Incremental Computation with **Adapton**

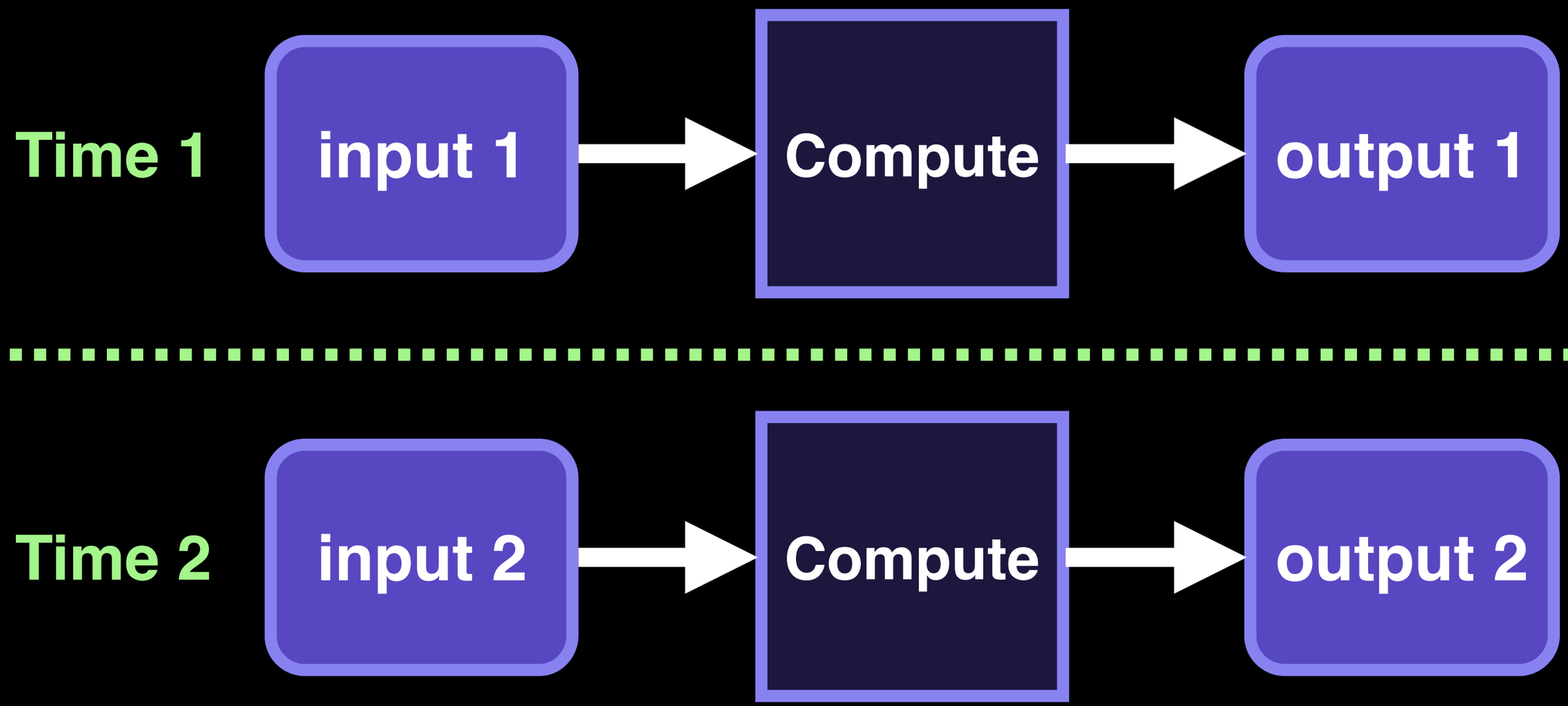
Matthew A. Hammer
Kyle Headley

University of Colorado, Boulder

PLEMM 2017 — Facebook
May 4, 2017

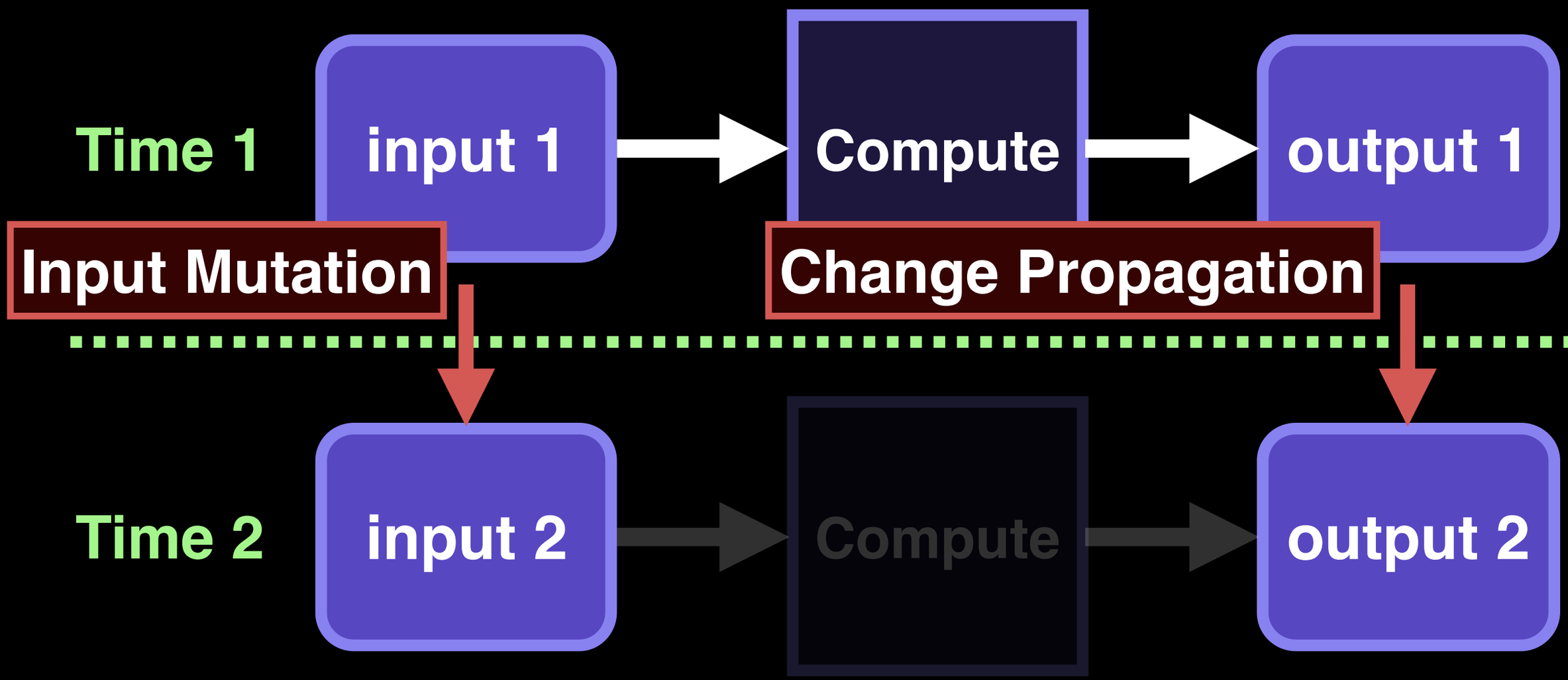
Incremental Computation

A computation is **incremental** if:
repeating it with a **changed input**
is **faster** than **from-scratch re-computation**



Incremental Computation

A computation is **incremental** if:
repeating it with a **changed input**
is **faster** than **from-scratch re-computation**



Incremental Computation

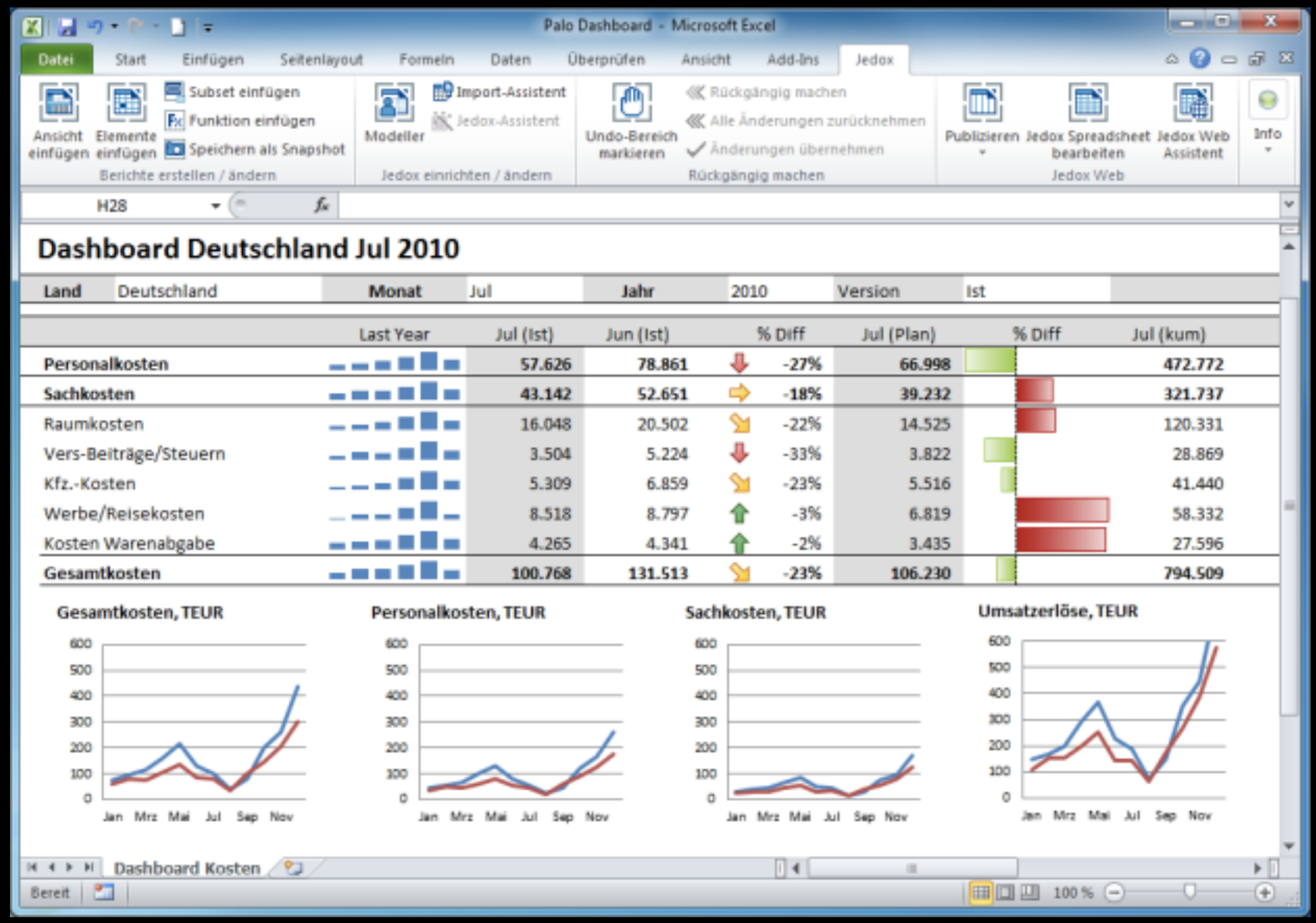
A computation is **incremental** if:
repeating it with a **changed input**
is **faster** than **from-scratch re-computation**

Incremental Computation
is Everywhere!

Spreadsheets

Change data and formula

Recompute data and charts



Excel

Makefiles / Build Systems

Change source code

Recompute compiled binaries

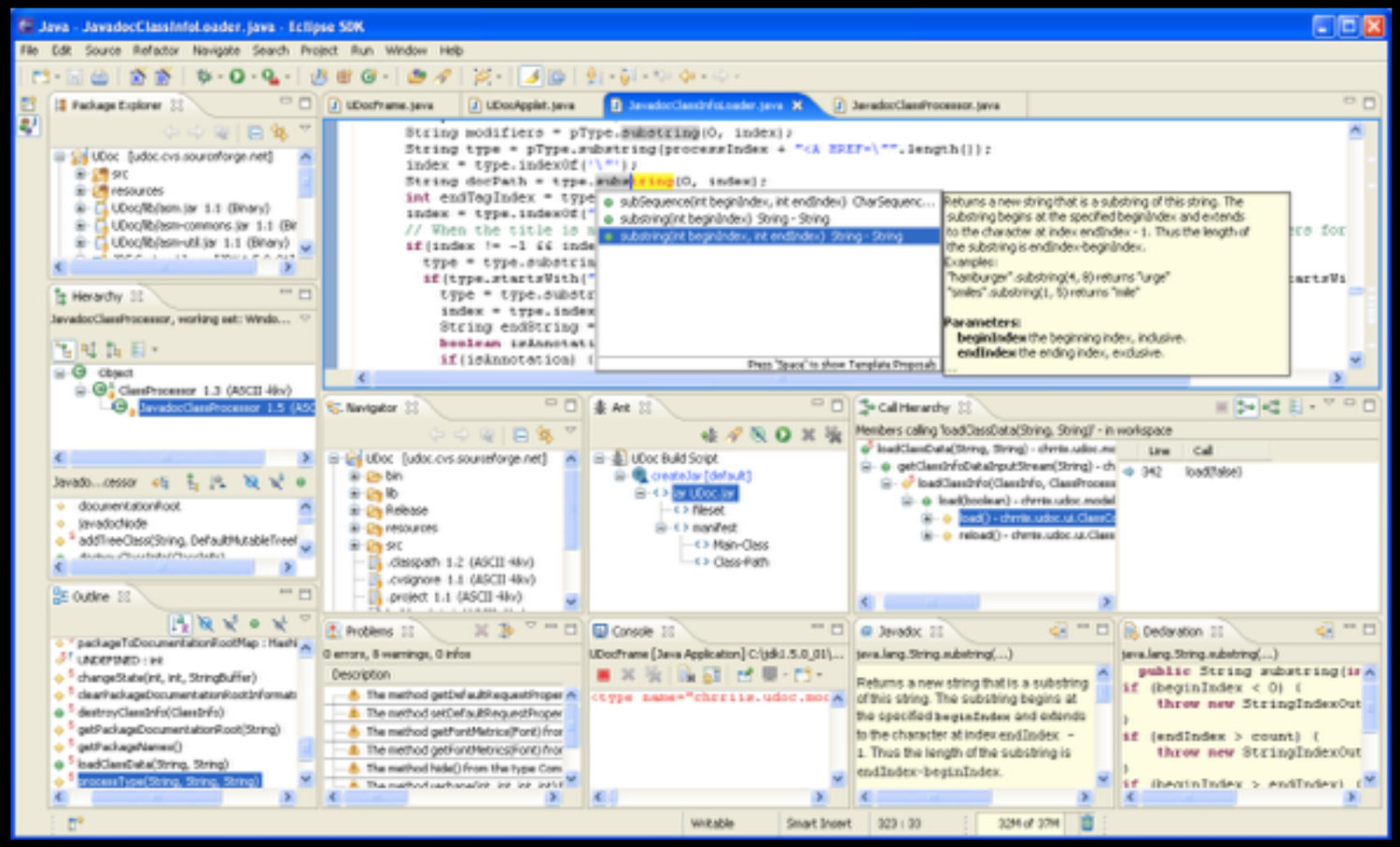
```
PROGRAM = foo
C_FILES := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(C_FILES))
CC = cc
CFLAGS = -Wall -pedantic
LDFLAGS =
LDLIBS = -lm

all: $(PROGRAM)

$(PROGRAM): .depend $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $(PROGRAM) $(LDLIBS)

depend: .depend

.depend: cmd = gcc -MM -MF depend $(var); cat depend >> .depend;
.depend:
    @echo "Generating dependencies..."
    @$(foreach var, $(C_FILES), $(cmd))
    @rm -f depend

-include .depend

# These are the pattern matching rules. In addition to the automatic
# variables used here, the variable $* that matches whatever % stands for
# can be useful in special cases.
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

%: %.c
    $(CC) $(CFLAGS) -o $@ $<

clean:
    rm -f .depend *.o

.PHONY: clean depend
```

Makefile

Integrated Development Environments

Change
source code

Recompute
types,
test results,
binaries, etc.

