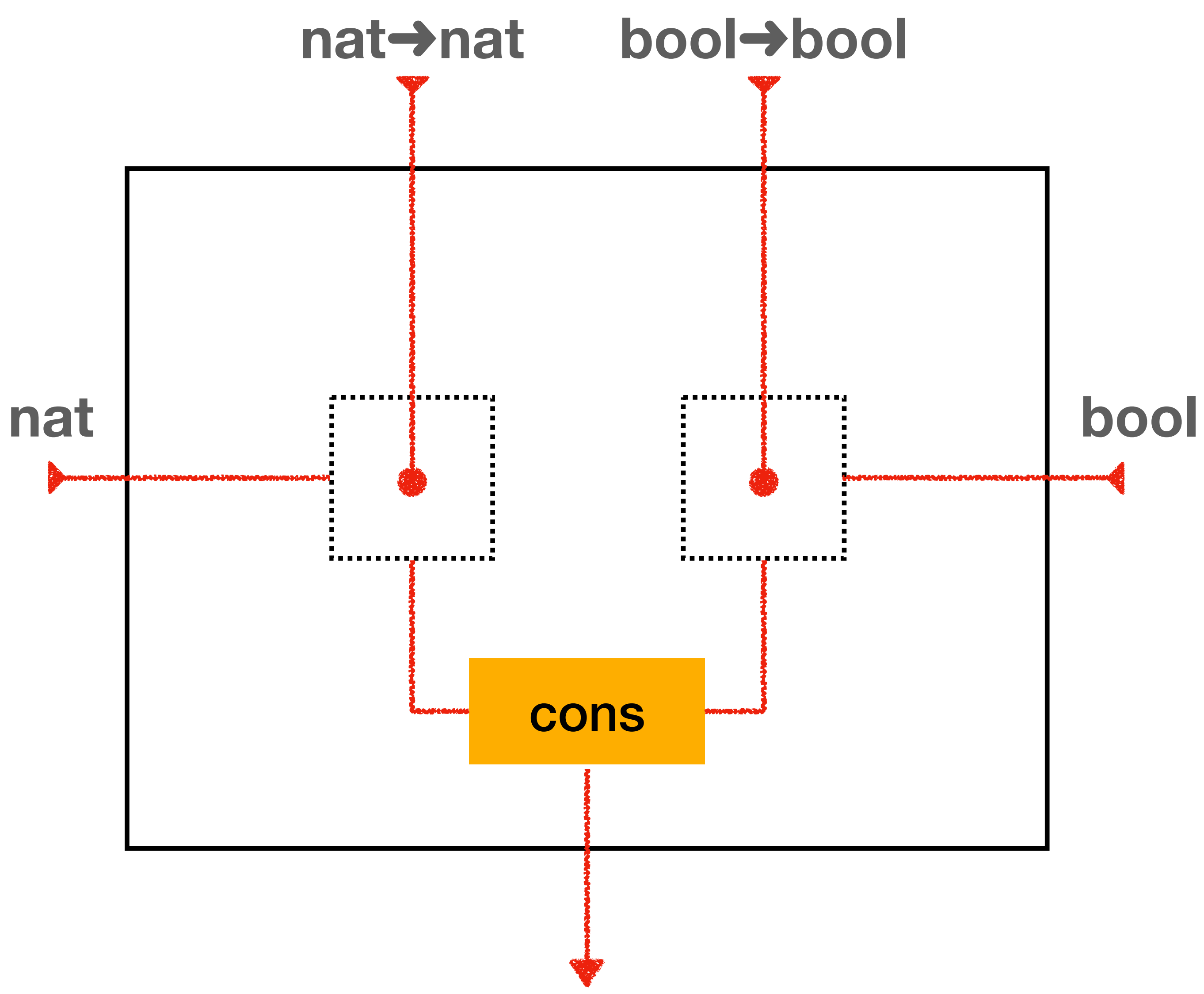


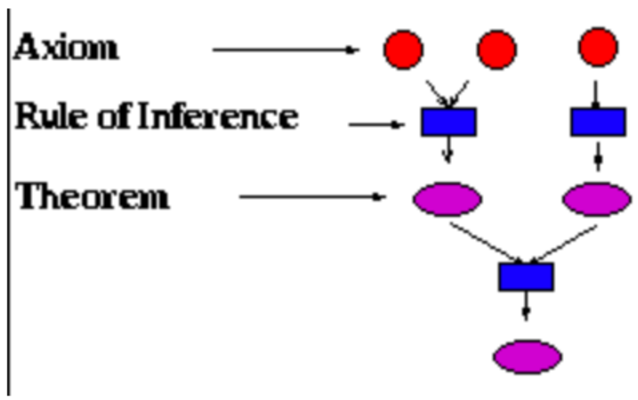


“bottom-up” inference

first-class (finite) polymorphism

Type Checking & C-H Isomorphism





Part III: Partial Evaluations

Binding-Time Analysis

annotates a program by marking as “eliminable” those parts which may be evaluated during partial evaluation

```
`(lambda (,g)
   (lambda (,d)
     ,((lambda (f)
         `((,g ,(f d)) ,f))
       (lambda (a) a))))
```

```
'(lambda (g)
   (lambda (d)
     ((g d) #<procedure>)))
```

$$\underline{\lambda} g : b_1 \underline{\to} (b_1 \overline{\to} b_1) \underline{\to} b_2 . \\ \underline{\lambda} d : b_1 . (\overline{\lambda} f : b_1 \overline{\to} b_1 . g \, \underline{@} \, (f \, \overline{@} \, d) \, \underline{@} \, f) \, \overline{@} \, \overline{\lambda} a : b_1 . a$$

$$\underline{\lambda} g : b_1 \underline{\to} (b_1 \overline{\to} b_1) \underline{\to} b_2 . \underline{\lambda} d : b_1 . g \, \underline{@} \, d \, \underline{@} \, \overline{\lambda} a : b_1 . a$$

Binding-Time Analysis

annotates a program by marking as “eliminable” those parts which may be evaluated during partial evaluation

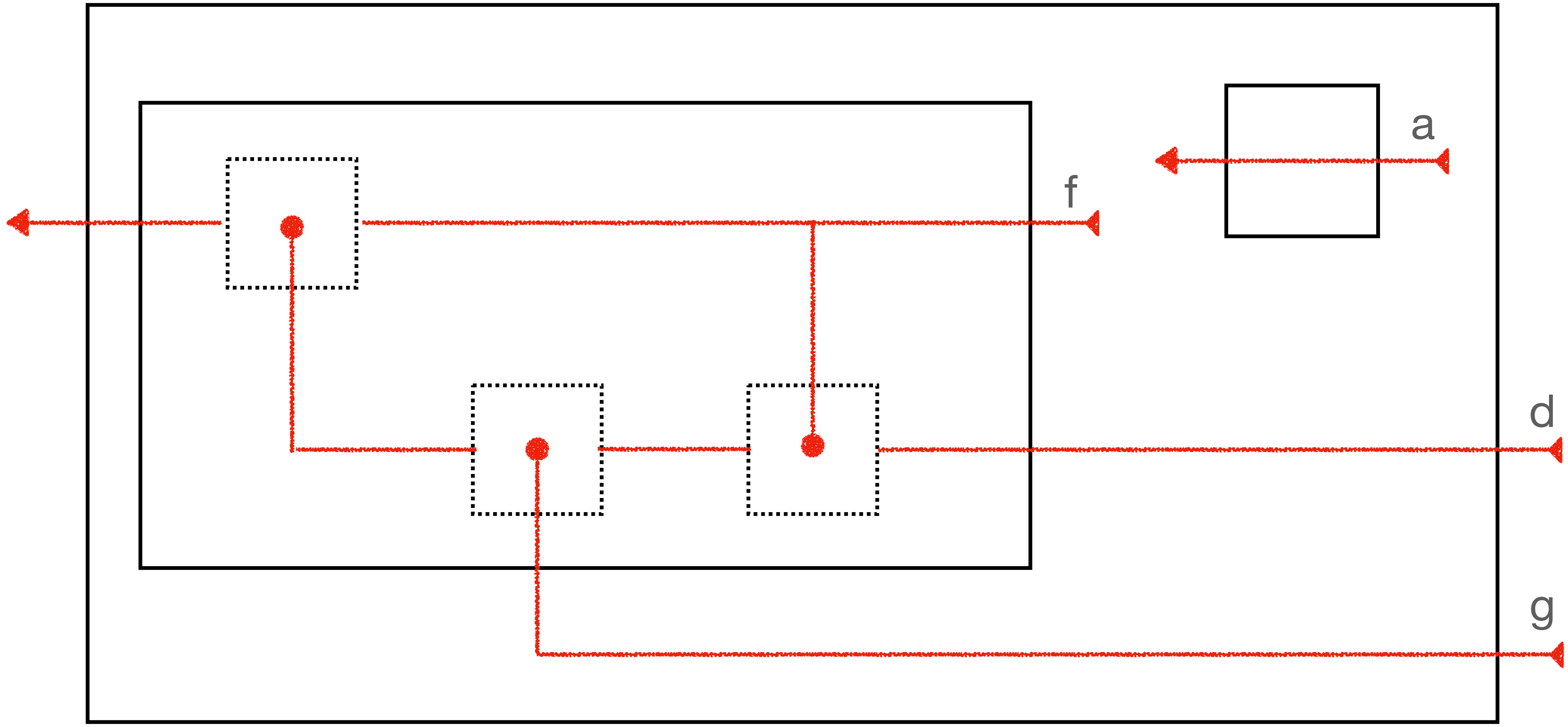
```
`(lambda (,g)
   (lambda (,d)
     ,((lambda (f)
         `((,g (,f ,d)) ,f))
       ‘(lambda (a) a))))
```

$$\underline{\lambda} g : b_1 \underline{\rightarrow} (b_1 \underline{\rightarrow} b_1) \underline{\rightarrow} b_2 .$$
$$\underline{\lambda} d : b_1 . (\overline{\lambda} f : b_1 \underline{\rightarrow} b_1 . g \,\underline{@}\, (f \,\underline{@}\, d) \,\underline{@}\, f) \,\overline{@}\, \underline{\lambda} a : b_1 . a$$