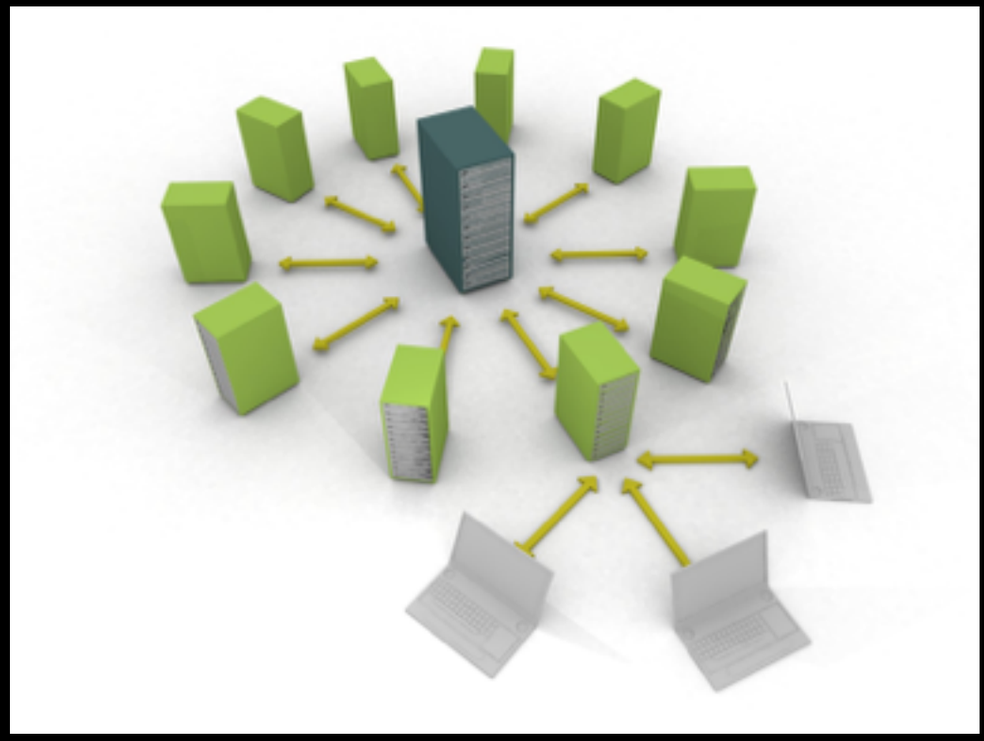


Eclipse

Database Systems

Change tables

Recompute queries and views



Web Browsers

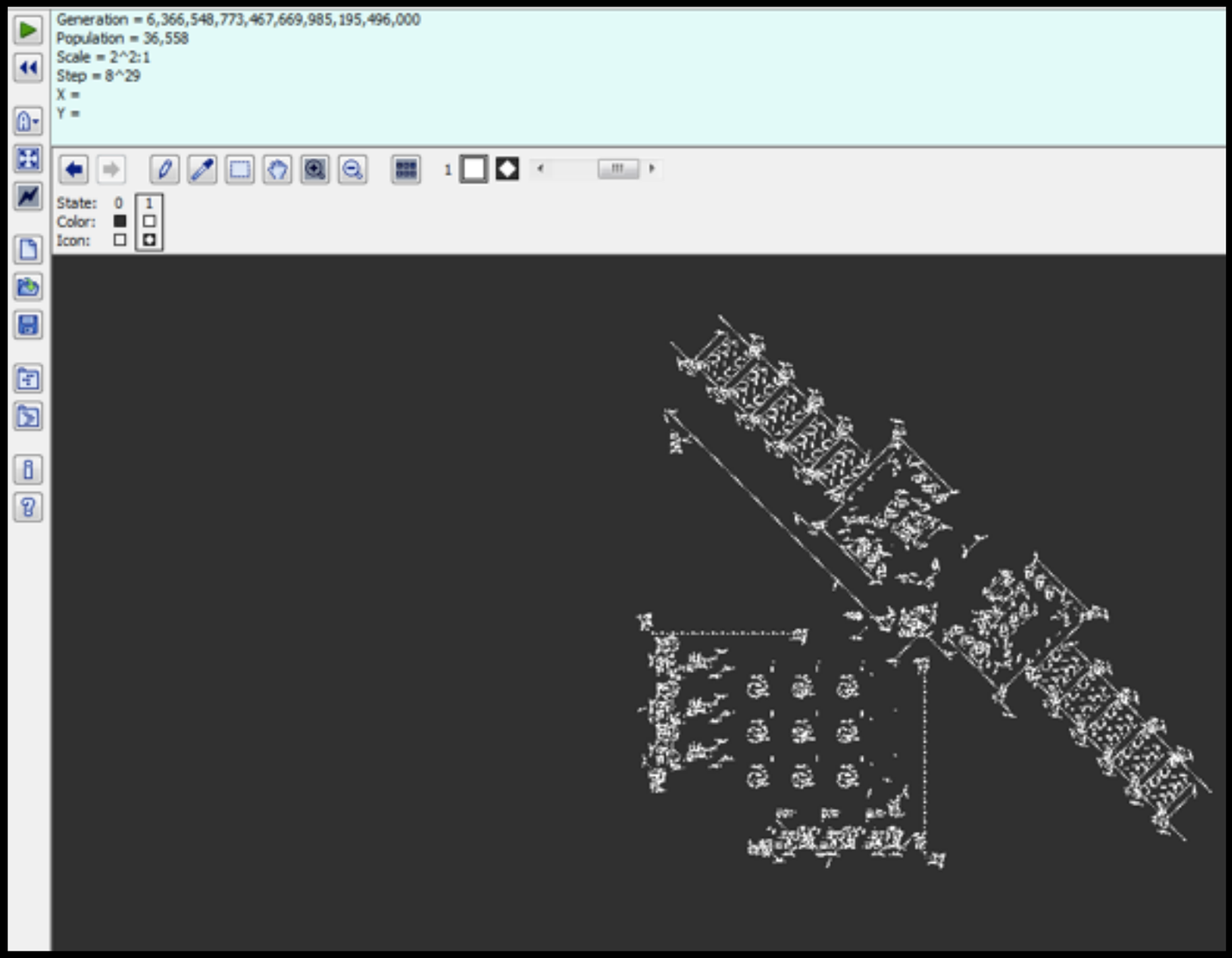
Change
CSS, DOM elements

Recompute
rendered page

Games, Simulations

Change
game state

Recompute
next game
state



Game of Life
(running a Turing Machine)

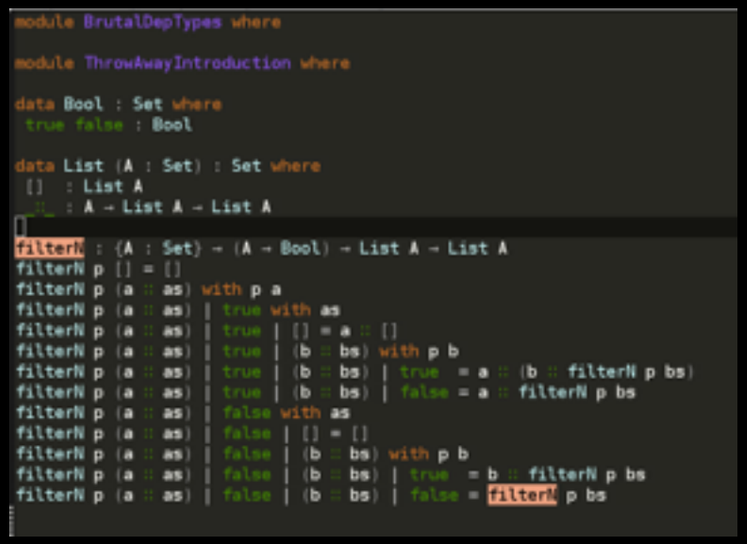
Proof Assistants

Change
definitions,
proof script,
program

Recompute
proof objects

Idris

Isabelle



Agda

Coq

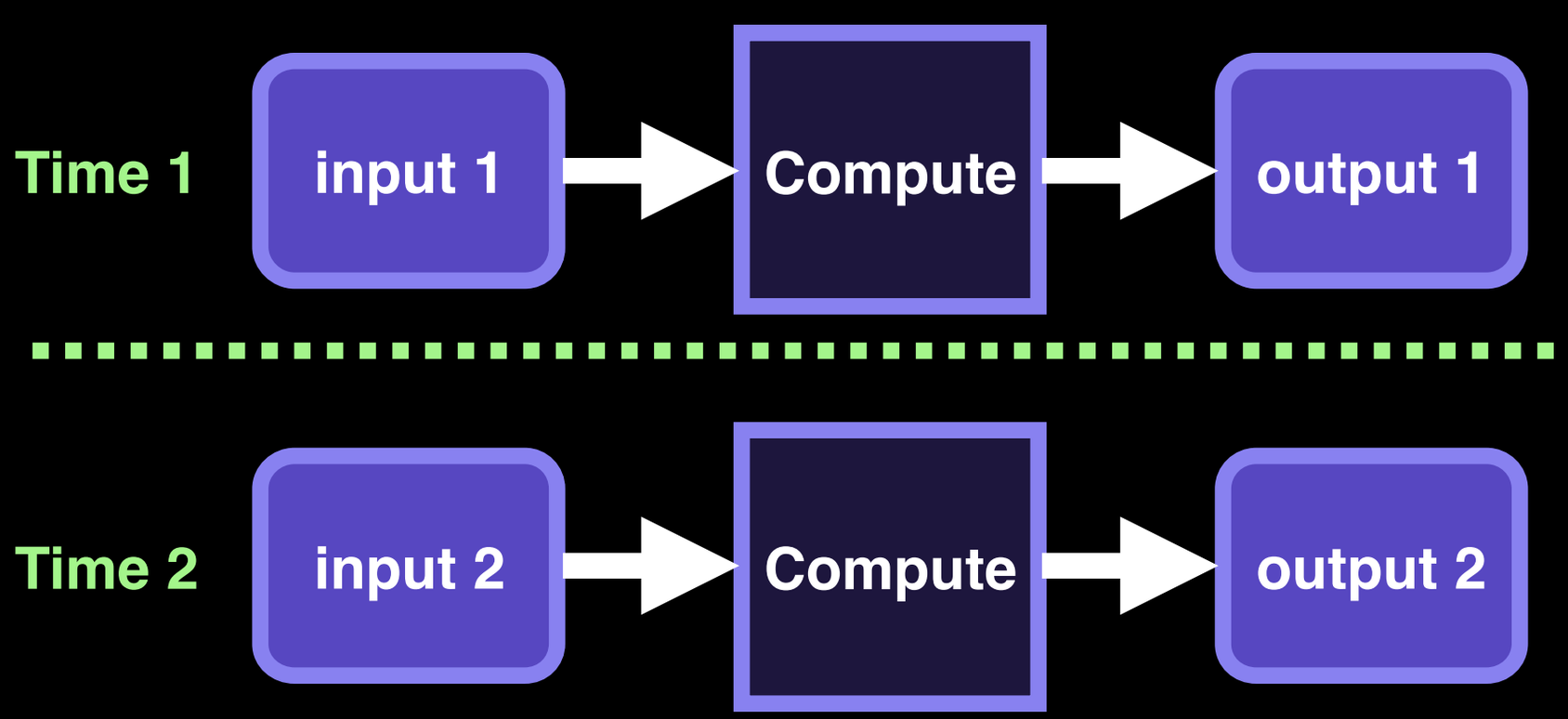
Grand Question: Is there a **Unified, Systematic Approach?**

Application Domain	Input	Compute	Output
spread sheets	Data, Formula	**evaluate formula**	Plots, Charts
build systems	Source Files	**compile**	Binaries, Test Results
databases	Tables	**evaluate query**	Tables
web browsers	HTML, CSS	**render**	Rendering
games, simulations	System State	**advance state**	System State
proof assistants	Proof Script	**proof search**	Proof Object

Incremental Computation
Towards a **Systematic Solution**

As a **programmer**, I want to:

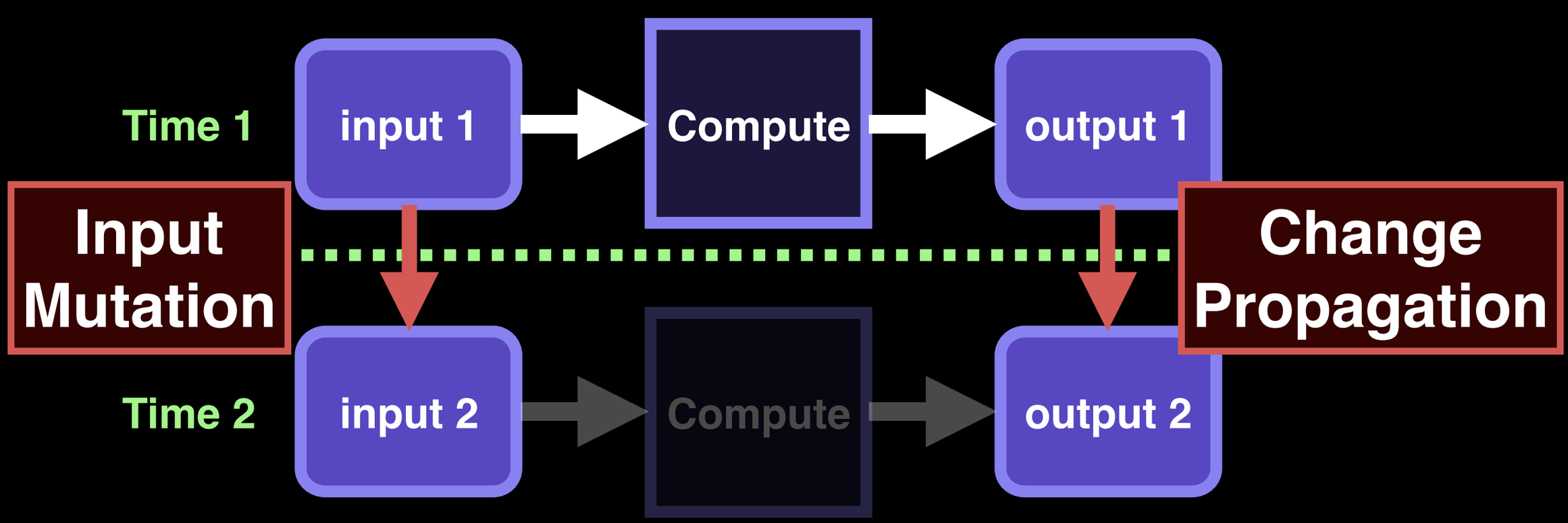
- **Program** the **from-scratch algorithm**
- **Systematically derive** the **incremental algorithm**



Incremental Computation
Towards a **Systematic Solution**

As a **programmer**, I want to:

- **Program** the **from-scratch algorithm**
- **Systematically derive** the **incremental algorithm**



Incremental Computation
Towards a **Systematic Solution**

As a **programmer**, I want to:
- **Program** the **from-scratch algorithm**
- **Systematically derive** the **incremental algorithm**

Simple: Think about **from-scratch algorithm**
Correct: Derived from **from-scratch algorithm**
Efficient: Compare to **from-scratch algorithm**

Incremental Computation
Towards a **Systematic Solution**

As a **programmer**, I want to:
- **Program** the **from-scratch algorithm**
- **Systematically derive** the **incremental algorithm**

Simple: Think about **from-scratch algorithm**
Correct: Derived from **from-scratch algorithm**
Efficient: Compare to **from-scratch algorithm**

Incremental Computation
Towards a **Systematic Solution**

As a **programmer**, I want to:
- **Program** the **from-scratch algorithm**
- **Systematically derive** the **incremental algorithm**

Simple: Think about **from-scratch algorithm**
Correct: Derived from **from-scratch algorithm**
Efficient: Compare to **from-scratch algorithm**

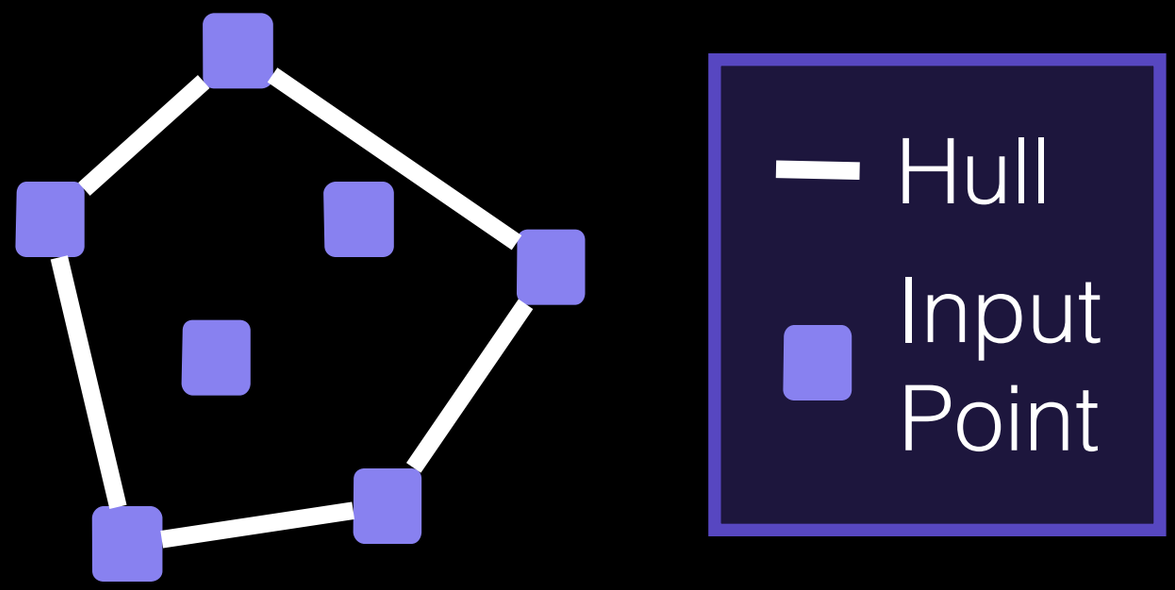
Incremental Computation
Towards a **Systematic Solution**

As a **programmer**, I want to:
- **Program** the **from-scratch algorithm**
- **Systematically derive** the **incremental algorithm**

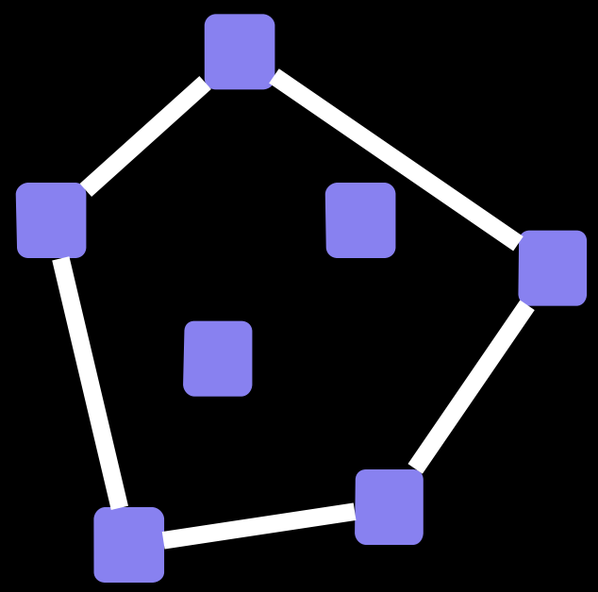
Simple: Think about **from-scratch algorithm**
Correct: Derived from **from-scratch algorithm**
Efficient: Compare to **from-scratch algorithm**

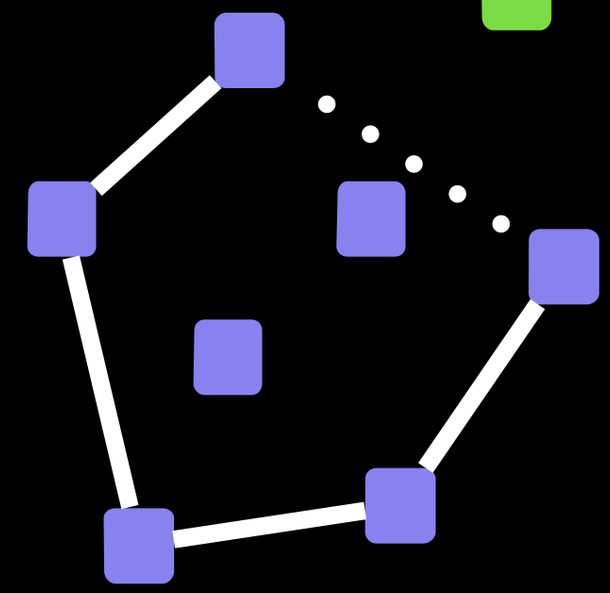
Adapton achieves these goals
through **programming language design**

Algorithms Example: Incremental Convex Hull



Algorithms Example:
Incremental Convex Hull



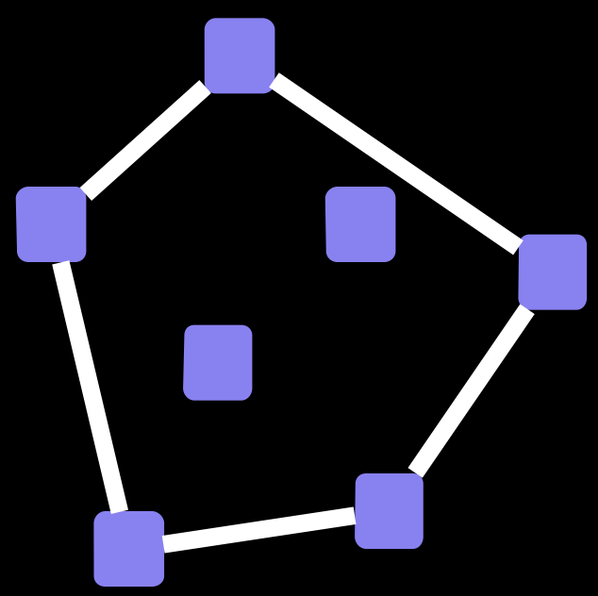


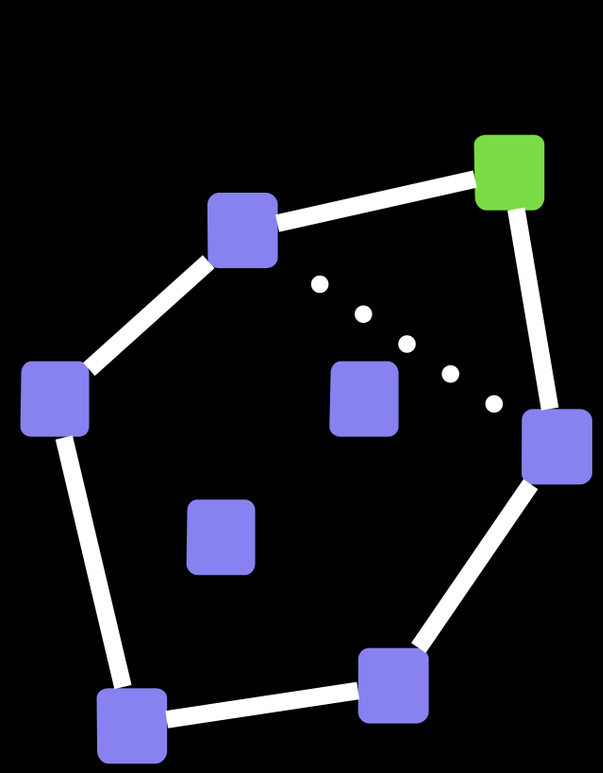
Insert Point

Hull

Input Point

Algorithms Example:
Incremental Convex Hull





Hull

Input
Point

Incremental Convex Hull

Convex Hull Applications:
computational-geometry, pattern recognition, image processing, statistics, geographic information systems, game theory, program analysis.

- **Fundamental Comp Sci Problem (“2D sort”)**
- **Custom solutions are complex!**

2002 PhD Dissertation
Riko Jacob

Incremental Convex Hull

Convex Hull Applications:
computational-geometry, pattern recognition, image processing, statistics, geographic information systems, game theory, program analysis.

- **Fundamental Comp Sci Problem (“2D sort”)**
- **Custom solutions are complex!**

2002 PhD Dissertation
Riko Jacob

Dynamic planar convex hull

Riko Jacob

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Incremental Convex Hull

Convex Hull Applications:
computational-geometry, pattern recognition, image processing, statistics, geographic information systems, game theory, program analysis.

- **Fundamental Comp Sci Problem (“2D sort”)**
- **Custom solutions are complex!**

2002 PhD Dissertation
Riko Jacob

Dynamic planar convex hull

Riko Jacob

PhD Dissertation

Departm
U

Algorithms & Proofs

No Implementation

> 100 pages

Incremental Convex Hull

The algorithm reasons about changes:

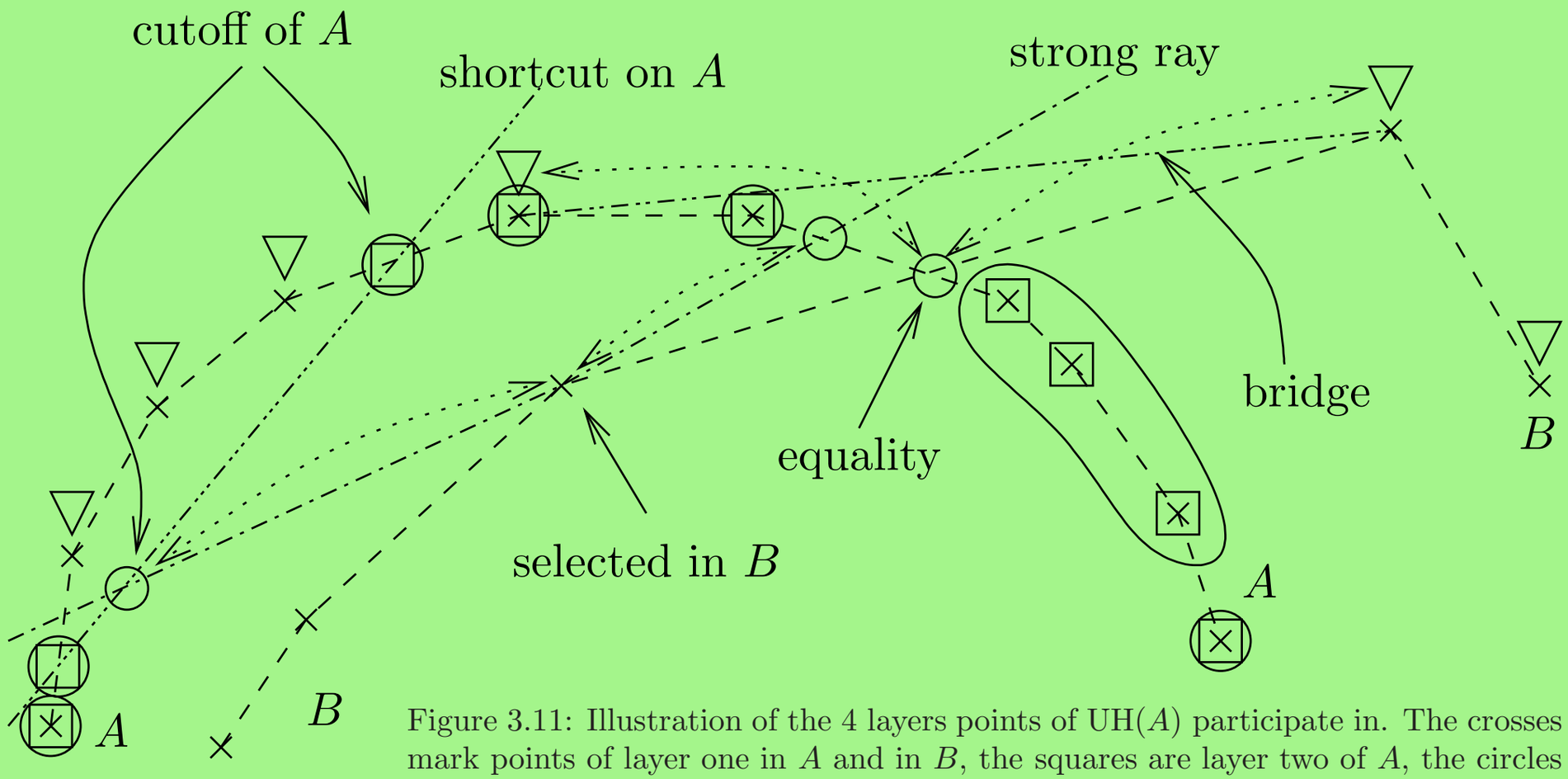


Figure 3.11: Illustration of the 4 layers points of UH(A) participate in. The crosses mark points of layer one in A and in B, the squares are layer two of A, the circles and encircled (in a splitter) points are layer three of A. The triangles indicate layer four. The dotted arrows show pointers aligning A and B

University of Aarhus
Denmark

Incremental Convex Hull

The algorithm reasons about changes:

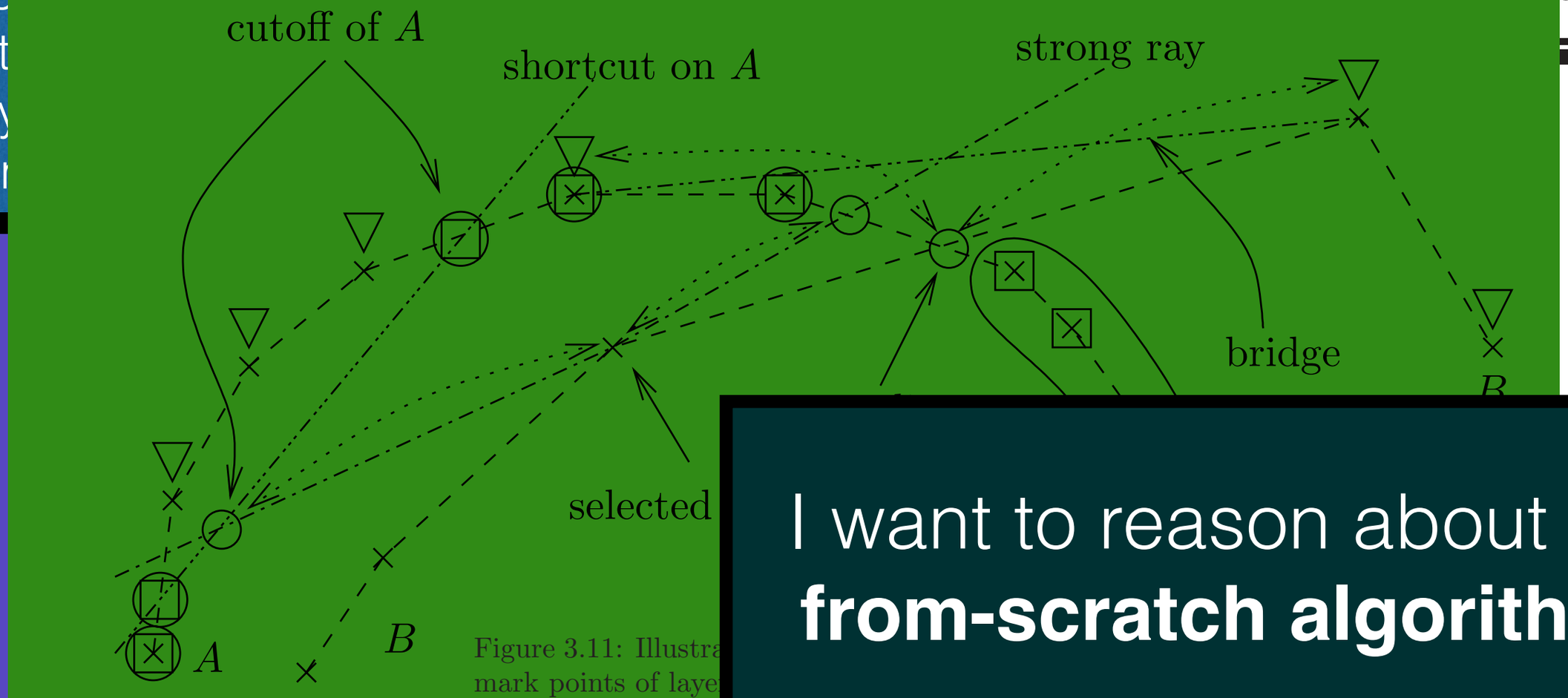


Figure 3.11: Illustra
mark points of laye
and encircled (in a s
four. The dotted ar

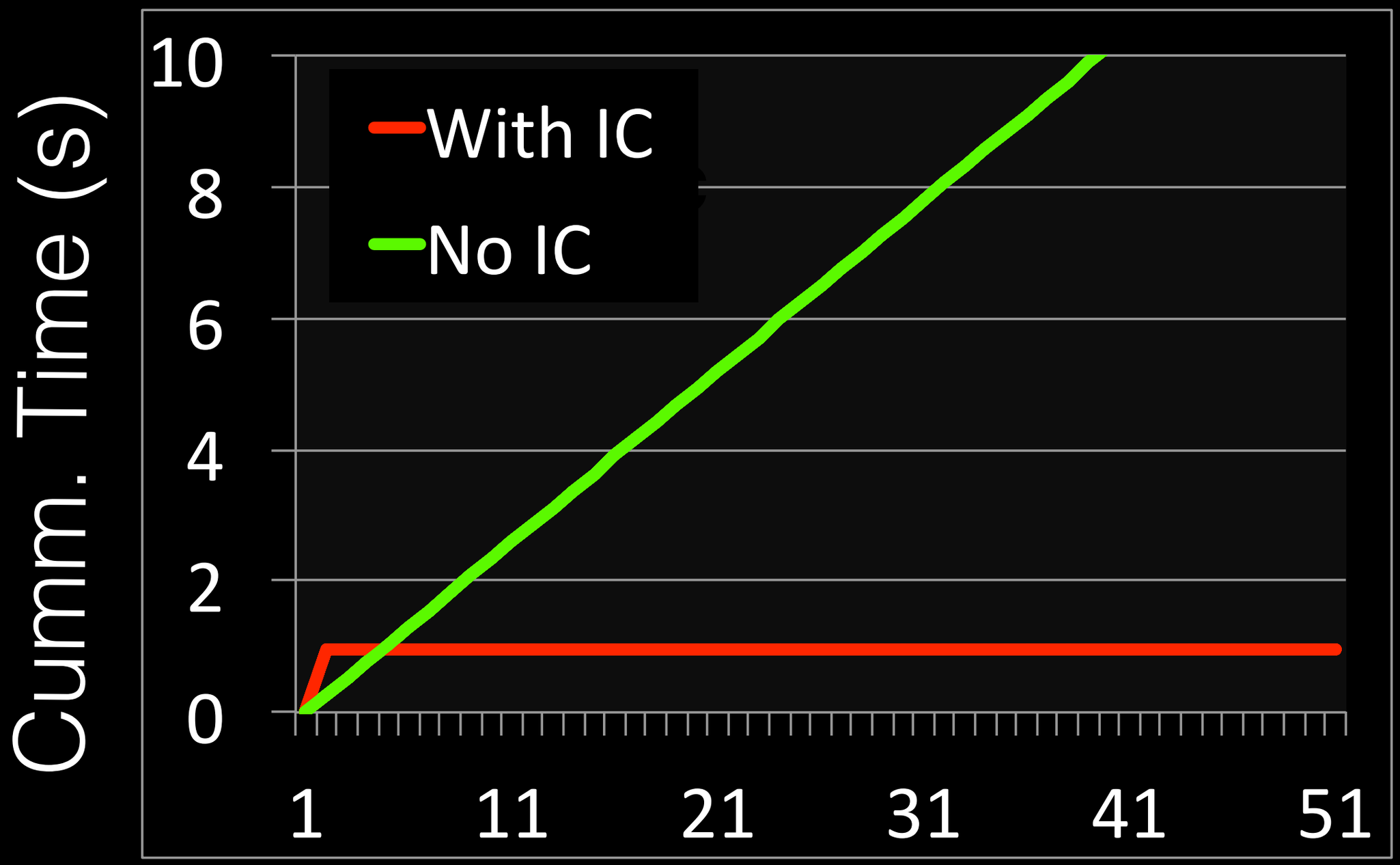
I want to reason about the **from-scratch algorithm!**

How does this relate?

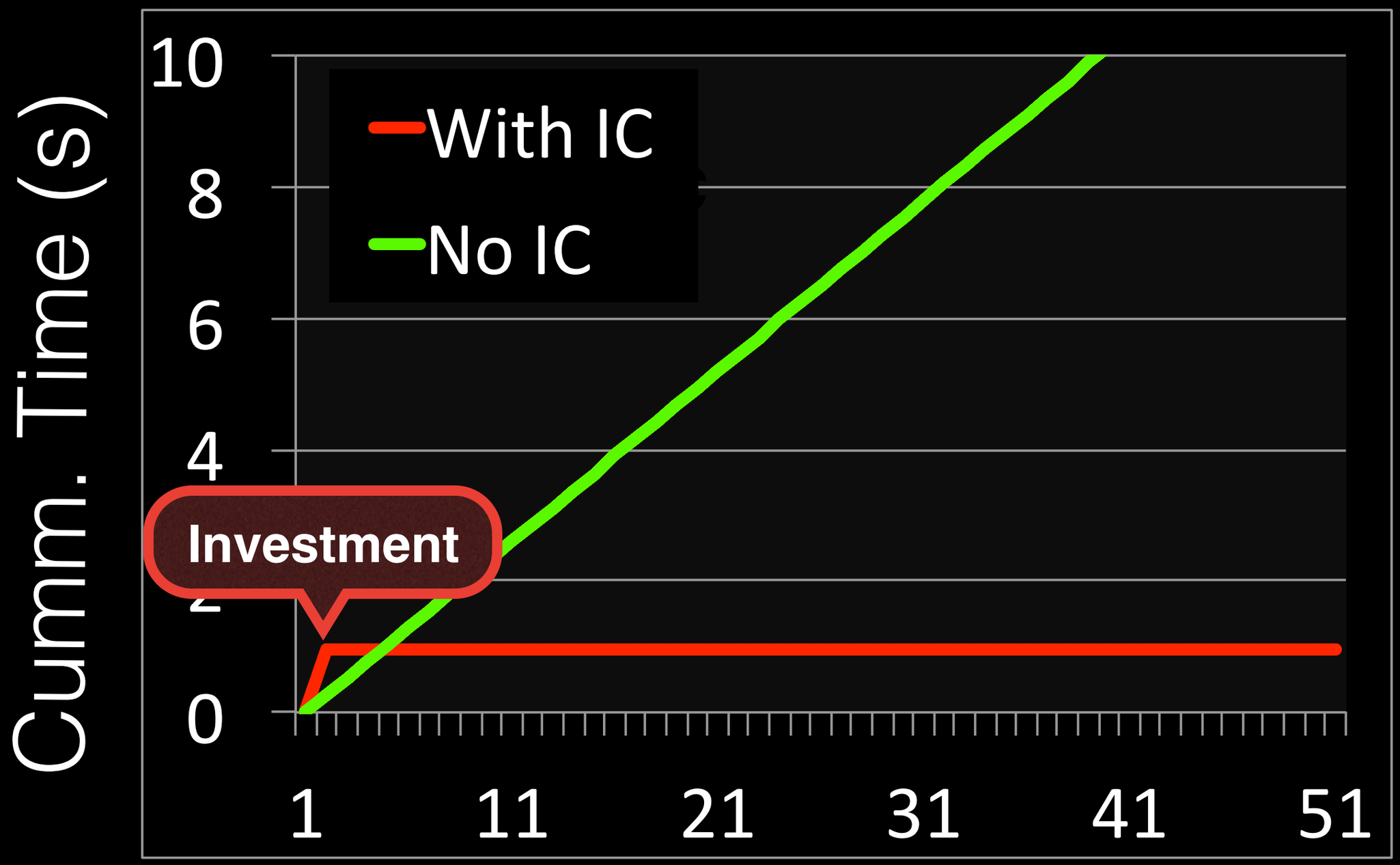
Adapton's Approach

- Write **from-scratch algorithm (QuickHull)**
- Language tools **instrument it** for **incrementality:**
  - **Record execution trace**
    (of from-scratch algorithm)
  - When **changes occur,**
    selectively reevaluate affected steps