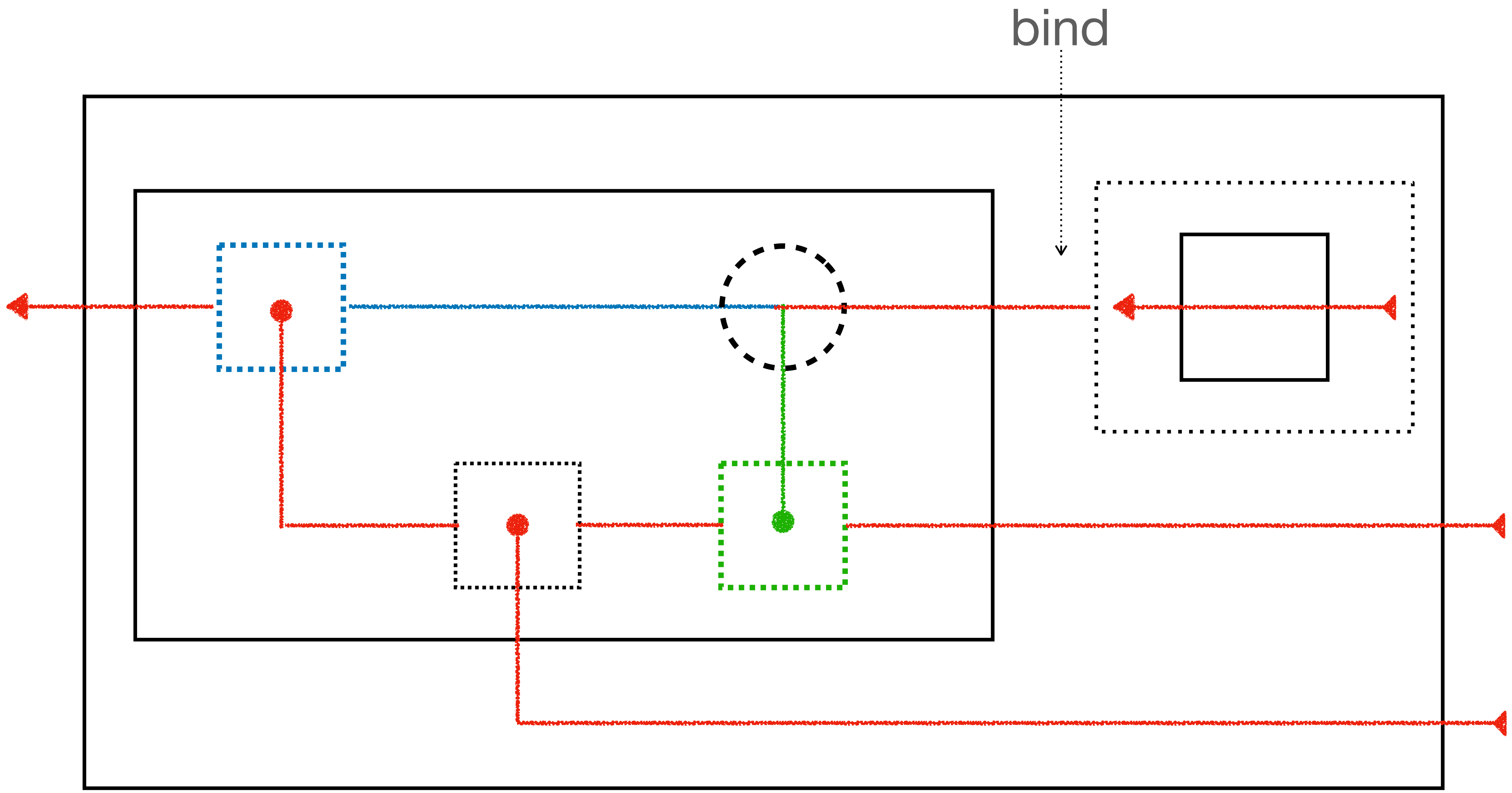
```
'(lambda (g)
   (lambda (d)
     ((g ((lambda (a) a) d))
      (lambda (a) a))))
```

$$\underline{\lambda} g : b_1 \underline{\rightarrow} (b_1 \underline{\rightarrow} b_1) \underline{\rightarrow} b_2 .$$
$$\underline{\lambda} d : b_1 . g \,\underline{@}\, ((\underline{\lambda} a : b_1 . a) \,\underline{@}\, d) \,\underline{@}\, \underline{\lambda} a : b_1 . a$$