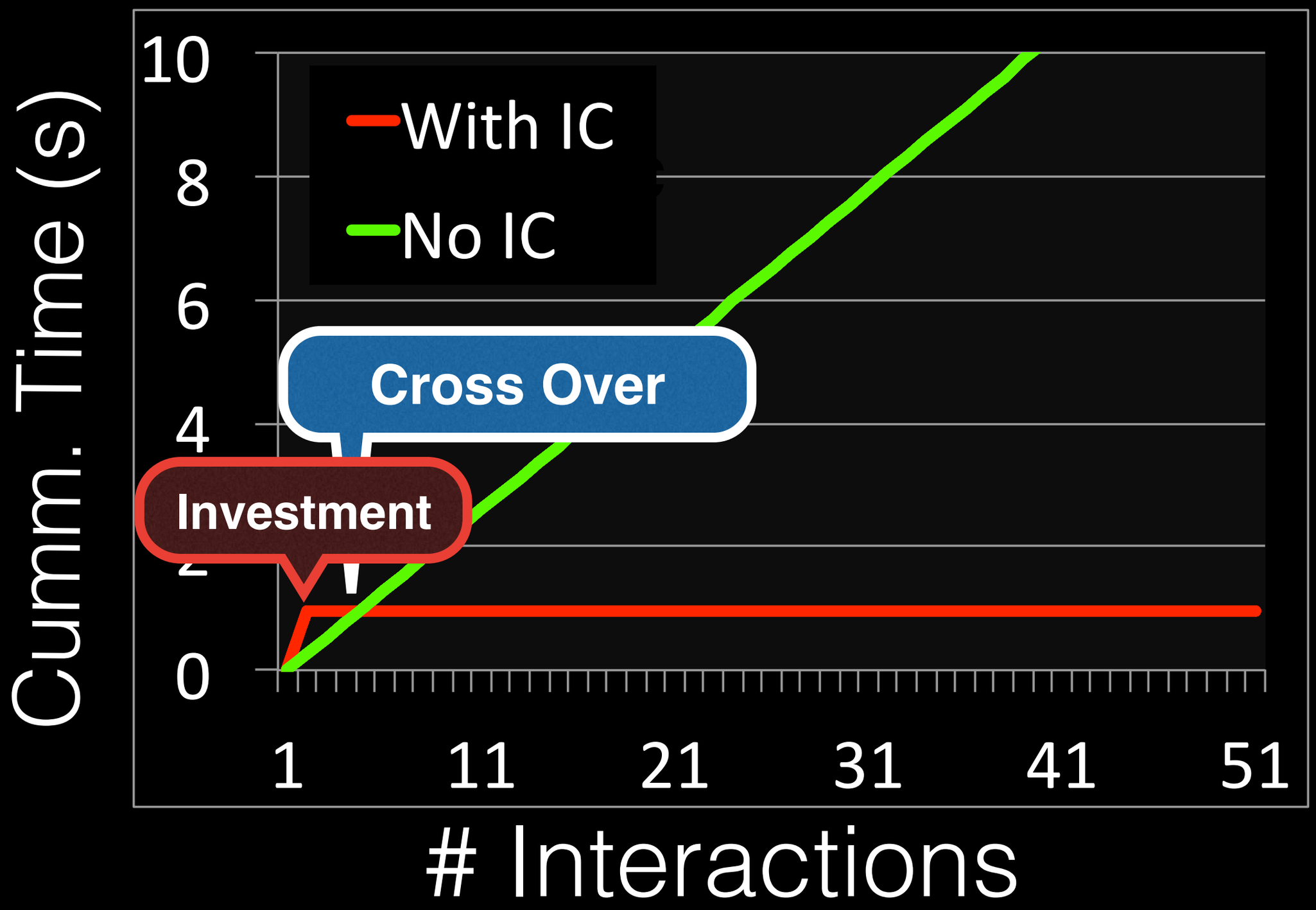
Interaction Time



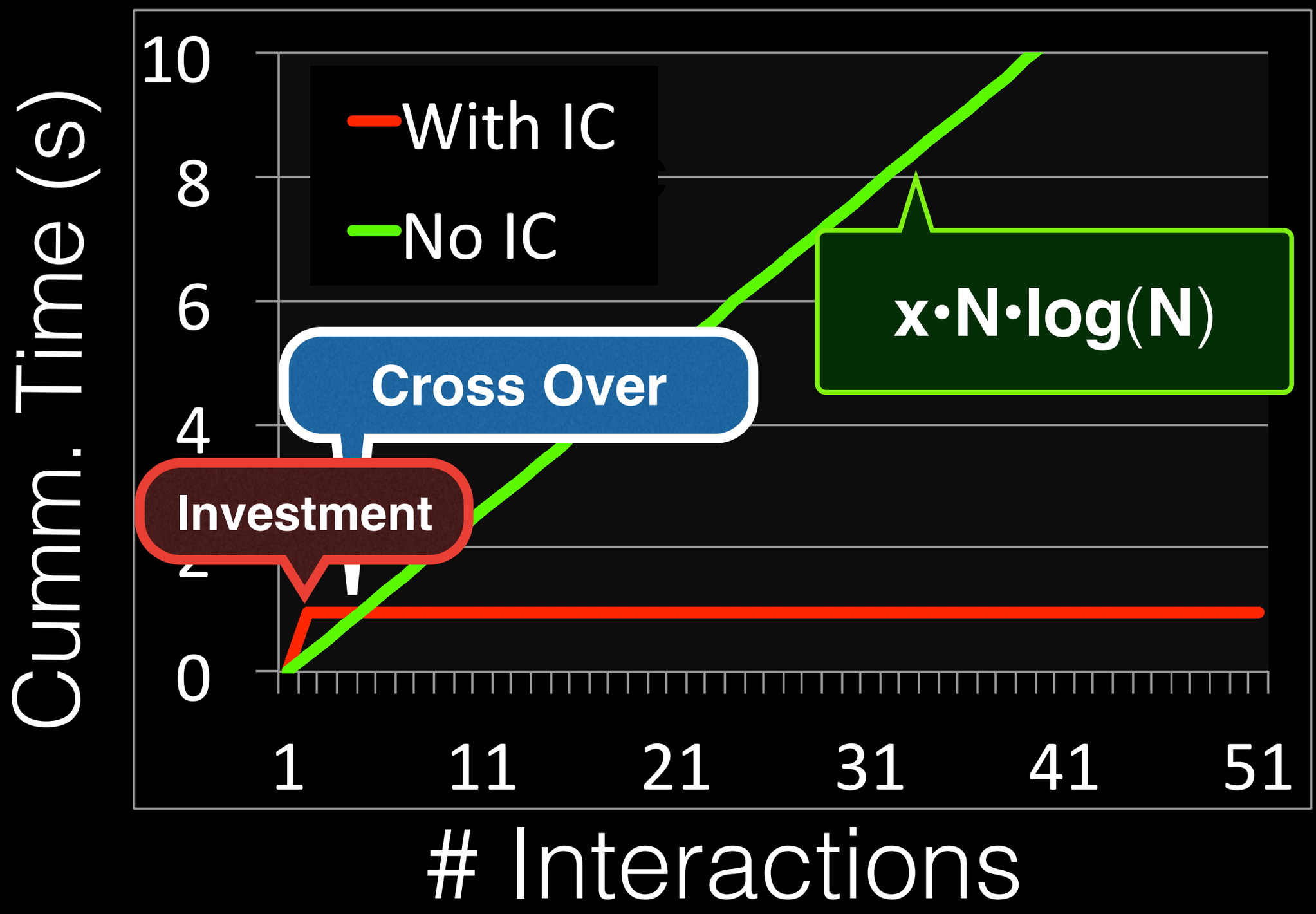
Interaction Time



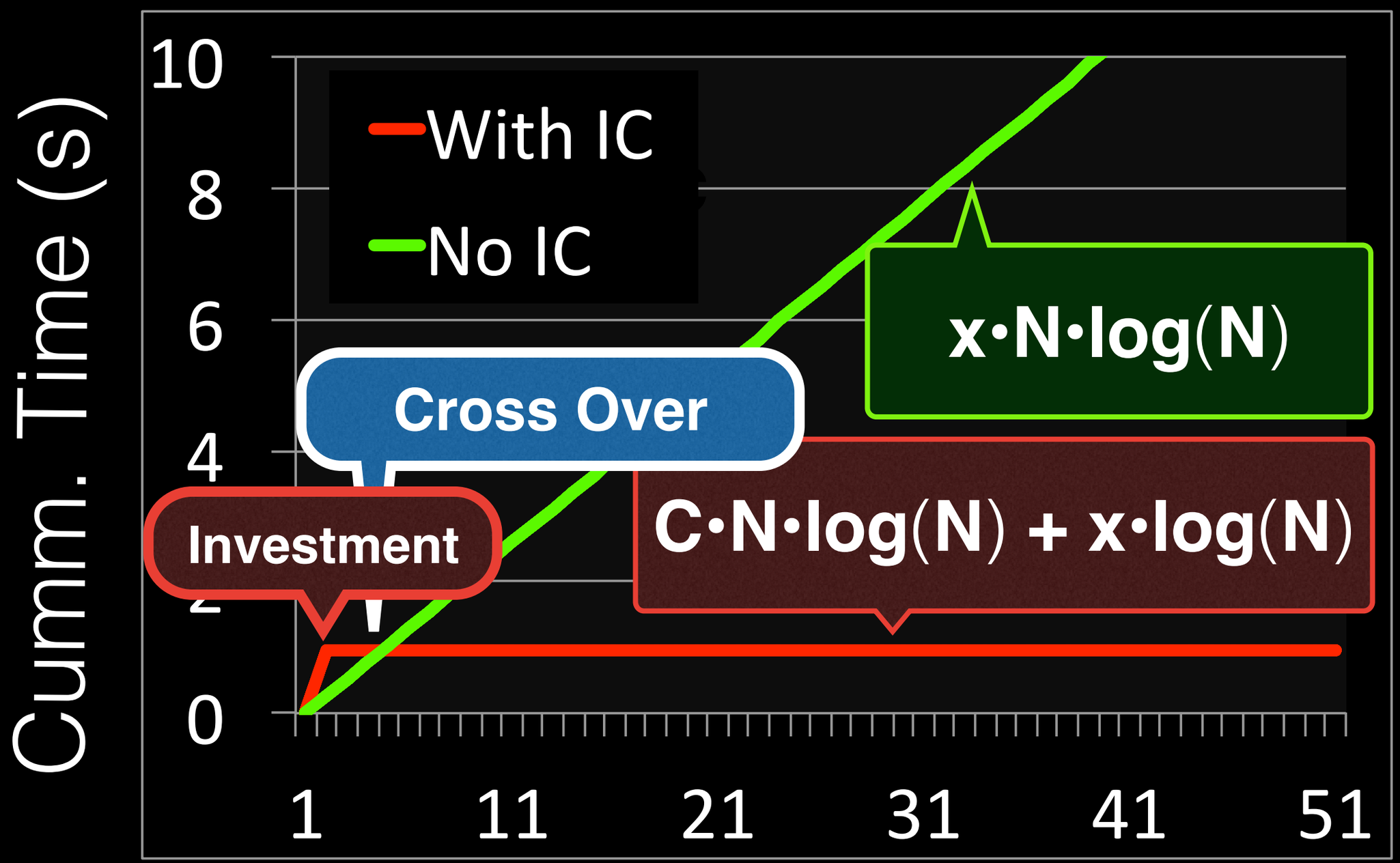
Interaction Time



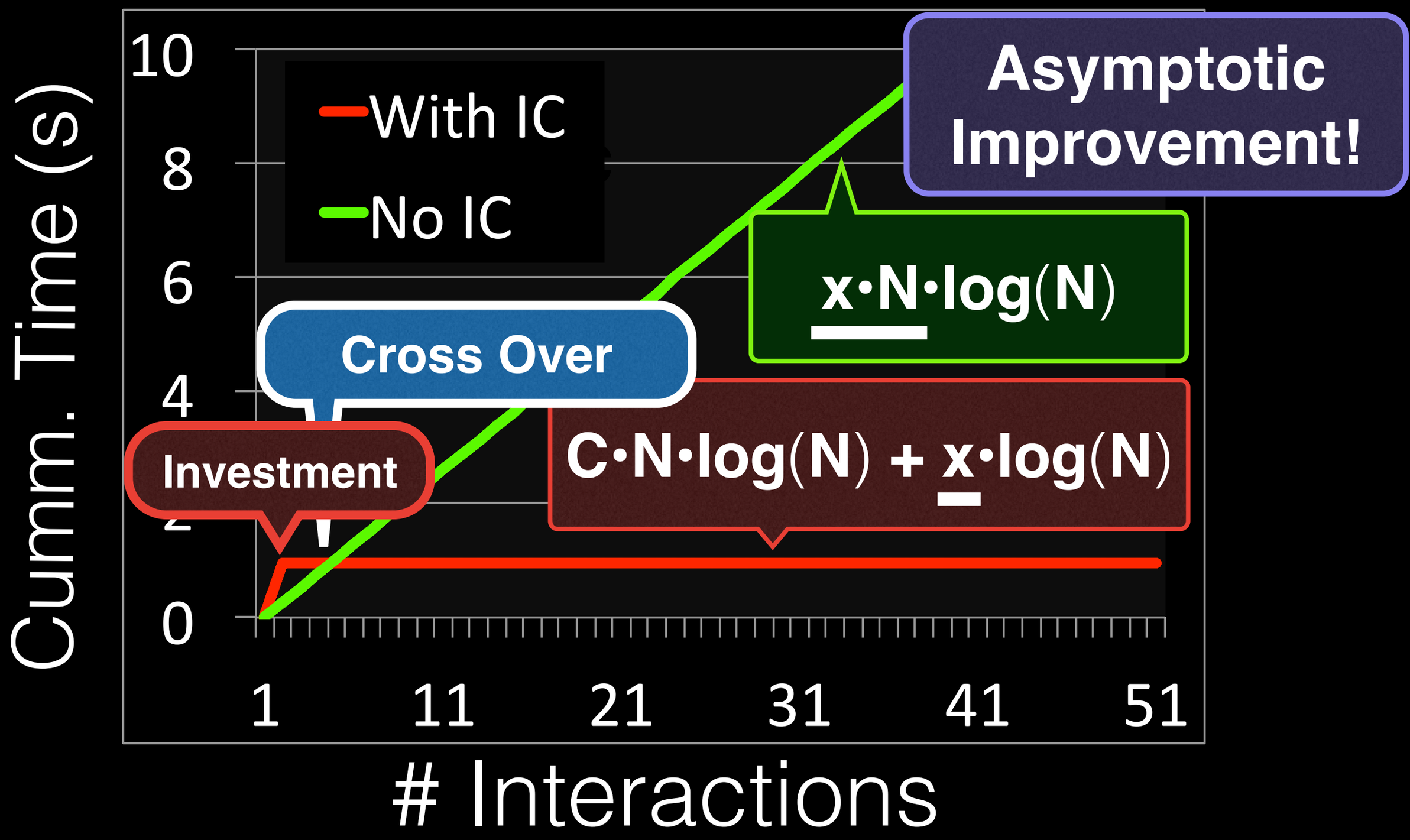
Interaction Time



Interaction Time



Interaction Time



Ad Hoc Approach
$\not\Rightarrow$ Compositionality

PL Approach
$\Rightarrow$ Compositionality

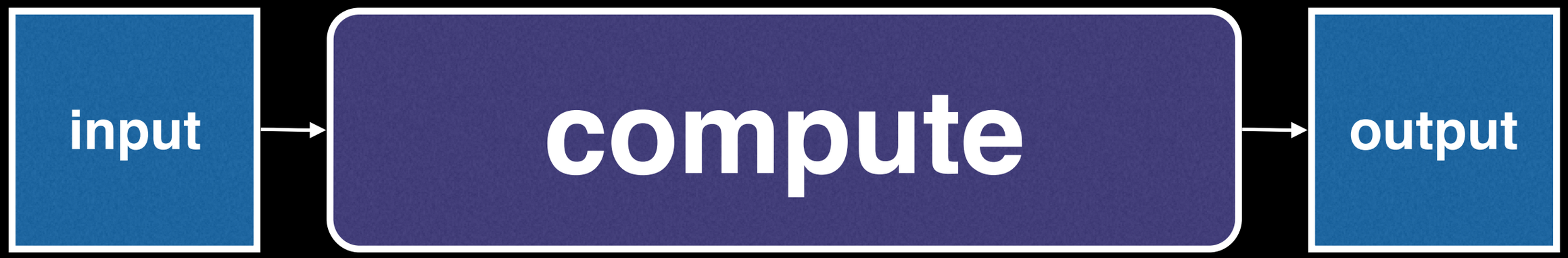
Ad Hoc Approach

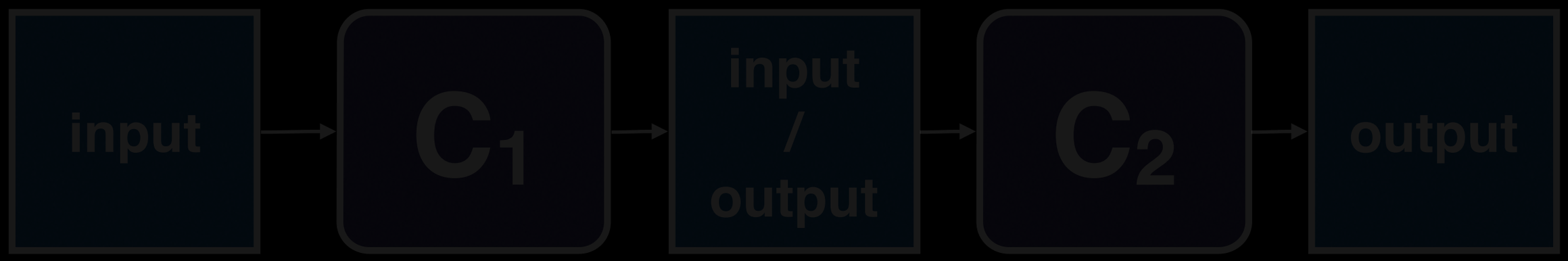
$\not\Rightarrow$ Compositionality

PL Approach

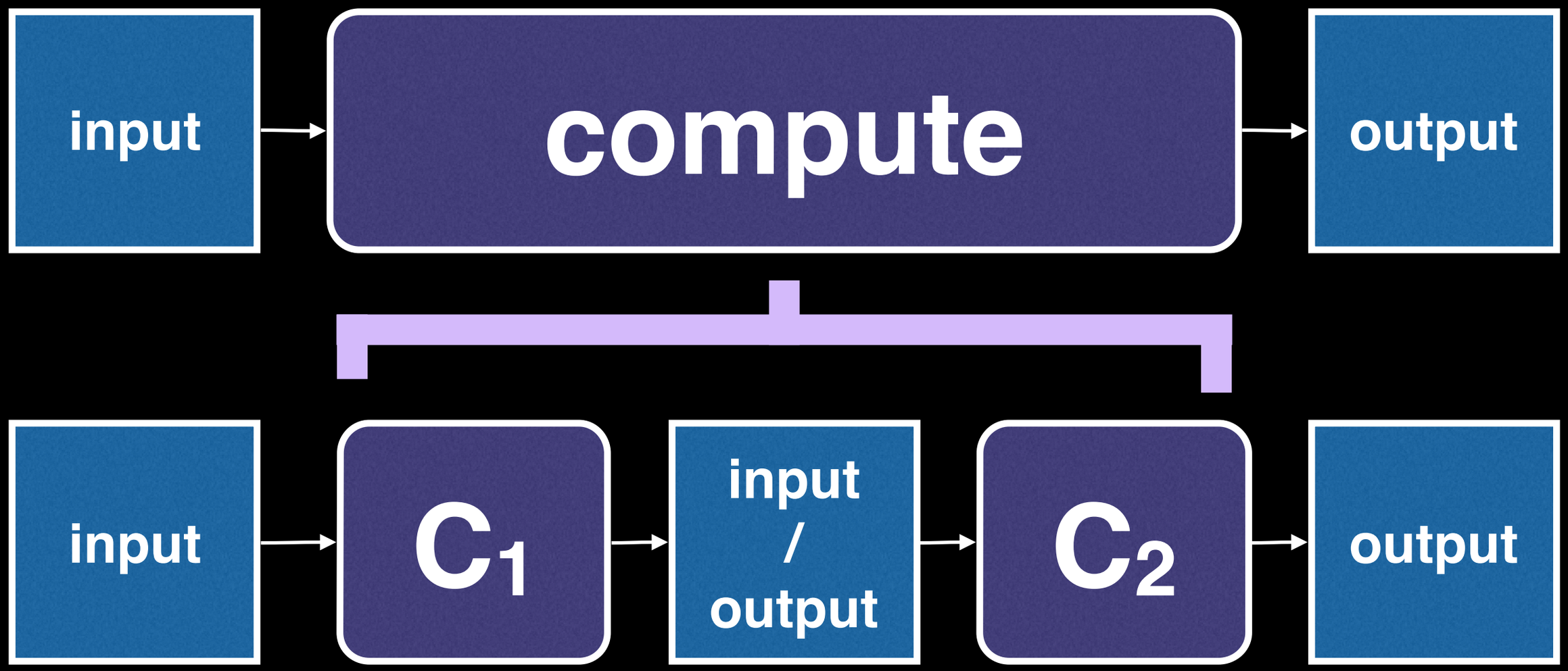
$\Rightarrow$ Compositionality

PL ⇒ “Compositionality"