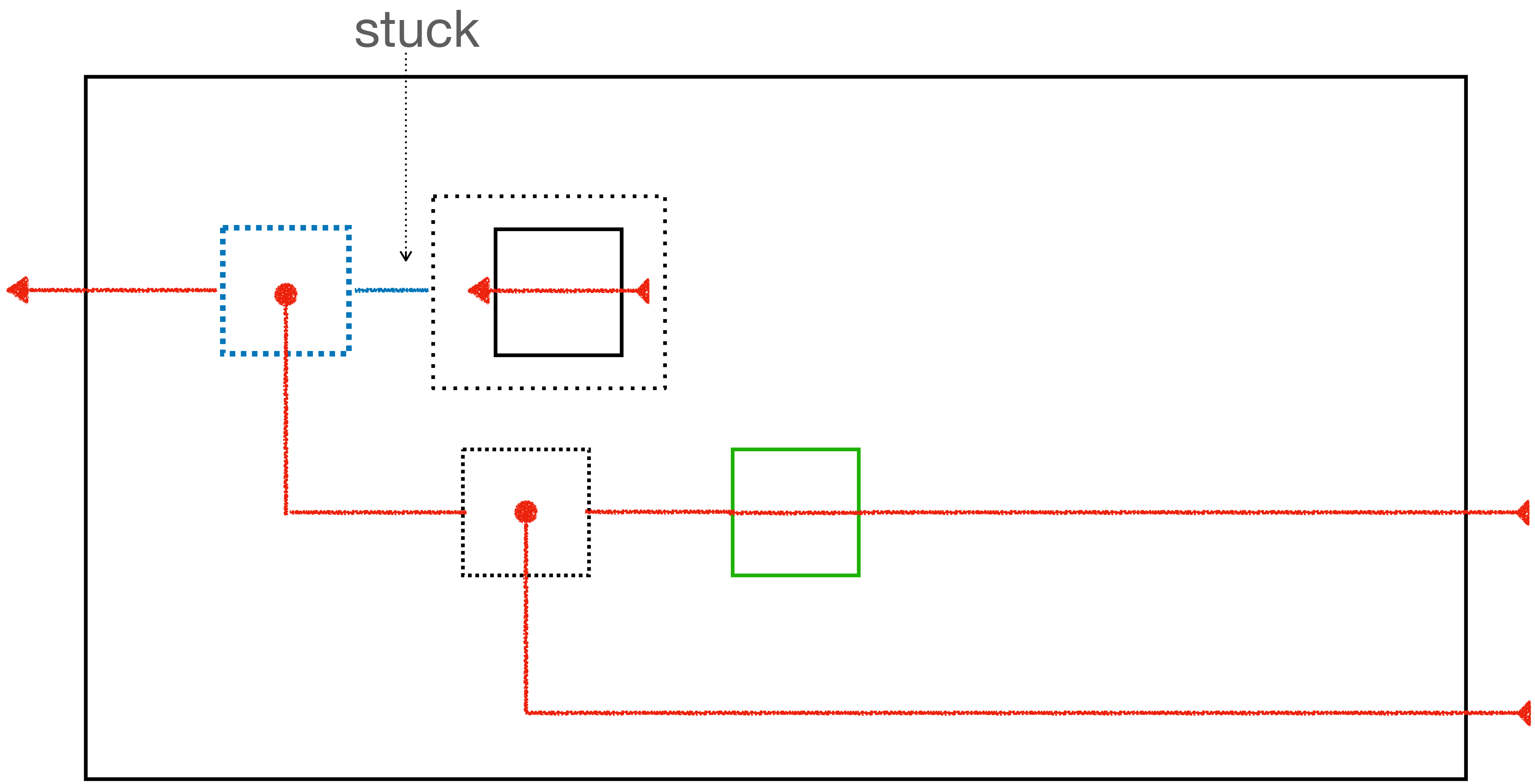
Binding-Time Analysis



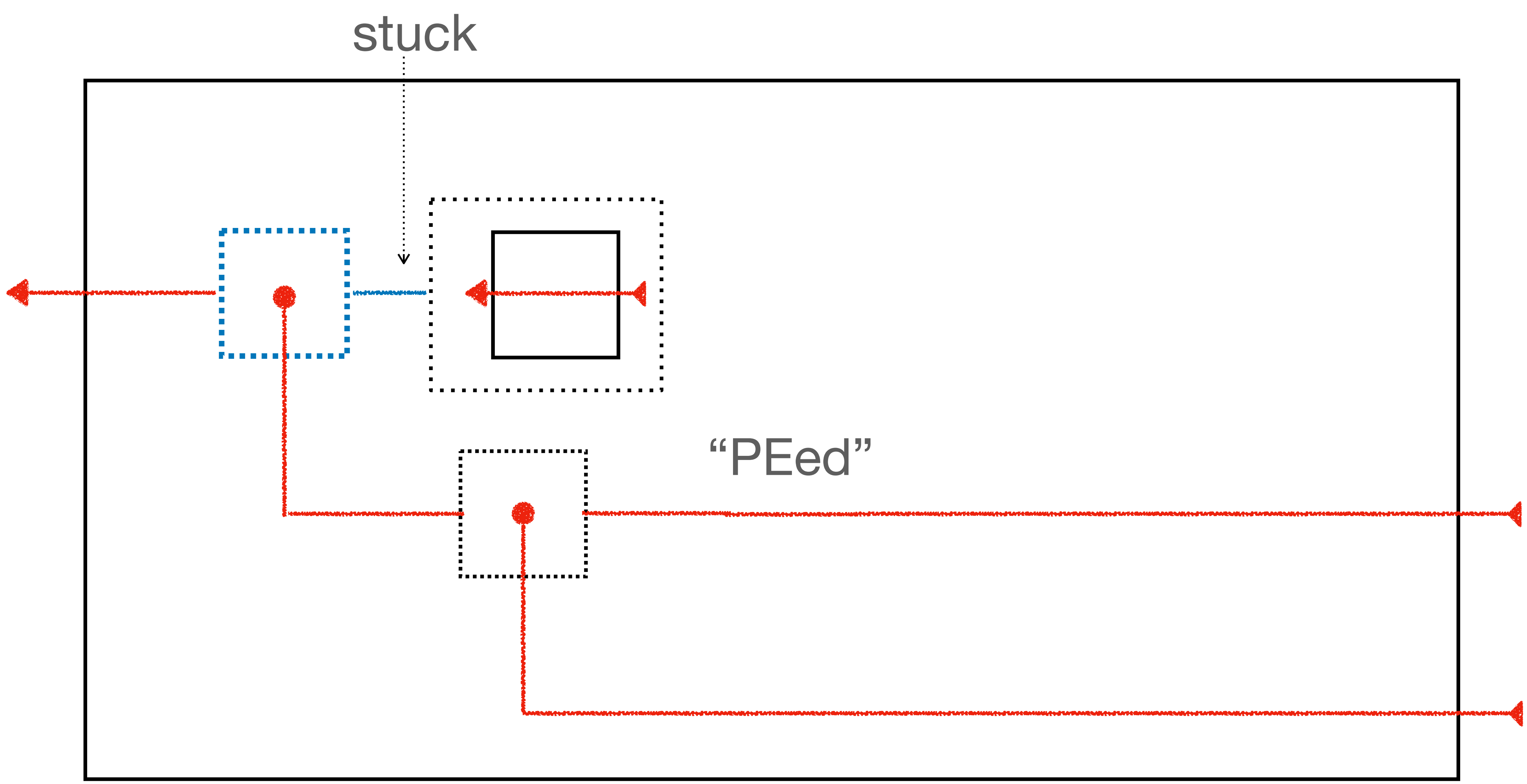
Binding-Time Analysis



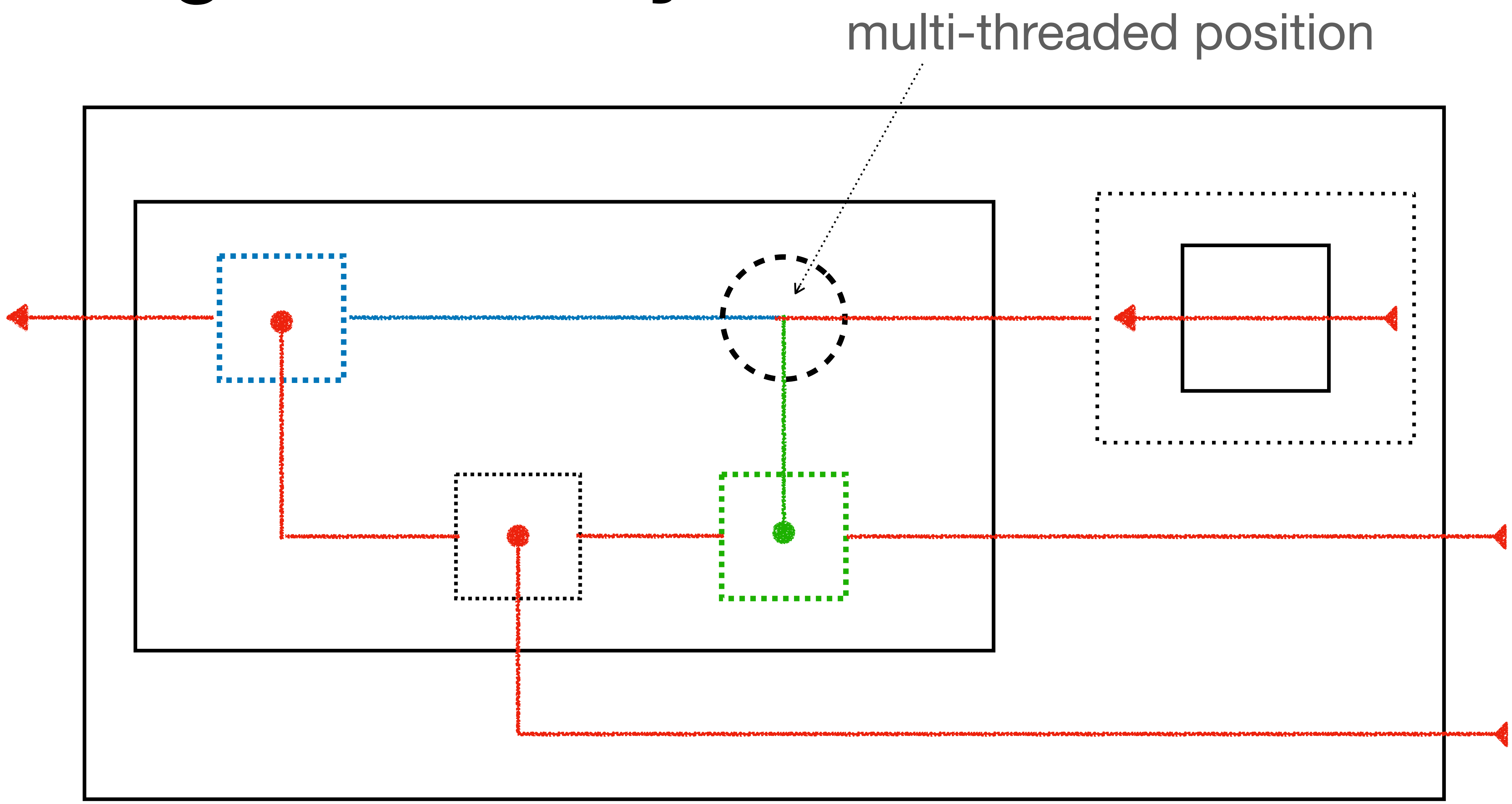
Binding-Time Analysis



Binding-Time Analysis



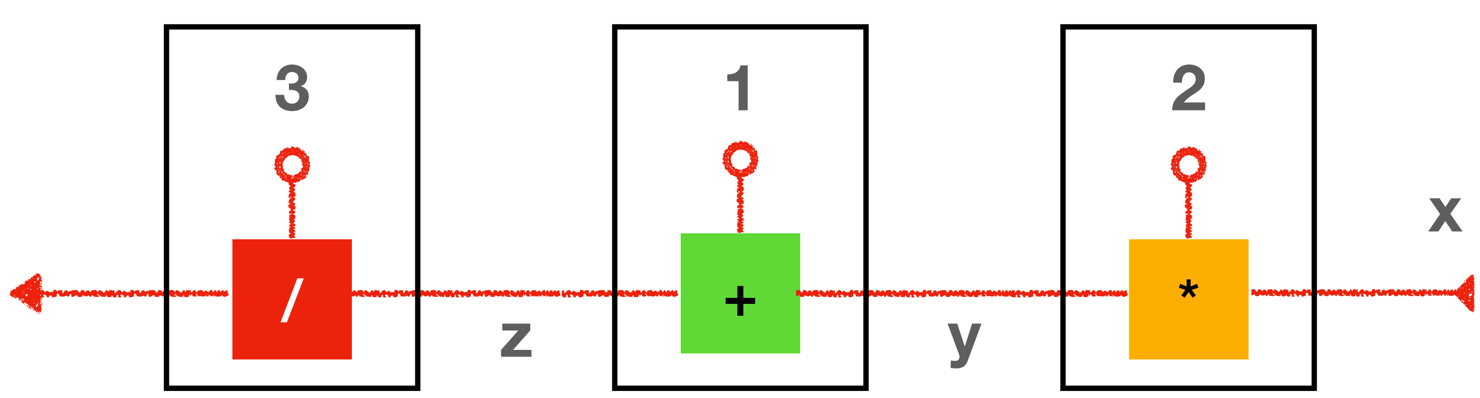
Binding-Time Analysis



"polymorphic annotations" for Nielson & Nielson's 2-level λ-calculus

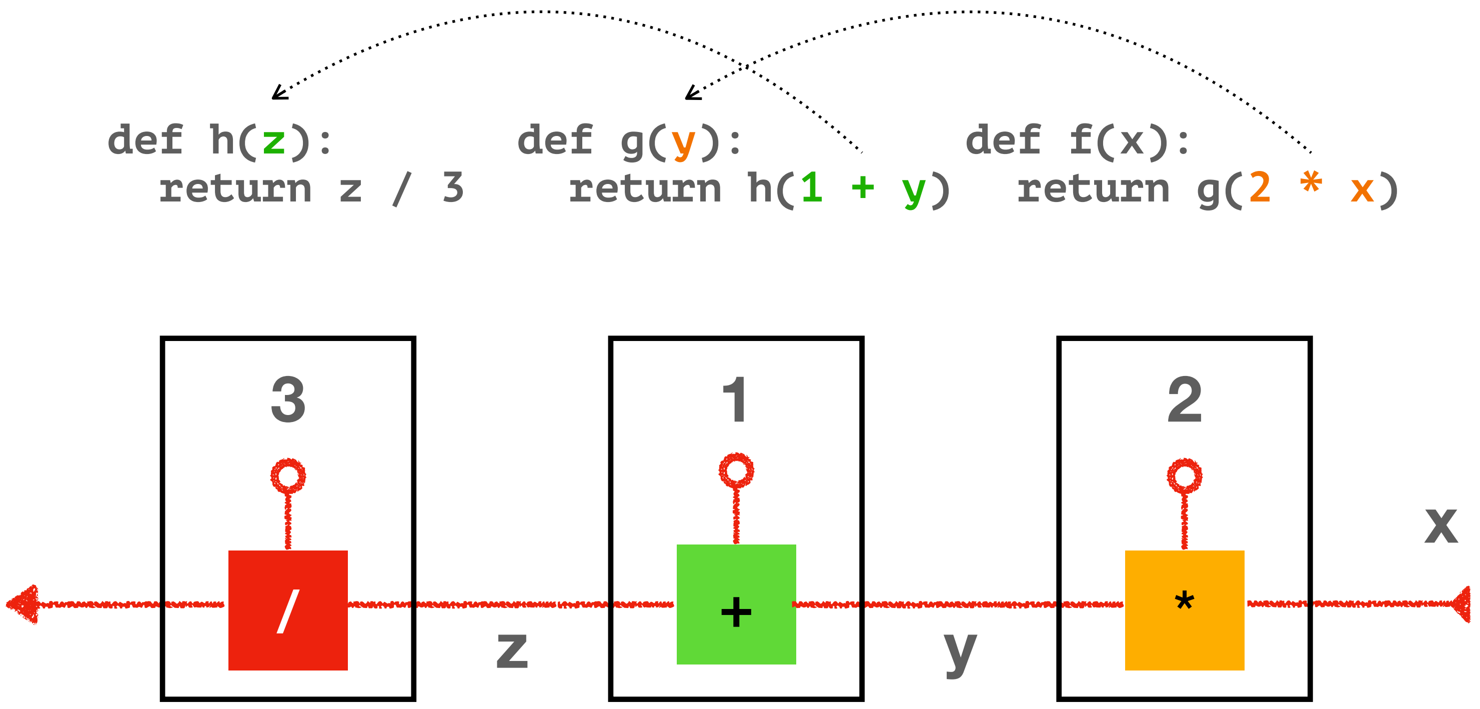
Inlining

```
def h(z):
  return z / 3