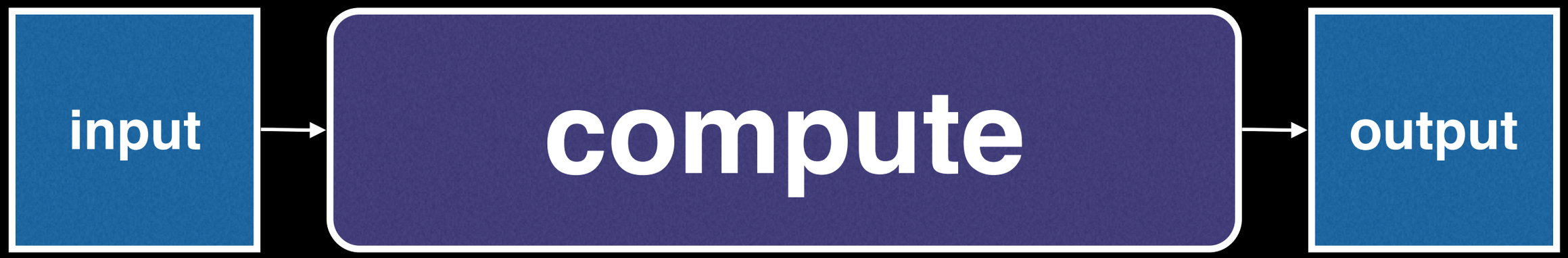
PL ⇒ “Compositionality"



Reverse Polish Calculator
Input-Output Examples

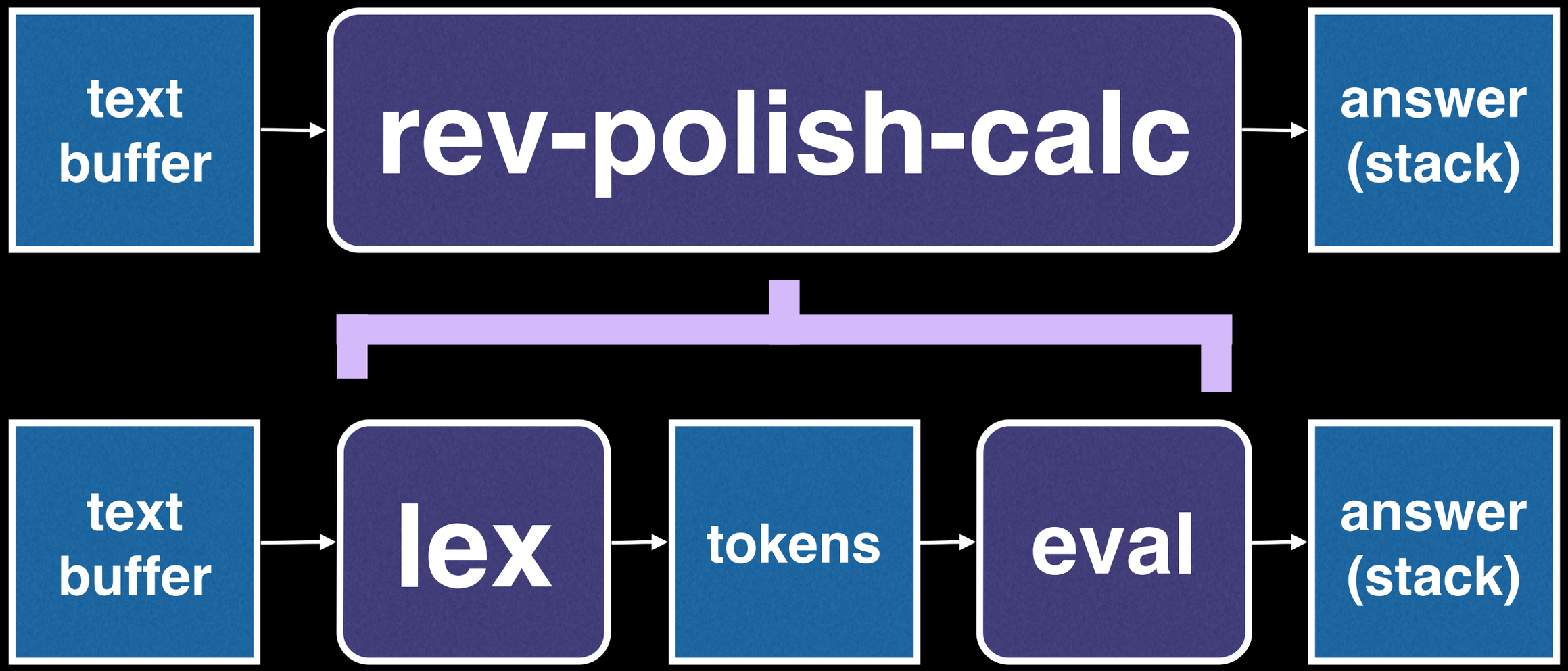
Input: String	**Output:** Stack<Int>
"10 2 +"	**[12]**
"10 2 4 +"	**[6, 10]**
"10 2 3 4 +"	**[7, 2, 10]**
"10 2 + 3 4 +"	**[7, 12]**

PL ⇒ “Compositionality"

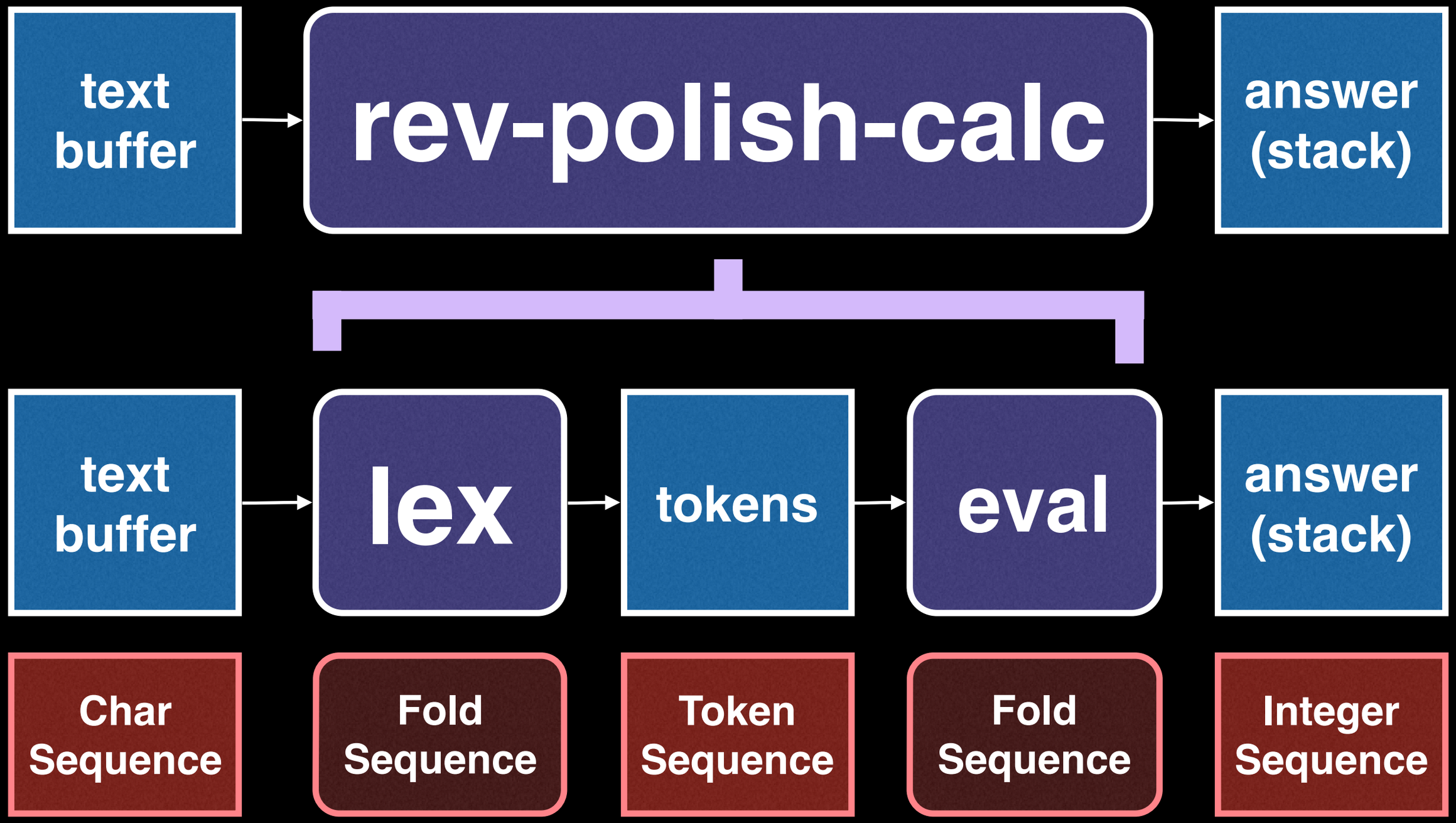


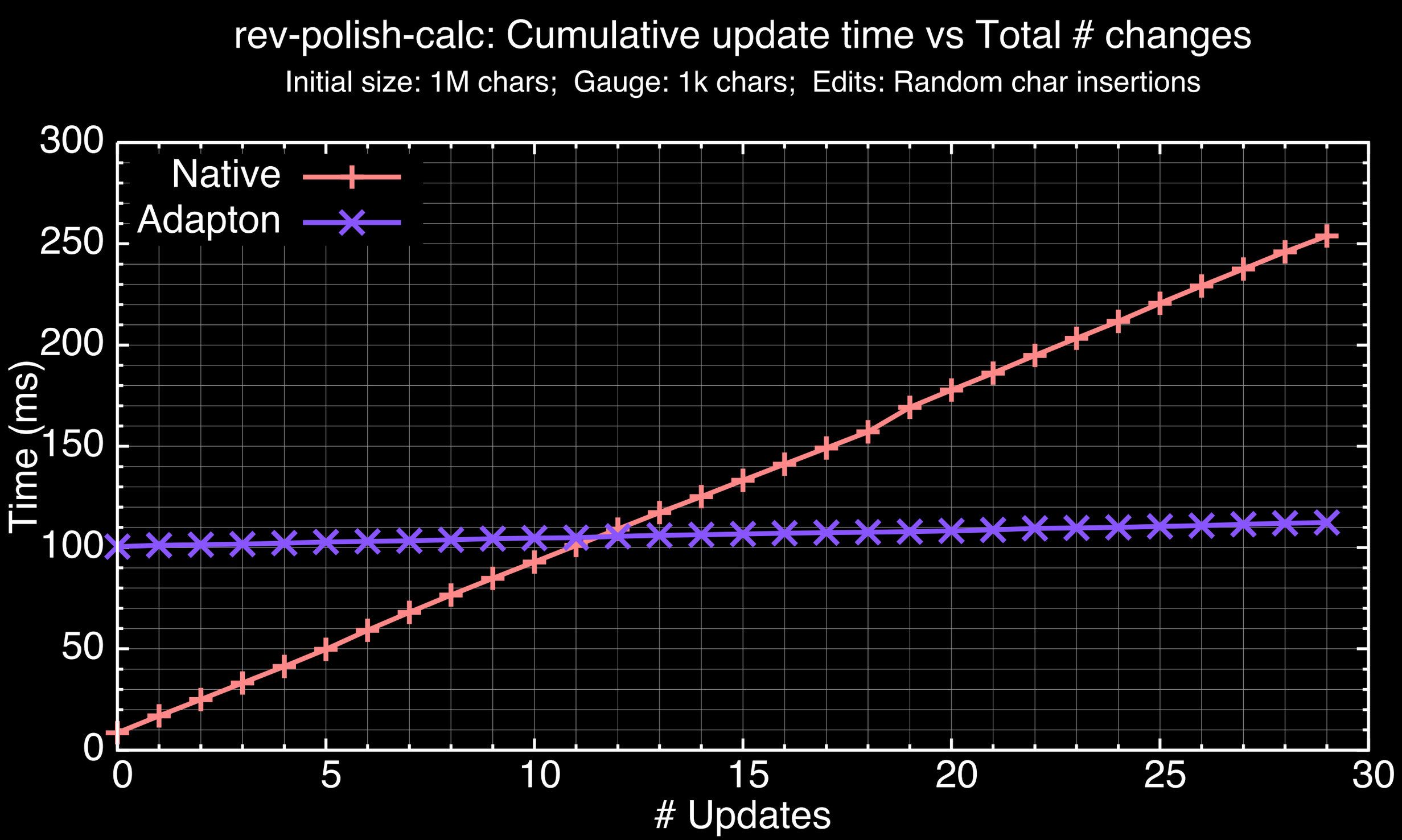
PL ⇒ “Compositionality"

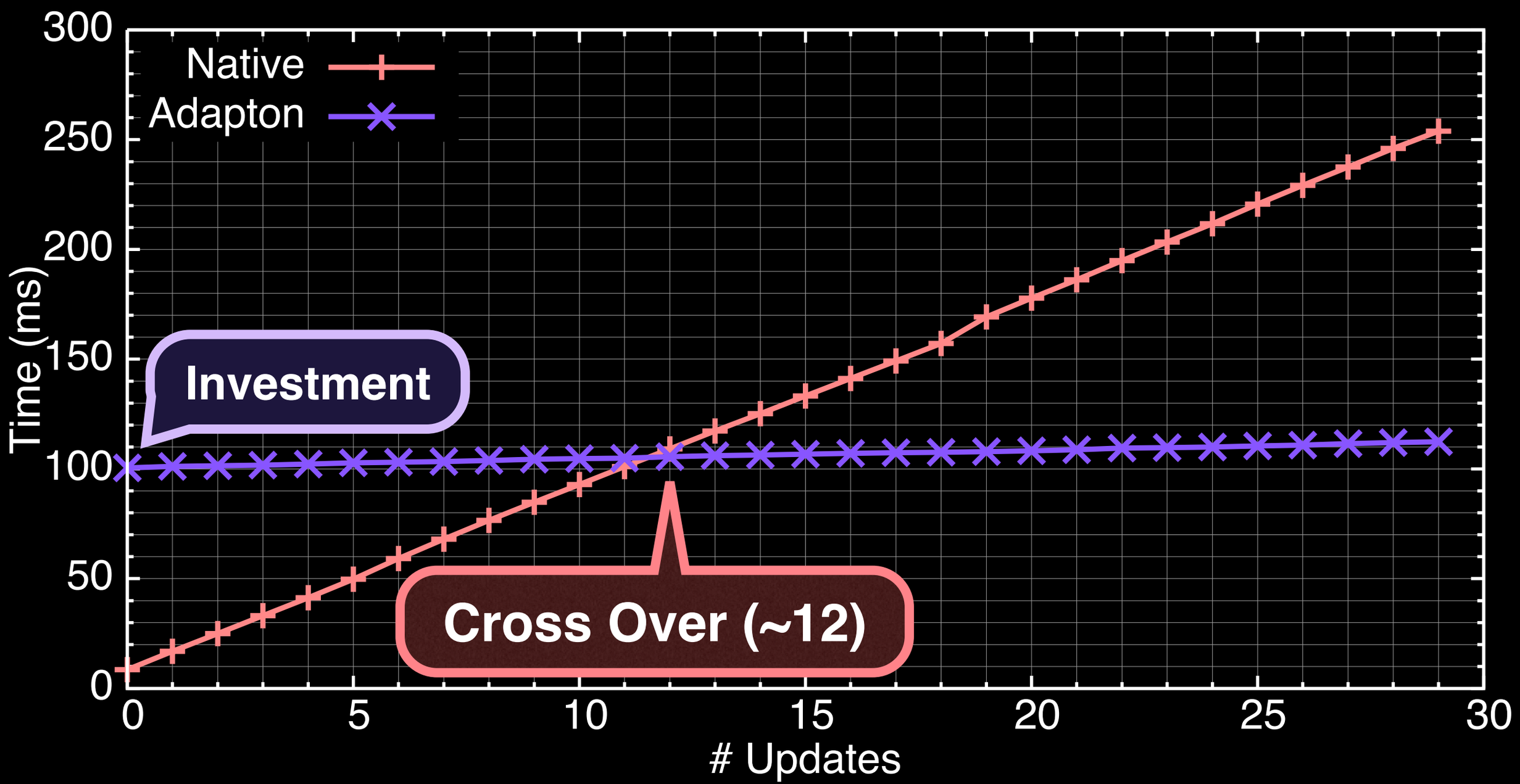
PL ⇒ "Compositionality"