def g(y):
  return h(1 + y)

def f(x):
  return g(2 * x)
```



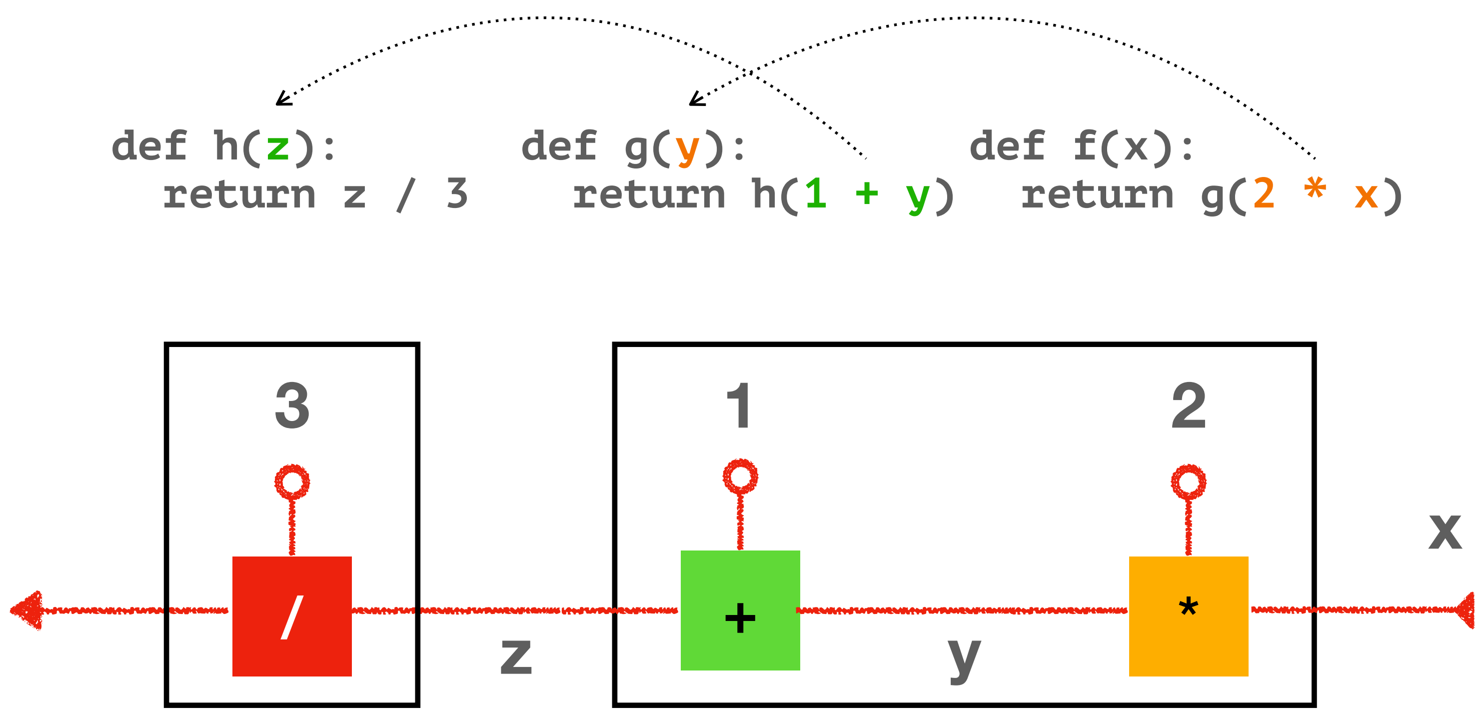
Inlining

```
def f(x):
  y = 2 * x
  return g(y)
```



Inlining

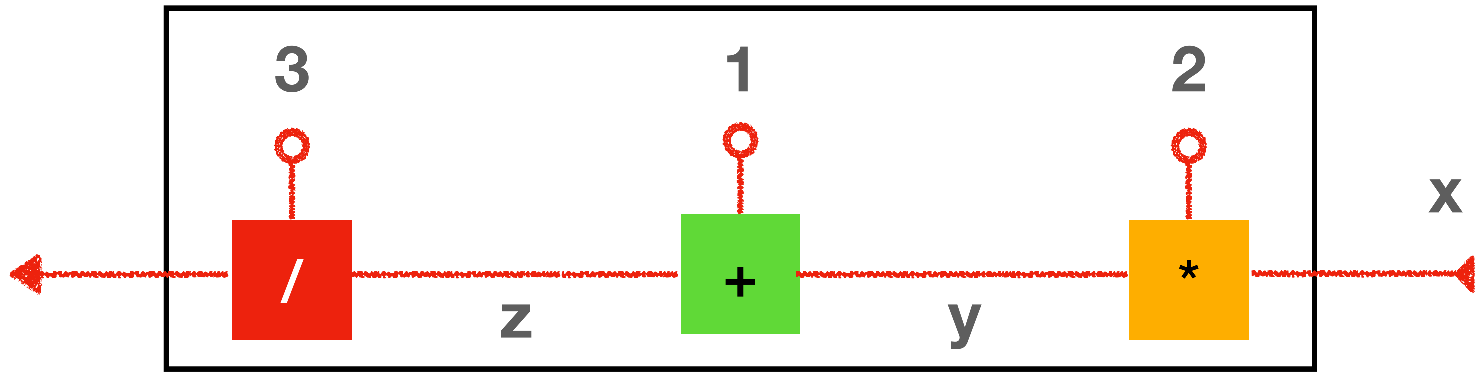
```
def f(x):
  y = 2 * x
  z = 1 + y
  return h(z)
```



Inlining

```
def f(x):
  y = 2 * x
  z = 1 + y
  return z / 3
```

```
def h(z):
  return z / 3

def g(y):
  return h(1 + y)

def f(x):
  return g(2 * x)
```



Inlining

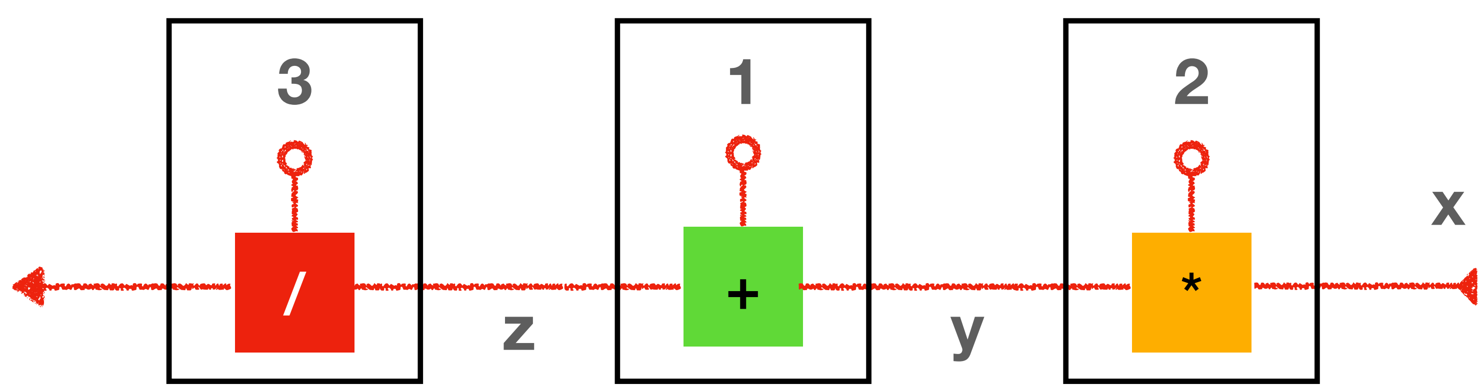
continuations!

```
def f(x):
  y = 2 * x
  z = 1 + y
  return z / 3
```

```
def h(z):
  return z / 3
```

```
def g(y):
  return h(1 + y)
```

```
def f(x):
  return g(2 * x)
```




```
(/ (+ 1 []) 3)

(/ [] 3)

(+ 1 [])
```

Inlining

Observations:

- variables and function arguments are the same in essence
- they are both “anchors” (or nodes, points) of data flow
- they are both wires (or pins) in λ-circuit

Related: “point-free” style

Supercompilation

Valentin F. Turchin: *The Concept of Supercompiler*

“Supercompilation can lead to a very deep structural transformation of the original program; it can improve the program even if all the actual parameters in the function calls are variable.”

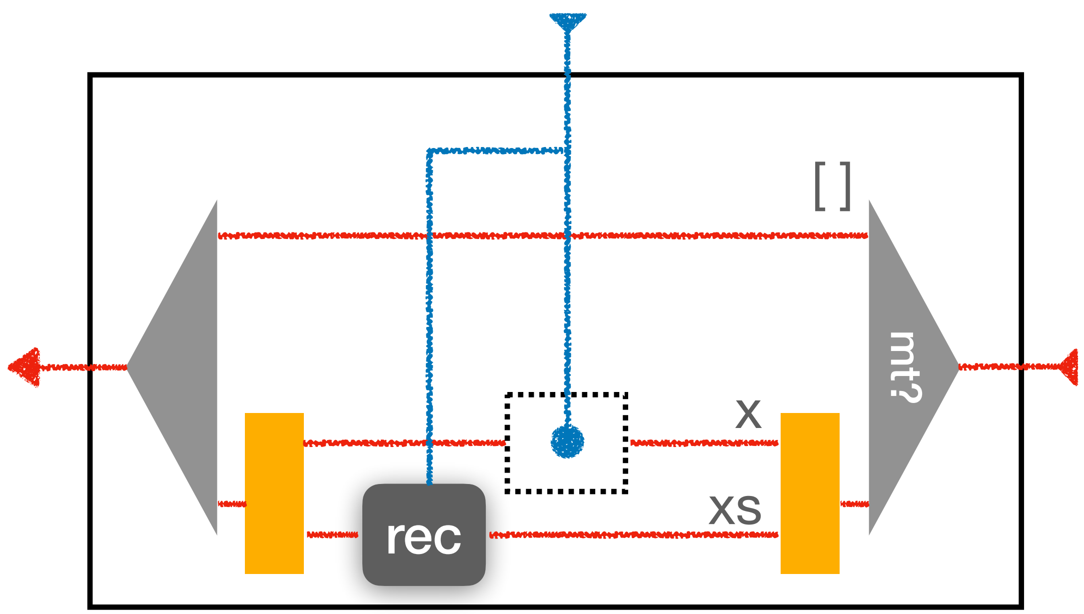
Neil Mitchell: *Rethinking Supercompilation*

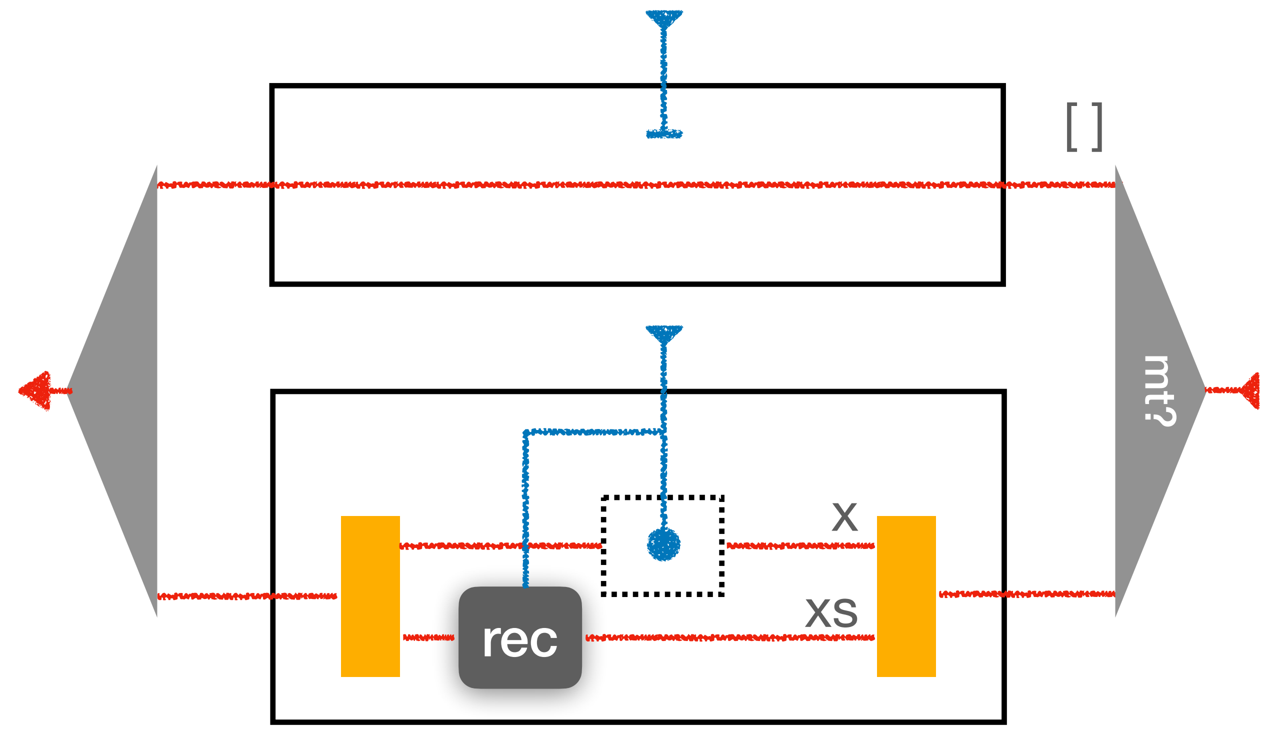
map f (map g ls) ⟶ map (f ∘ g) ls

root f g ls = map f (map g ls))