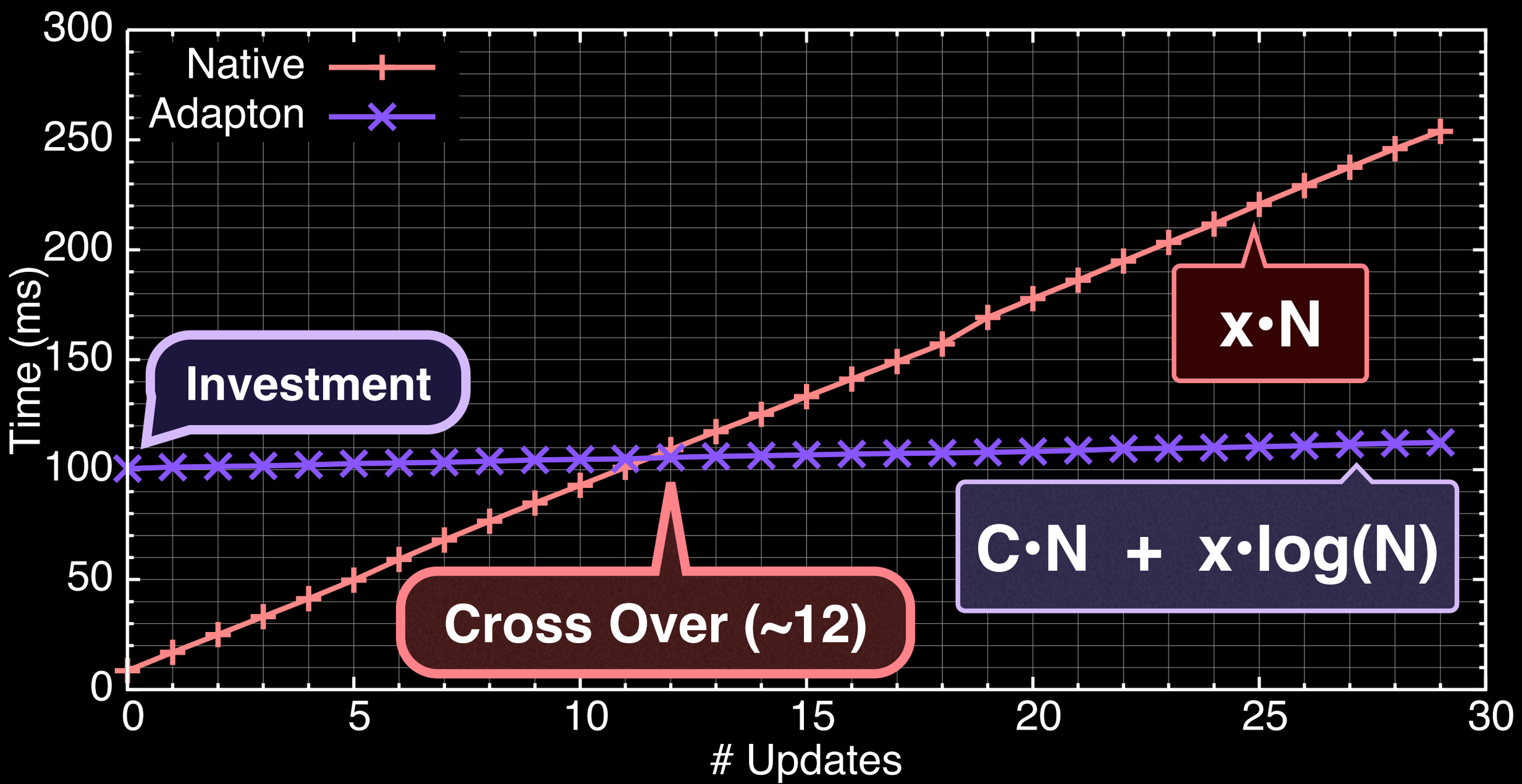


PL ⇒ “Compositionality"




rev-polish-calc: Cumulative update time vs Total # changes
Initial size: 1M chars; Gauge: 1k chars; Edits: Random char insertions
Native
Adapton
Time (ms)
300
250
200
150
100
50
0
0
5
10
15
20
25
30
Updates


rev-polish-calc: Cumulative update time vs Total # changes
Initial size: 1M chars; Gauge: 1k chars; Edits: Random char insertions
Native
Adapton
Investment
Cross Over (~12)
Time (ms)
Updates
300
250
200
150
100
50
0
0
5
10
15
20
25
30


rev-polish-calc: Cumulative update time vs Total # changes
Initial size: 1M chars; Gauge: 1k chars; Edits: Random char insertions
Native
Adapton
Investment
x·N
C·N + x·log(N)
Cross Over (~12)
Time (ms)
Updates
300
250
200
150
100
50
0
0
5
10
15
20
25
30

Inside Adapton

The Demanded Computation Graph (DCG)

num

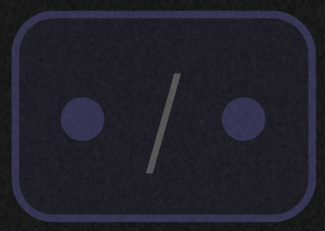
42

den

2

Allocate input cells

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```



check

```
if • == 0
then  None
else Some(•)
```

num

42

den

2

Allocates thunks …

• / • **div**

check

if • == 0
then None
else Some(•)

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

num

42

den

2

• / •

div

check

if • == 0
then None
else Some(•)

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
]
```

Demand output, mutate inputs

num

42

den

2

• / •

div

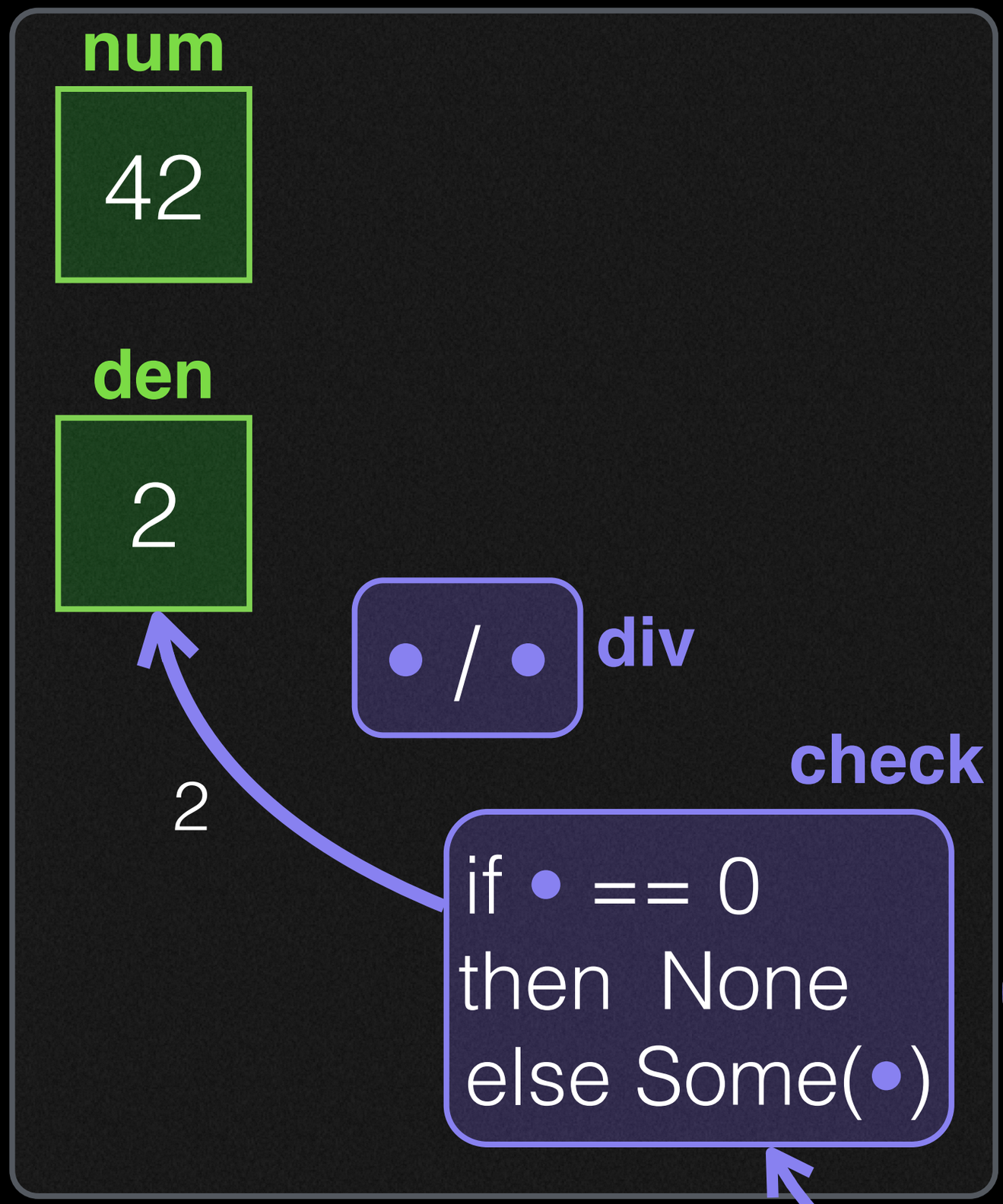
check

if • == 0
then None
else Some(•)

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

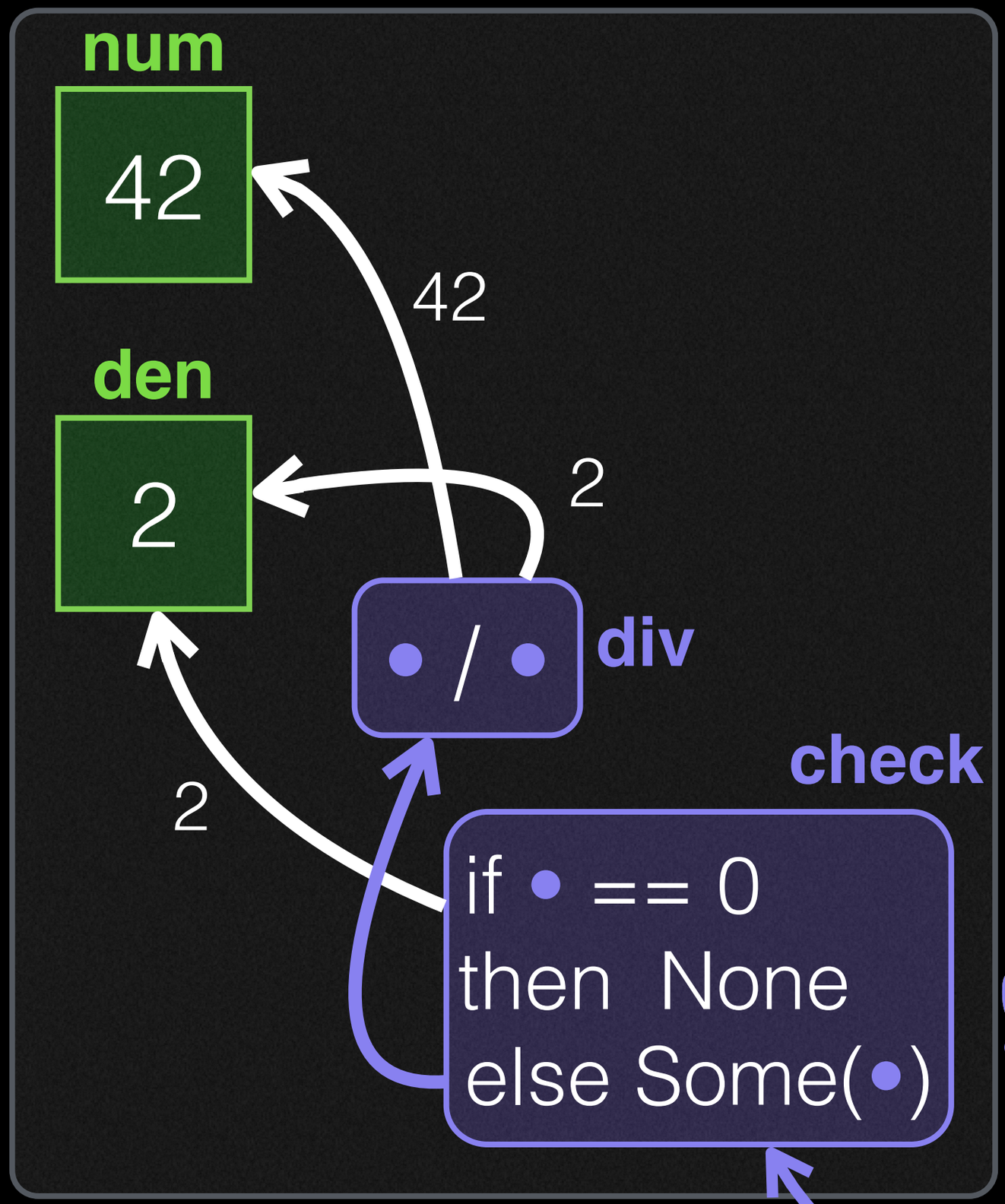
root

From-scratch evaluation



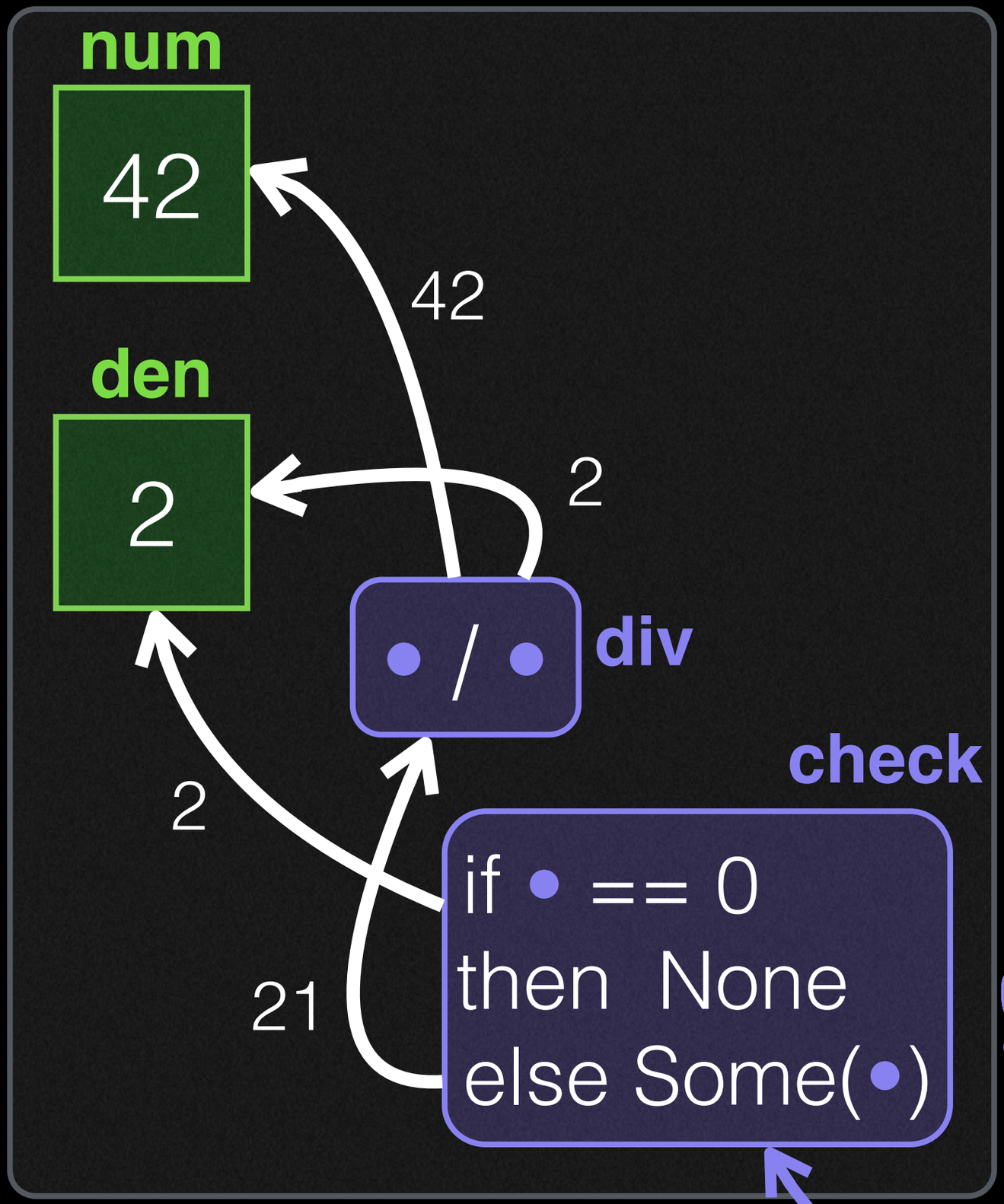

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

From-scratch evaluation



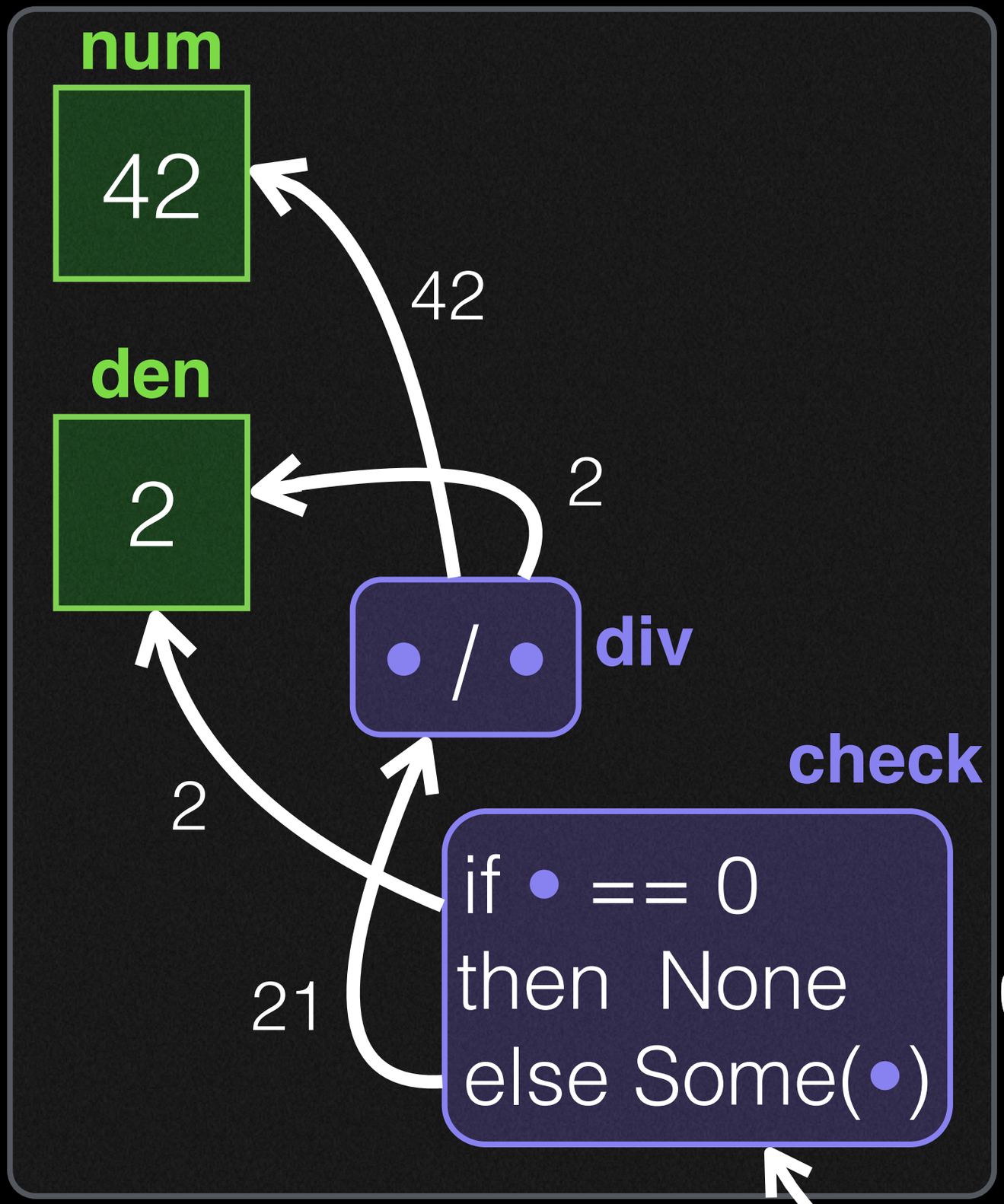

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

From-scratch evaluation



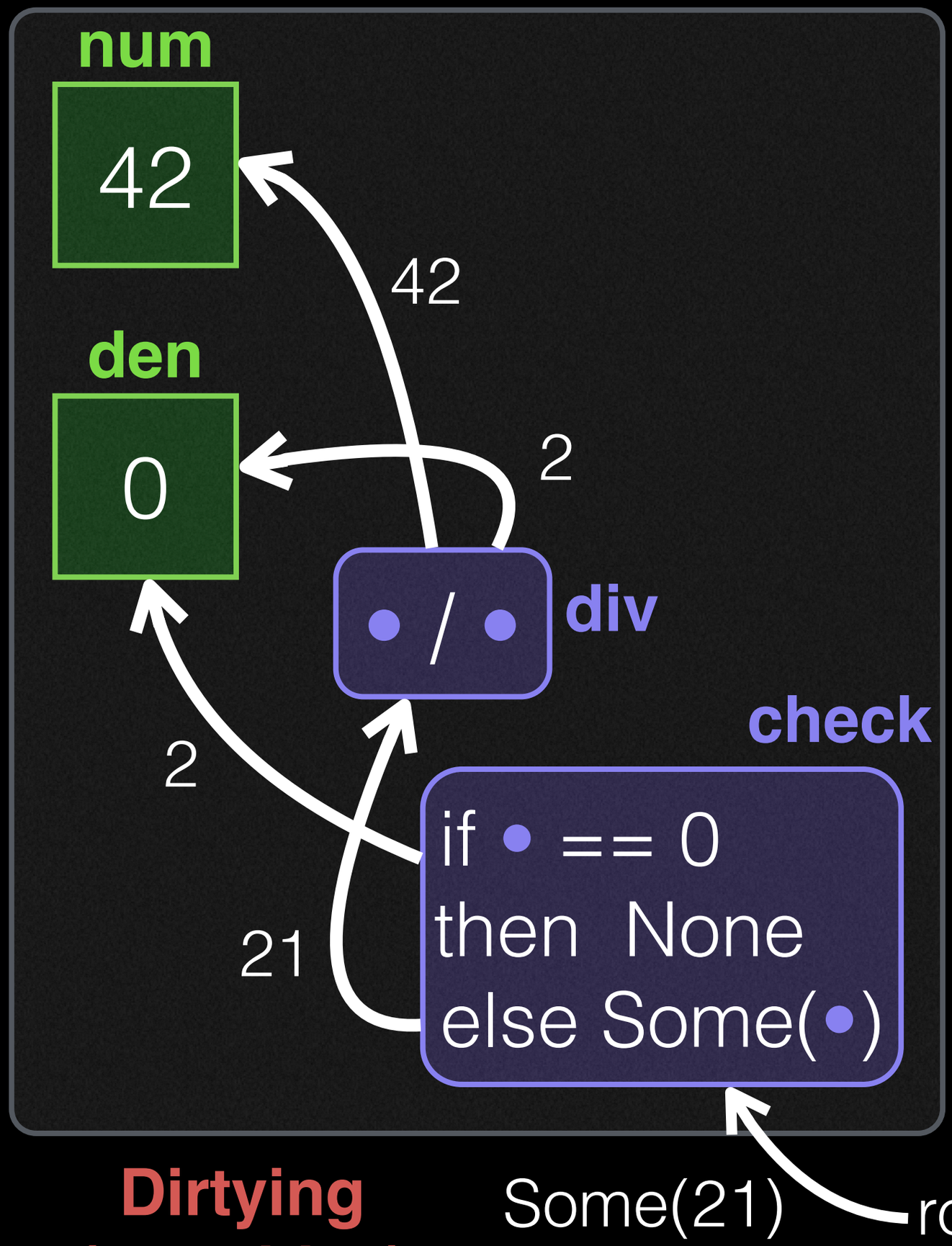

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

From-scratch evaluation

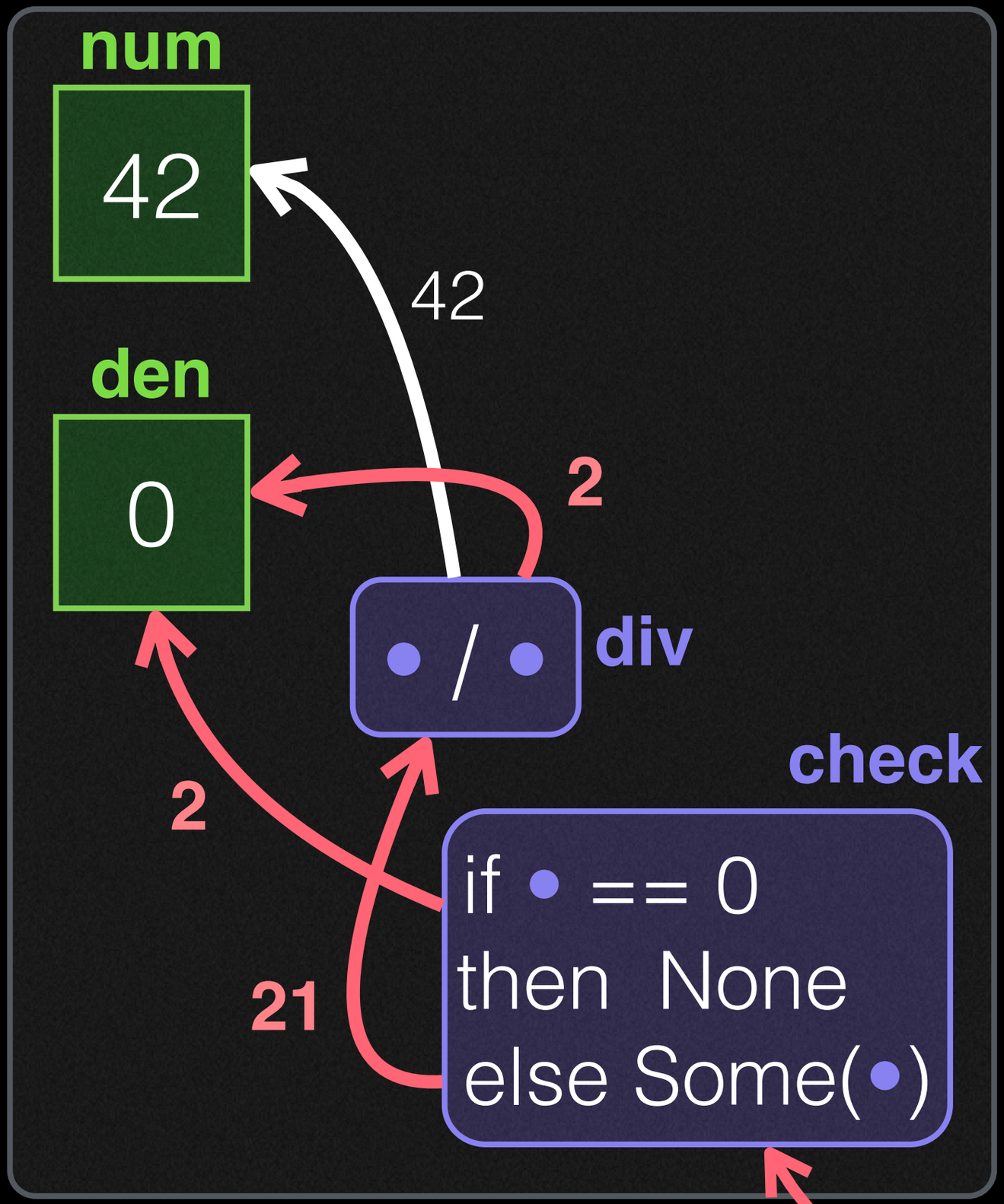