transform a 2-pass algorithm into 1-pass

Supercompilation

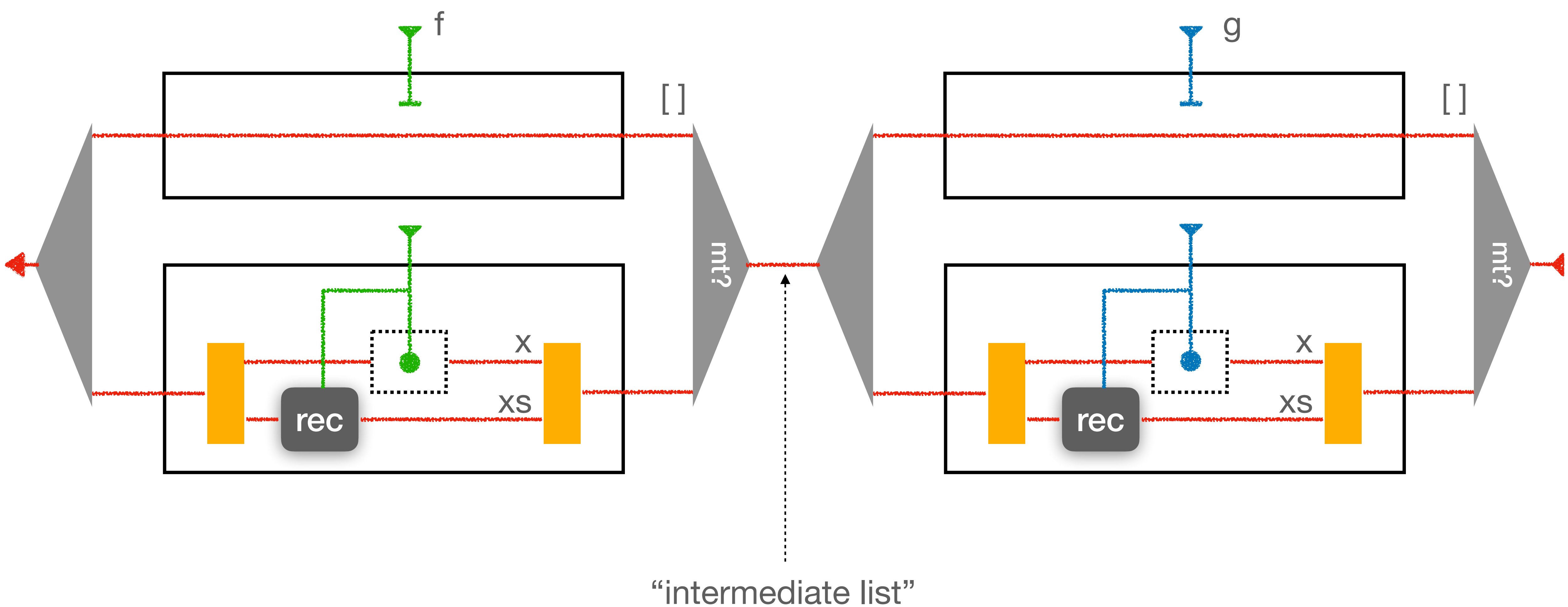




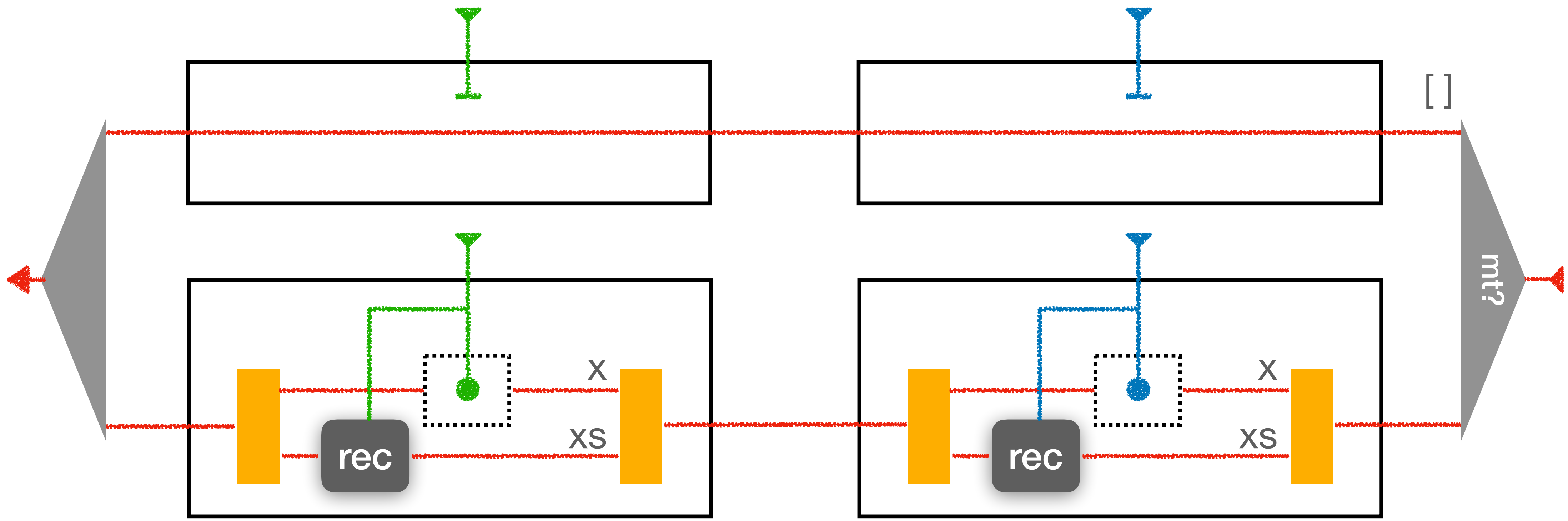

```
map f ls = case ls of
             [] -> []
             x:xs -> (f x):(map f xs)
```

```
map f [] = []
map f x:xs = (f x):(map f xs)
```