```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

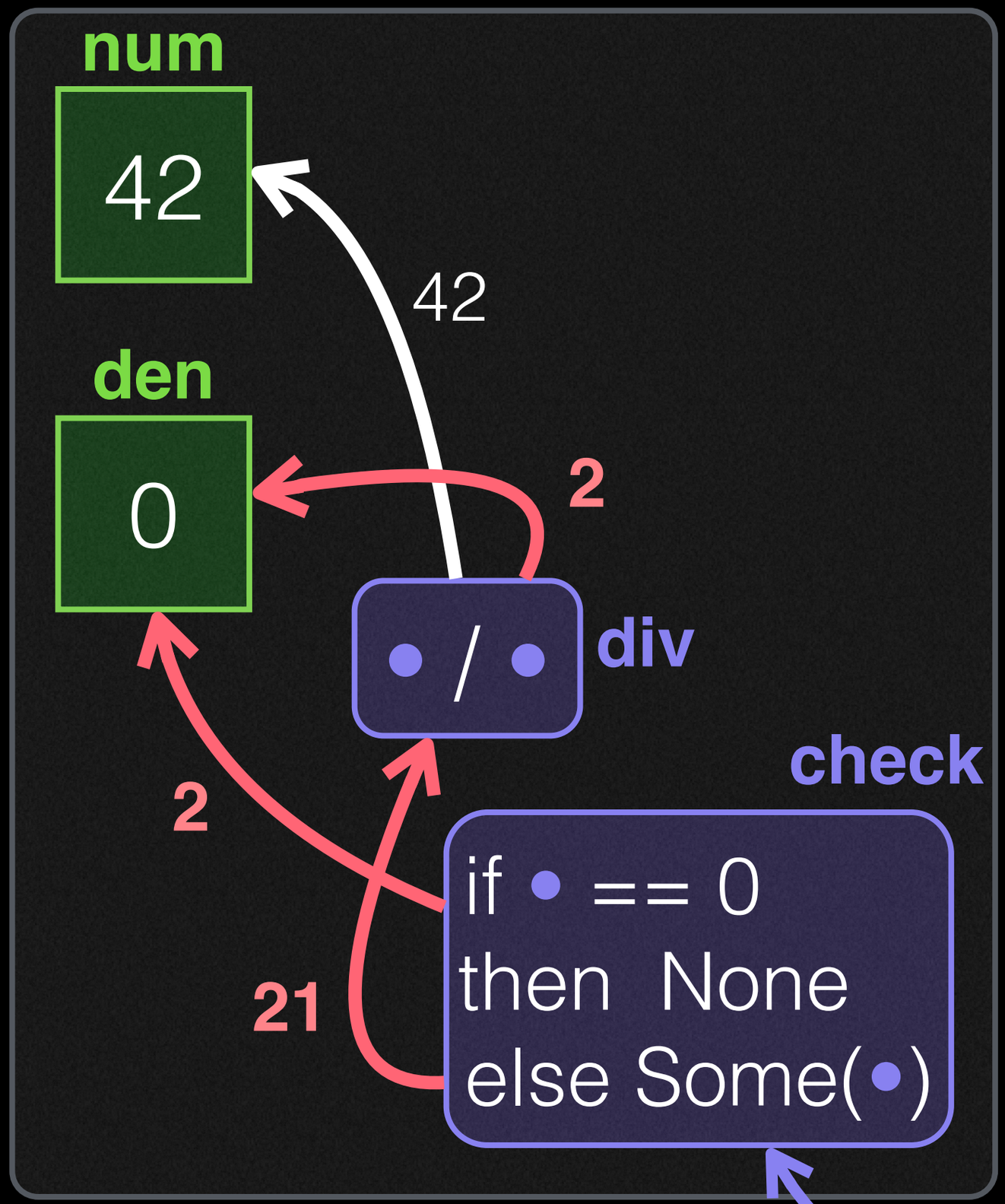
From-scratch evaluation




```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```



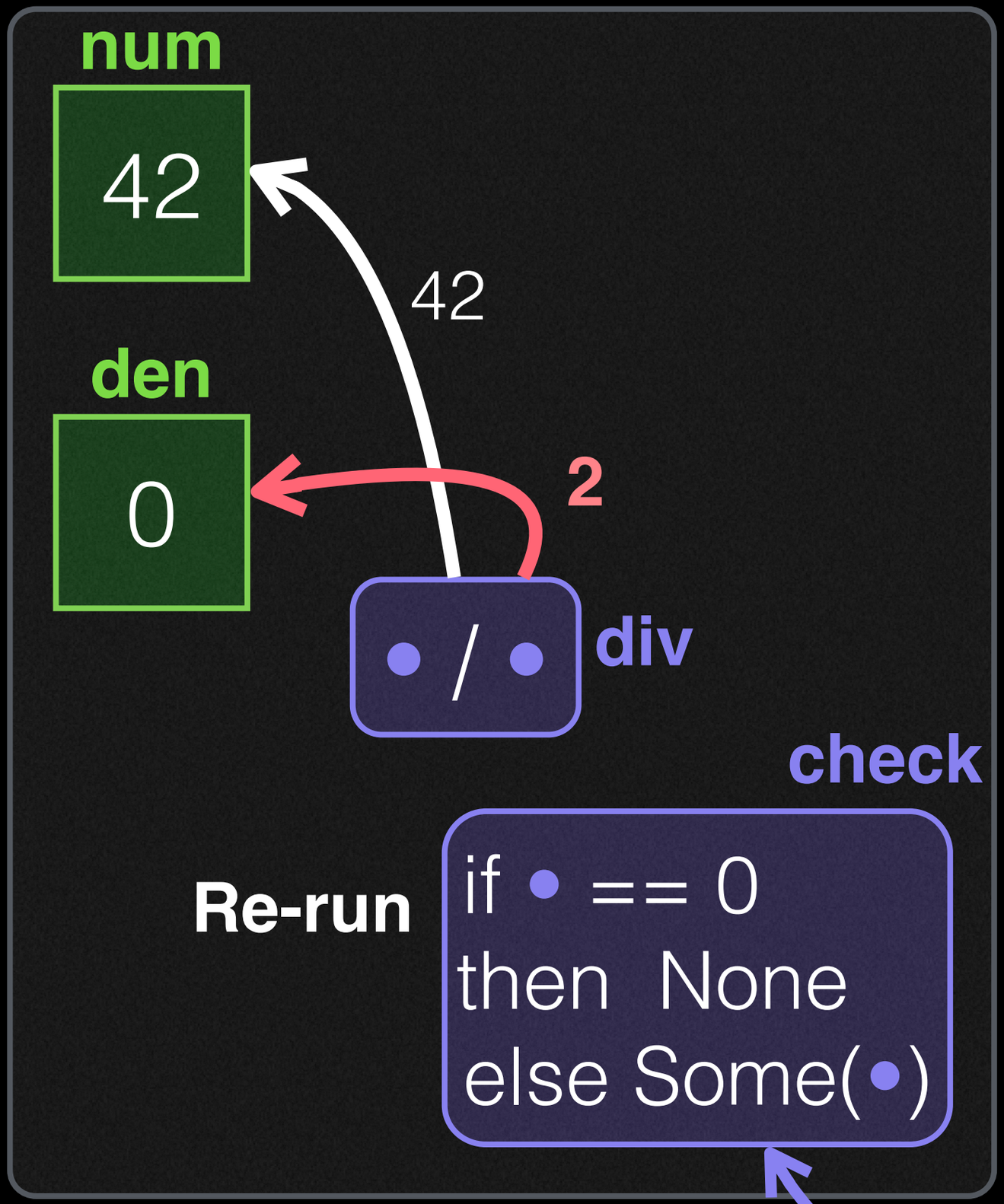

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```




```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

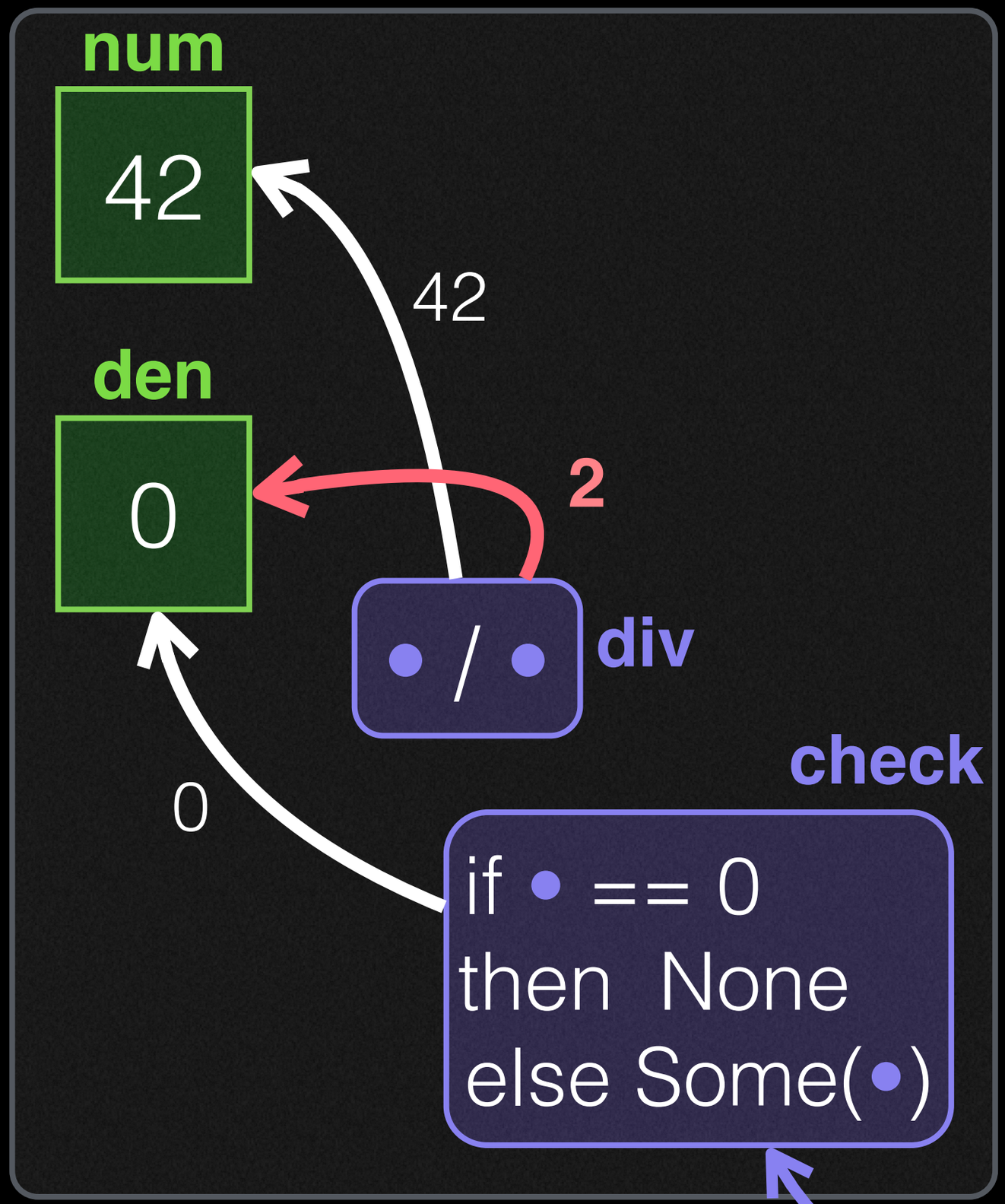
root

Change propagation (transitive cleaning)