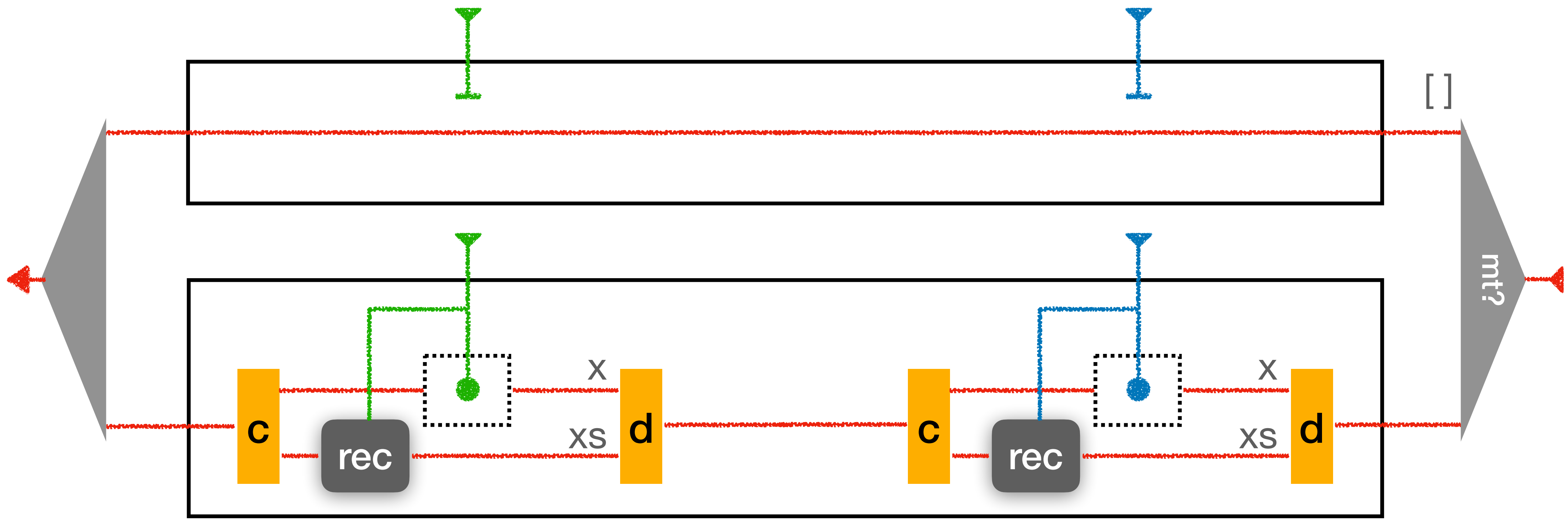
Supercompilation



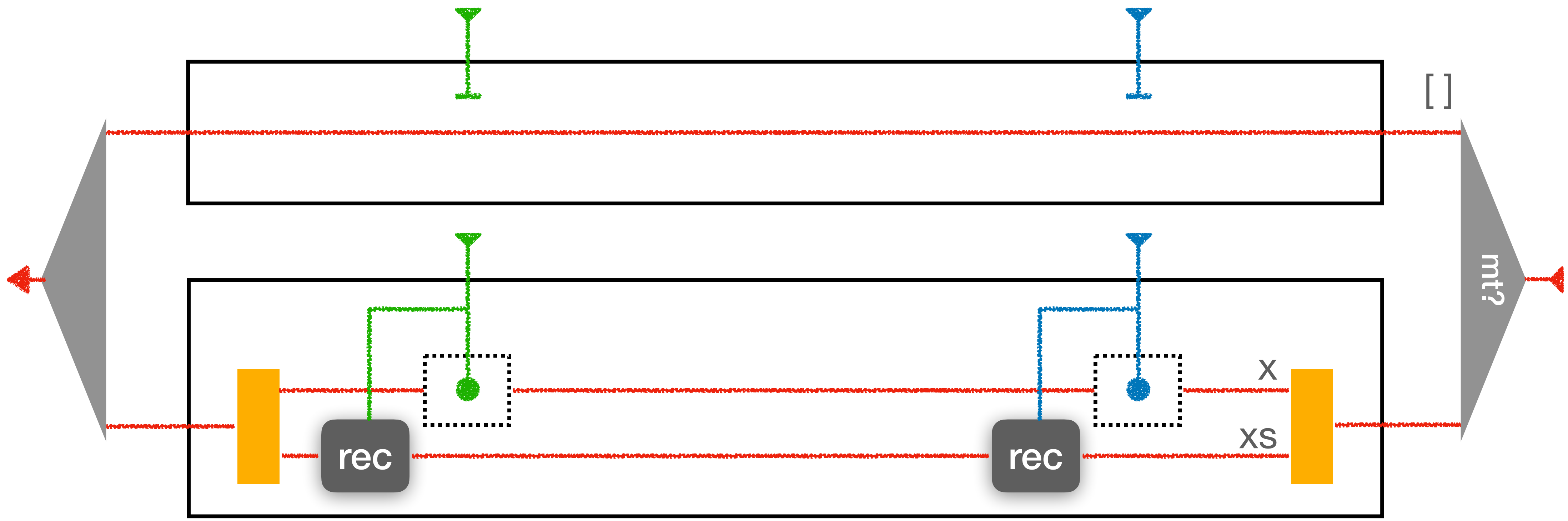
Supercompilation



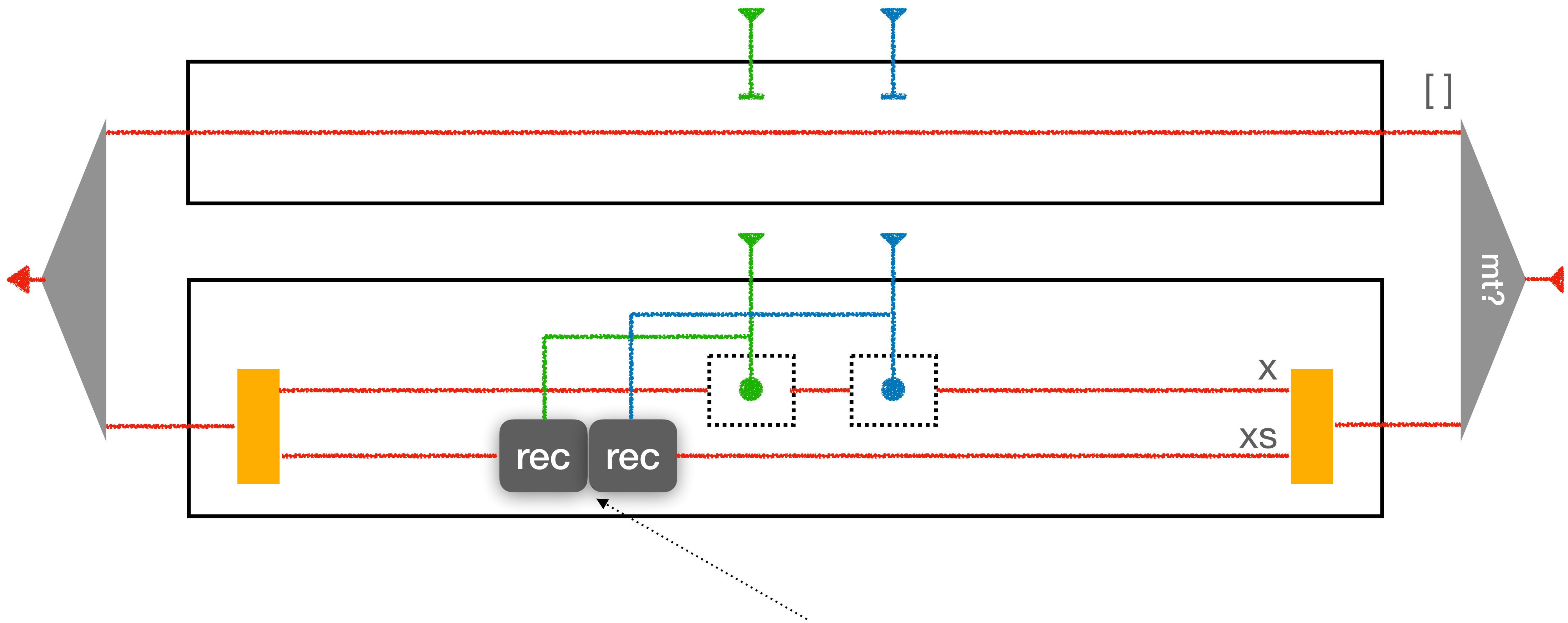
Supercompilation



Supercompilation

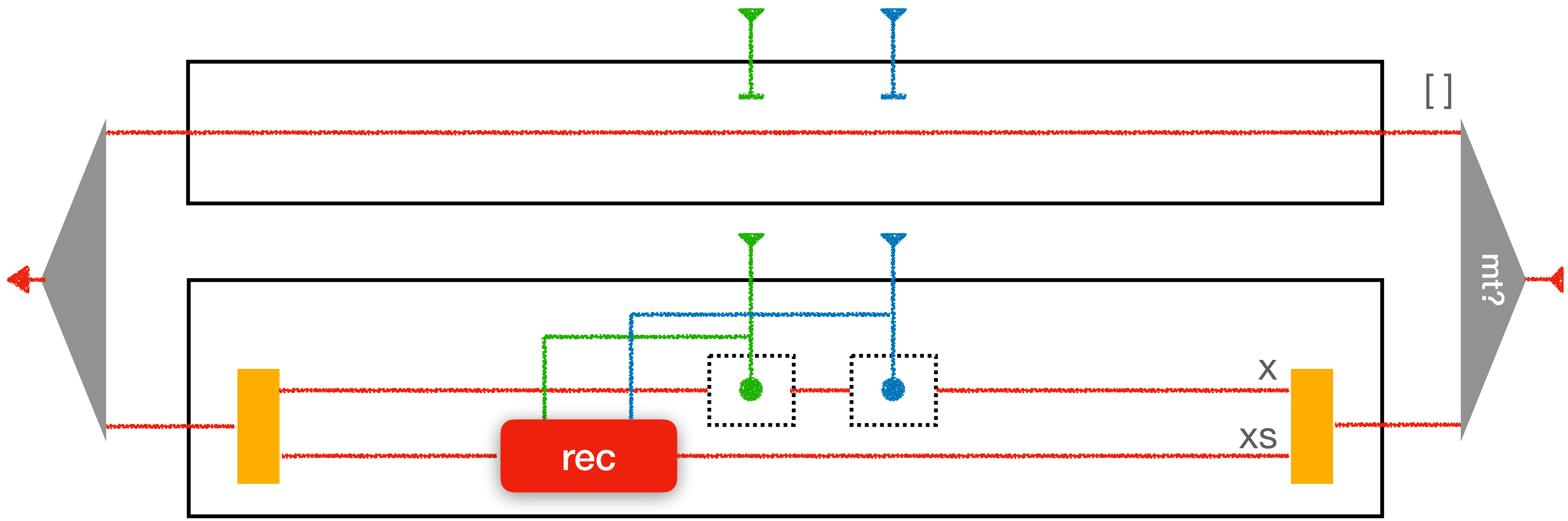


Supercompilation



root f g ls = map f (map g ls))   map f (map g xs)) = root f g xs

Supercompilation



root f g ls = map f (map g ls)) ⟶ root f g [] = []
root f g x:xs = (f (g x)) : (root f g xs)

map (f ∘ g) ls

Supercompilation

map f (map g ls) <=> map (f ∘ g) ls

programs as propositions
program transformations as proofs?

PE summary

- remove boxes (β-reduction, static app.)
- add boxes (η-expansion, Kleene's Smn)
- swallow boxes (e.g. inlining)
- split (larger) boxes
- collapse DEMUX (e.g. constant prop.) ⟵ reduce "interpretive overheads"
- cancel MUX/DEMUX pairs
- cancel cons/elim pairs
- …

What about …

- Handy and intuitive for draft
- Alternative view on PL
- Concepts made explicit
- Just for fun

- Not formal
- Hard for proofs
- Hard to formalize
- Higher-order (>2)
- …

borderlines between:

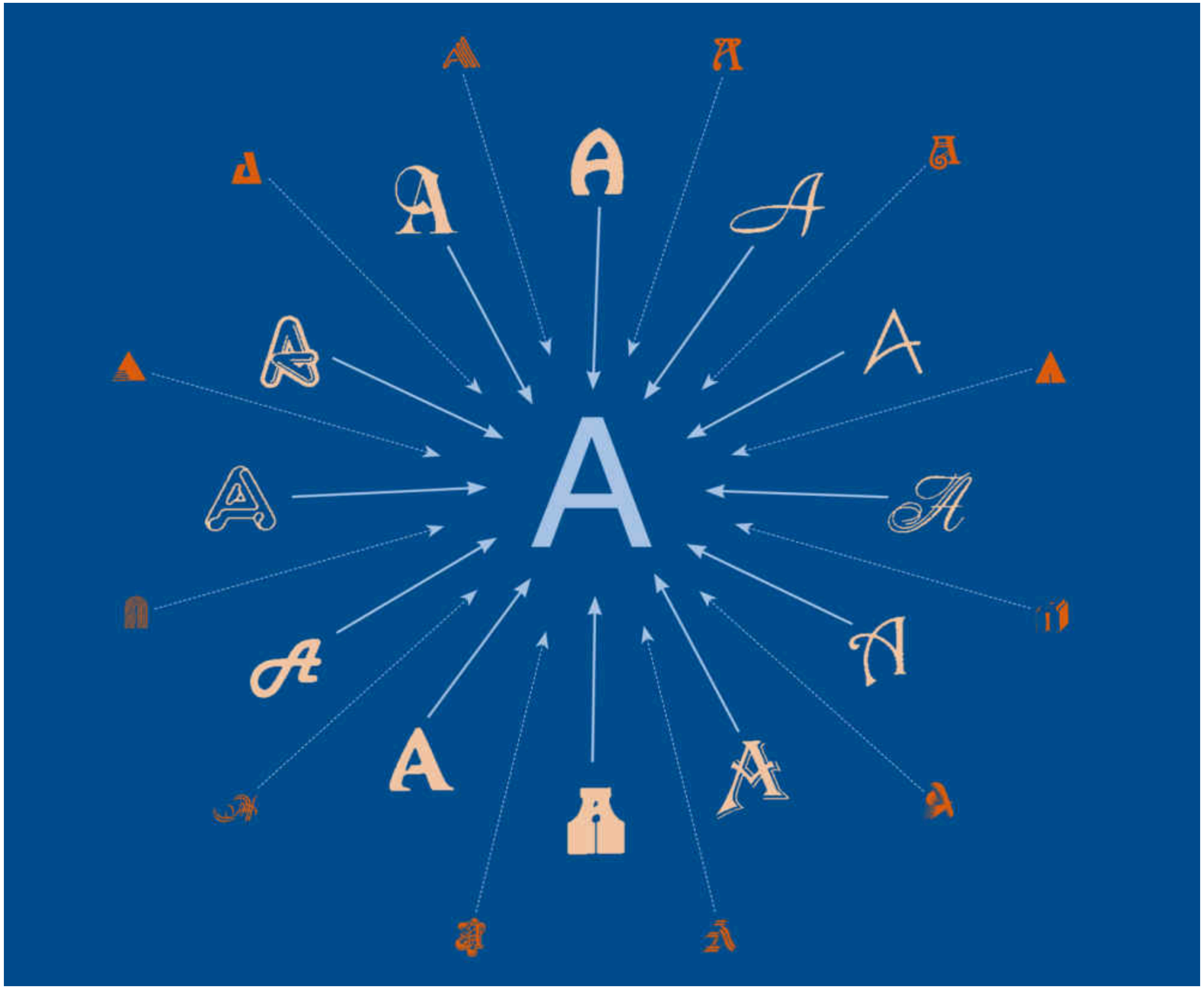
- “textual” and “diagrammatic”
- formal and informal

A New Model Of Computation?

No.

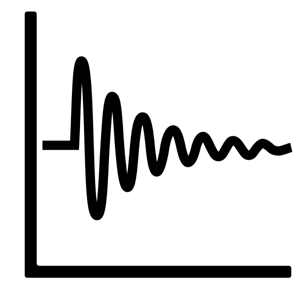
It is the λ-calculus you are familiar with.

“Different in surfaces, but the same in essences.”



Discussion: States, Pureness, & Side-Effects?

wires (wired)

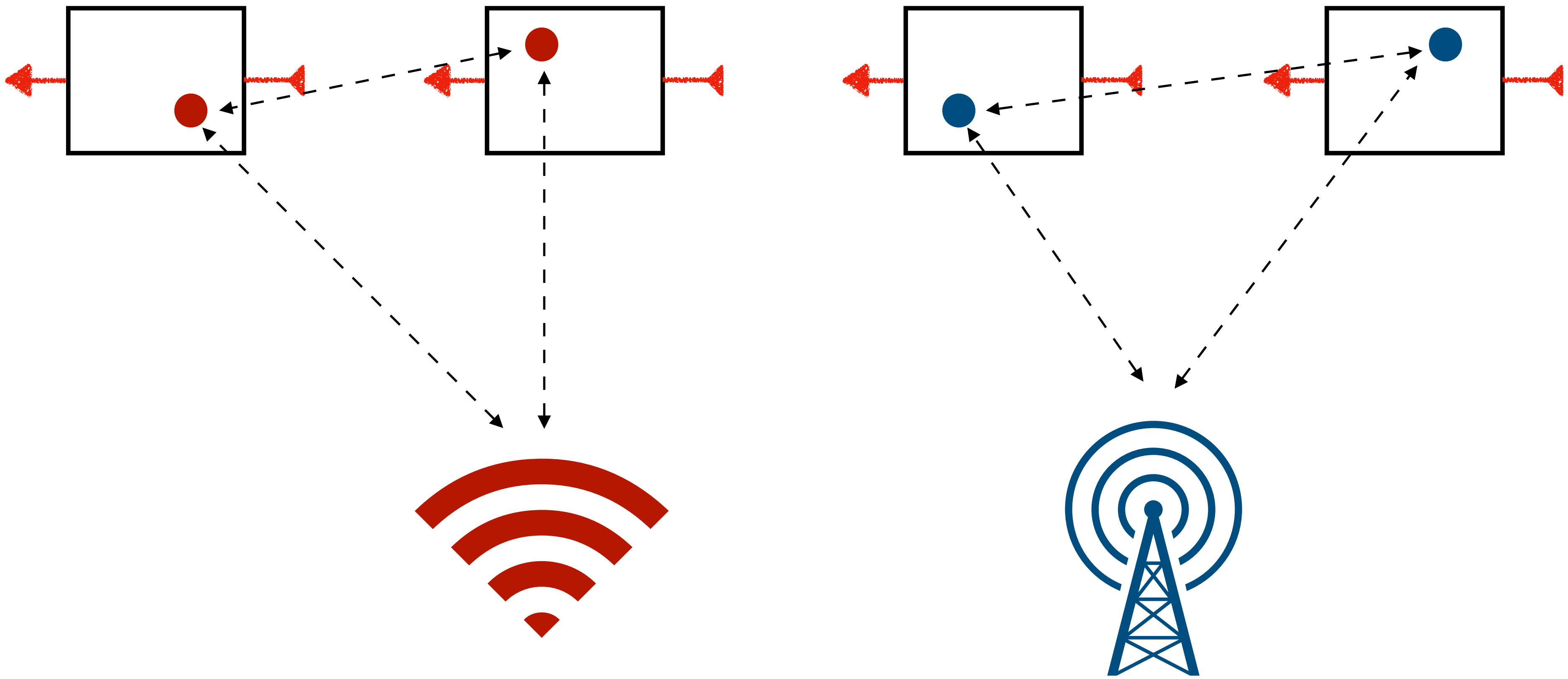


waves (wireless)

convey information losslessly

Global States ~ Covert Channels

unsafe?

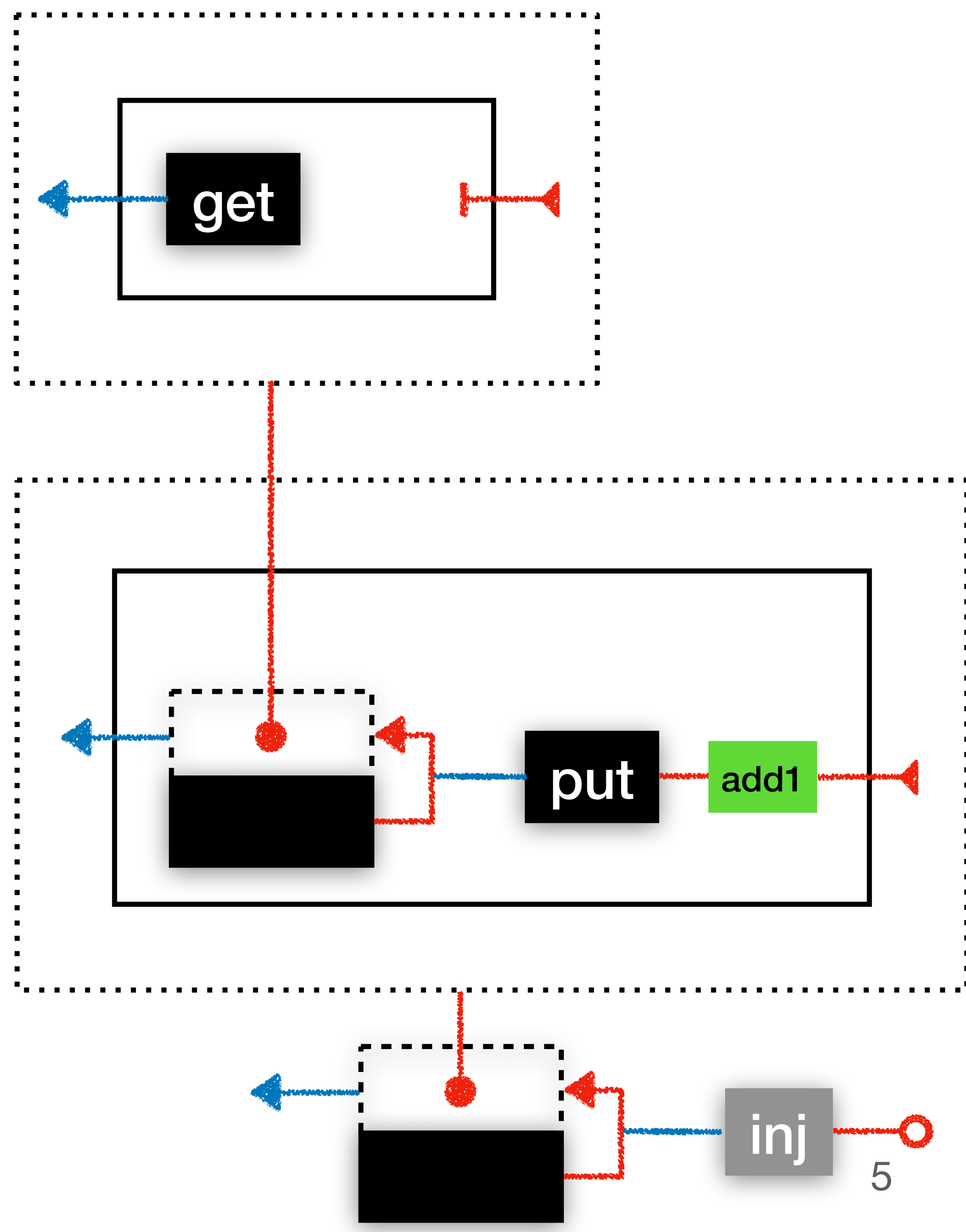


global side-effects are like WiFi!

State Monads: A “Wired” World

“junction boxes”

```
(bind-state
  (inj-state 5)
  (λ (x)
    (bind-state
      (put (add1 x))
      (λ (_)
        (get)))))
```



References & Inspirations

- https://csvoss.com/circuit-notation-lambda-calculus
- https://lukc1024.github.io/visualize-lambda/
- https://dkeenan.com/Lambda/ (To Dissect a Mocking Bird)
- Wang, Y.: A Fresh View at Type Inference
- Danvy, O.: Type Directed Partial Evaluation
- Mitchell, N.: Rethinking Supercompilation
- Nielson, F. and Nielson, H.: Two-Level Functional Languages
- Turchin, V.: The Concept of a Supercompiler

Many graphical representations of lambda calculus…