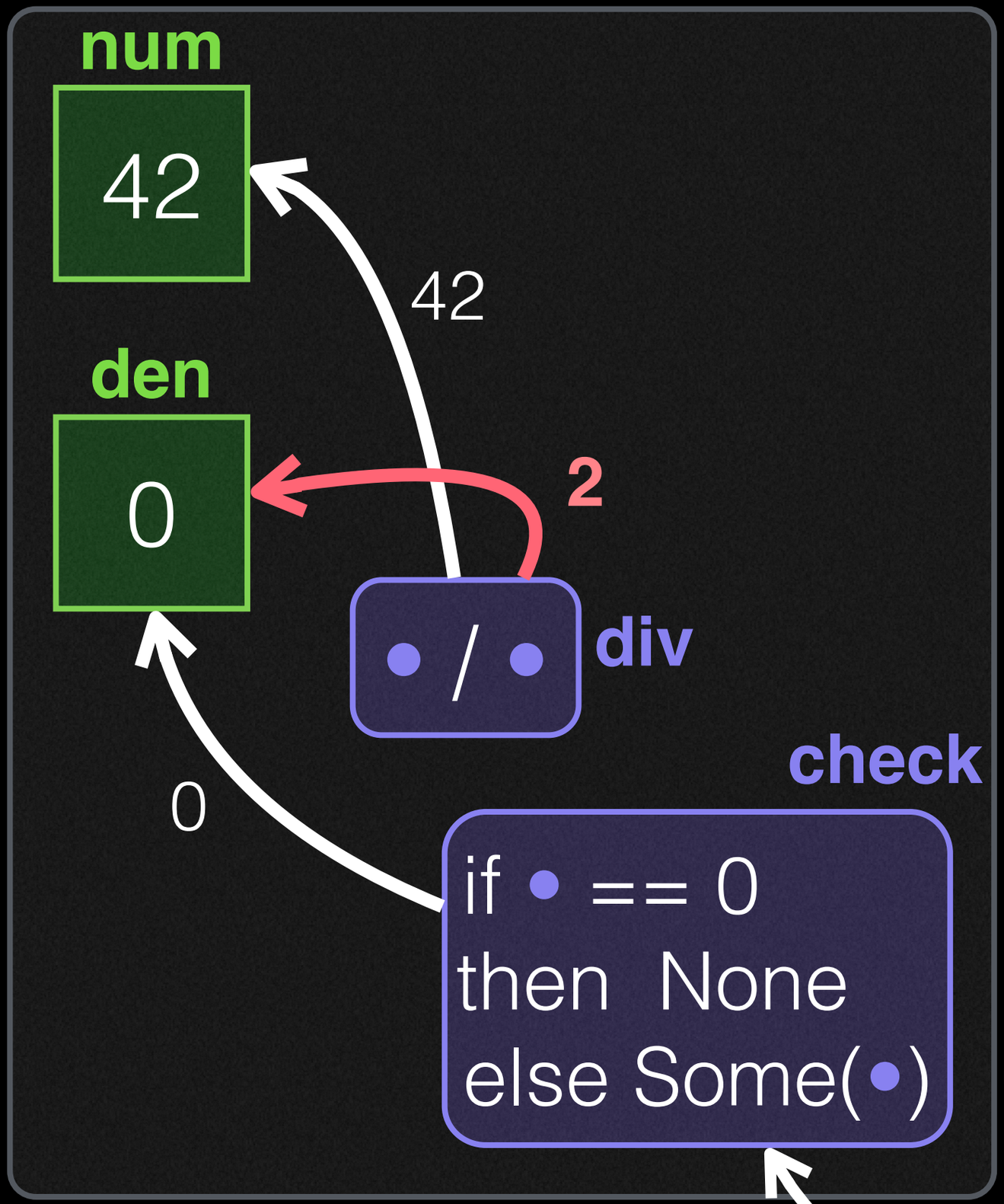

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

Change propagation (transitive cleaning)



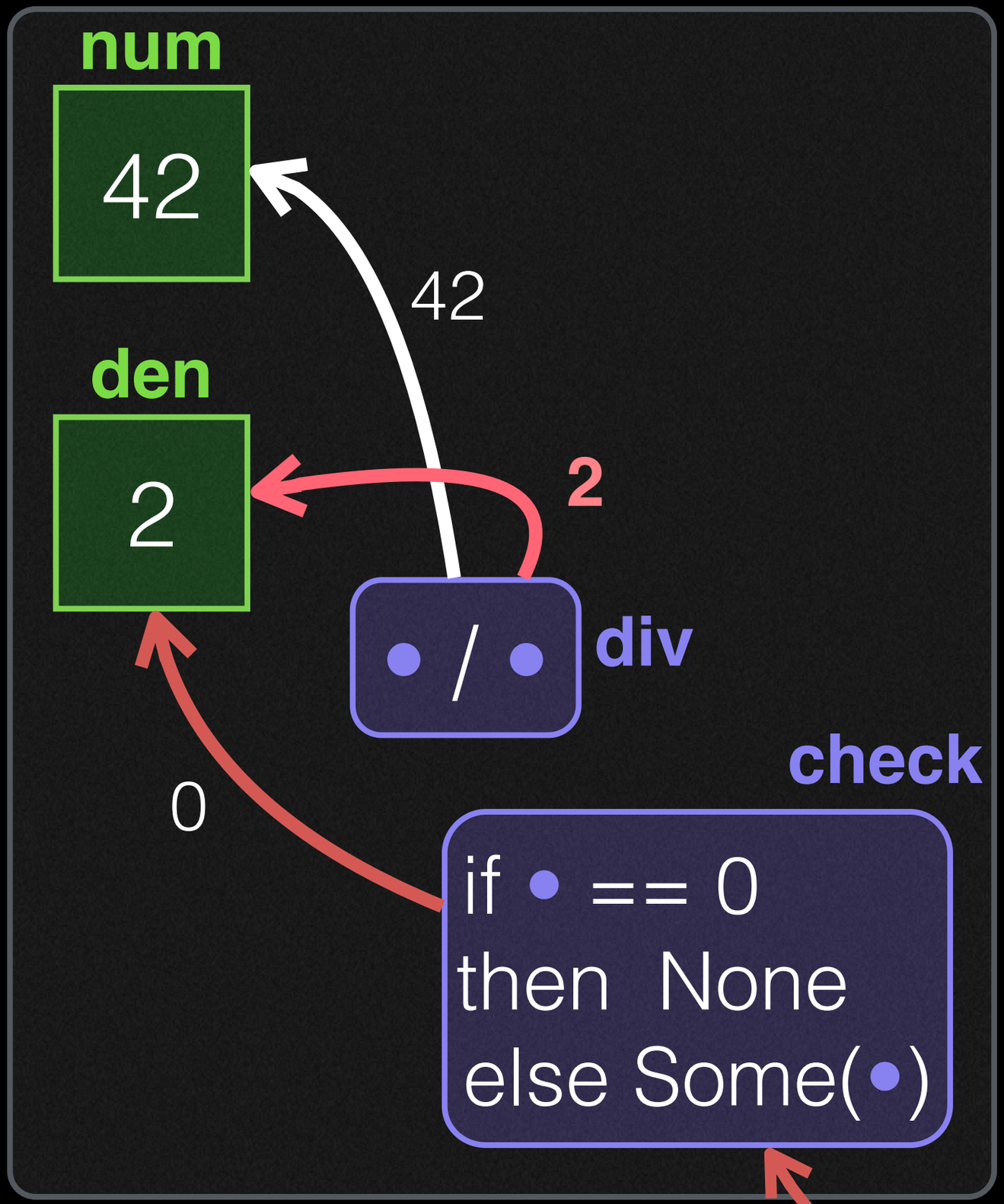

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

Change propagation (transitive cleaning)

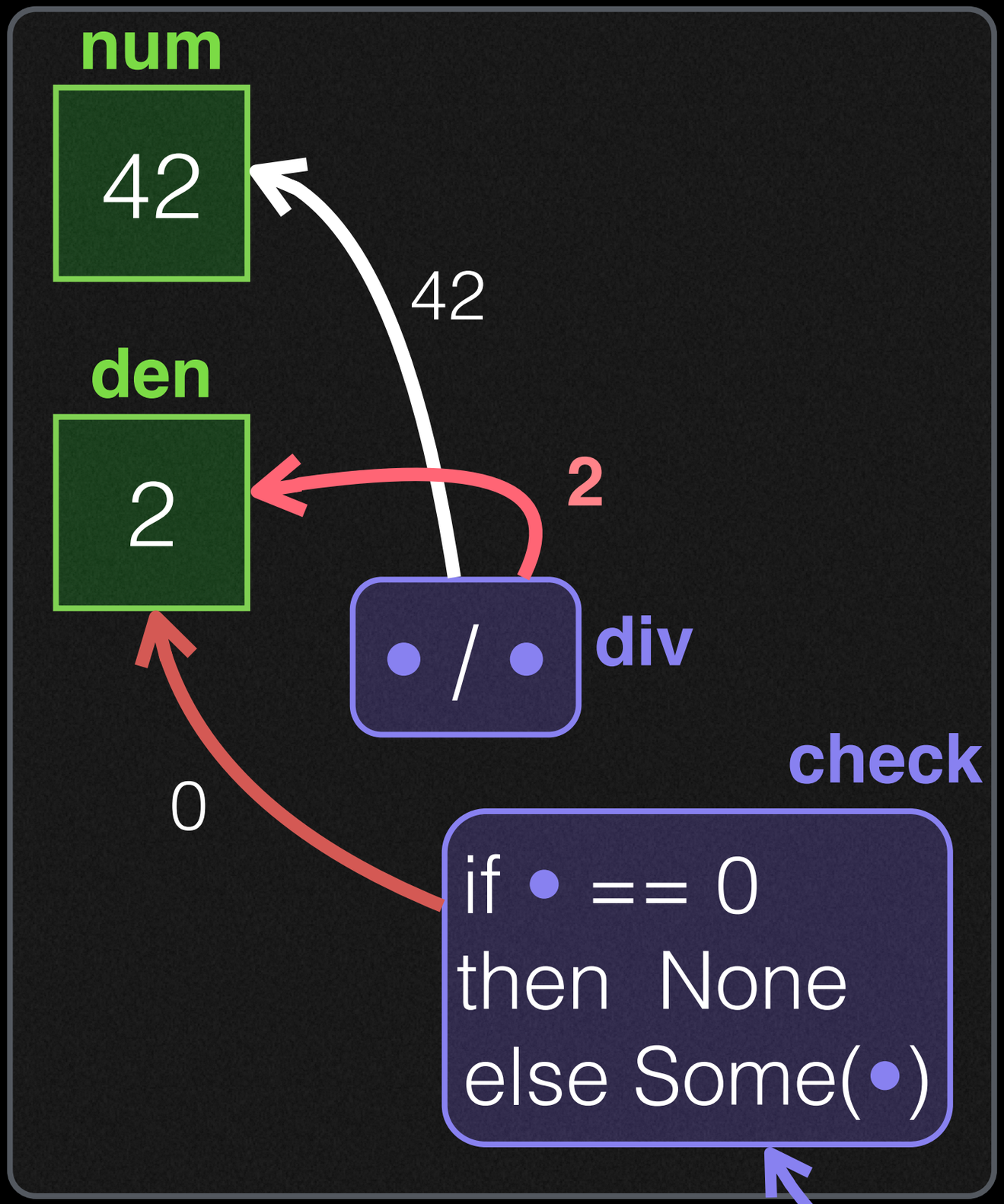



```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

Change propagation (transitive cleaning)

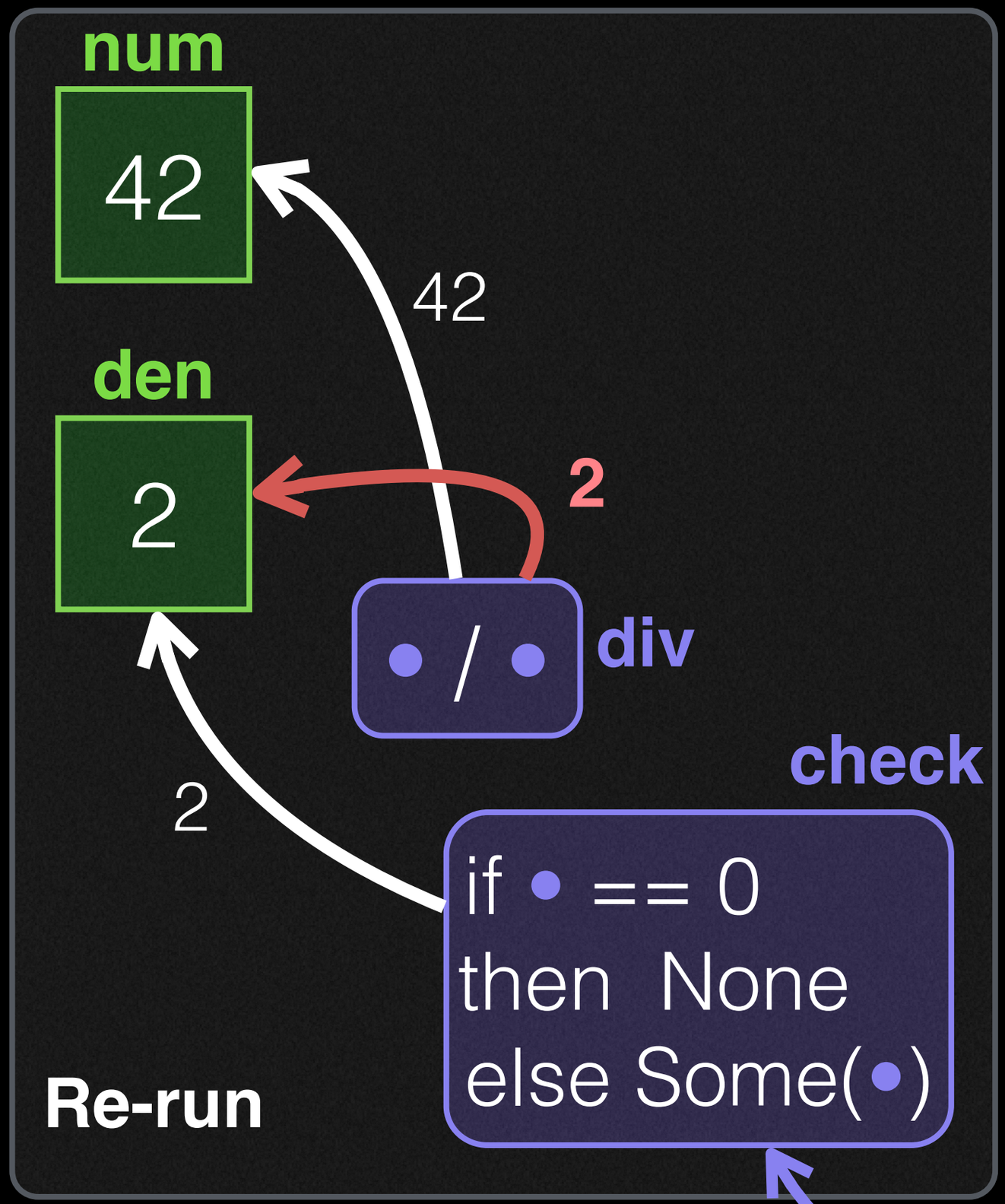



```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

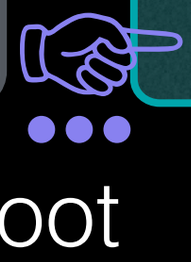



```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

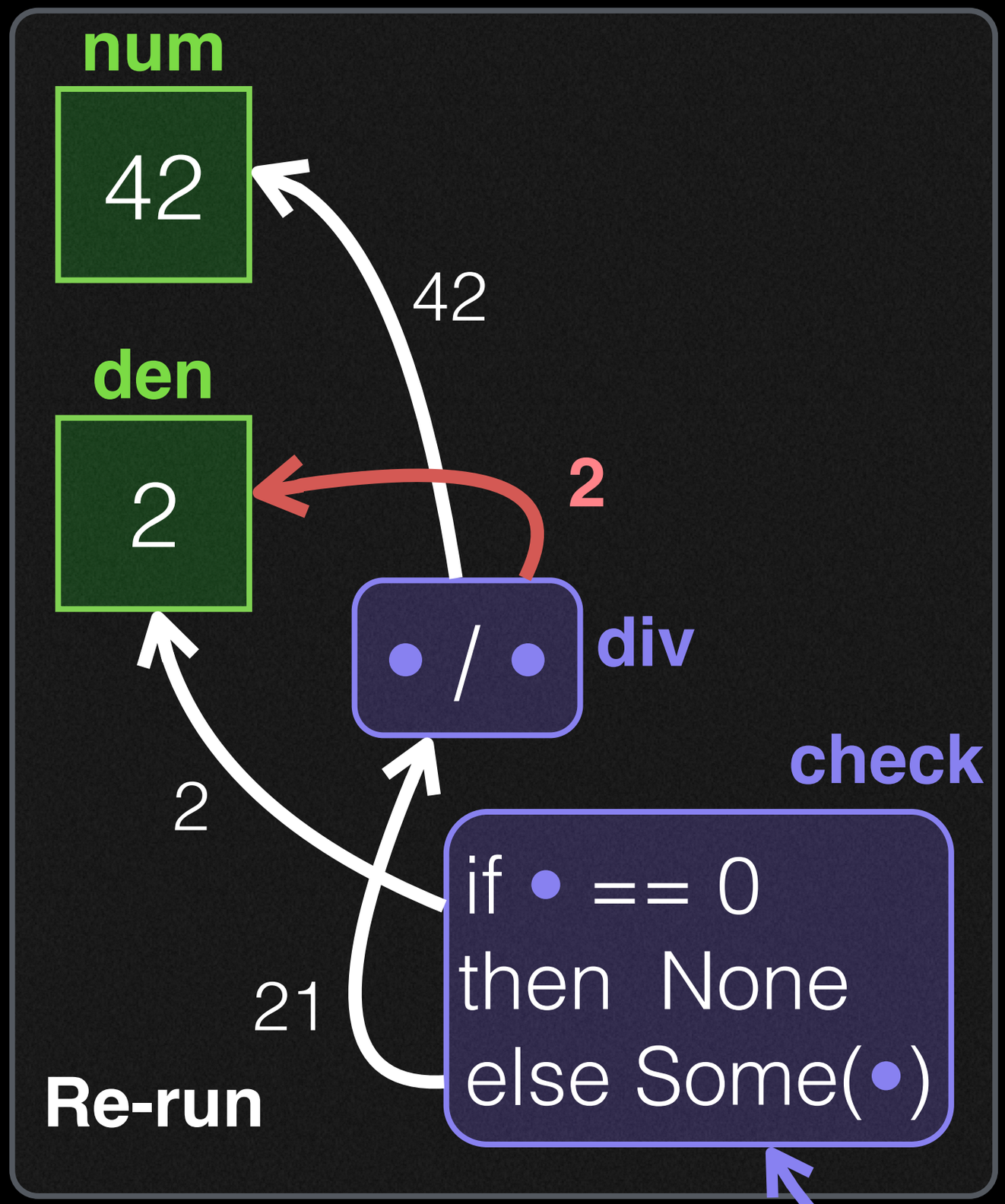
Change propagation (transitive cleaning)




```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```



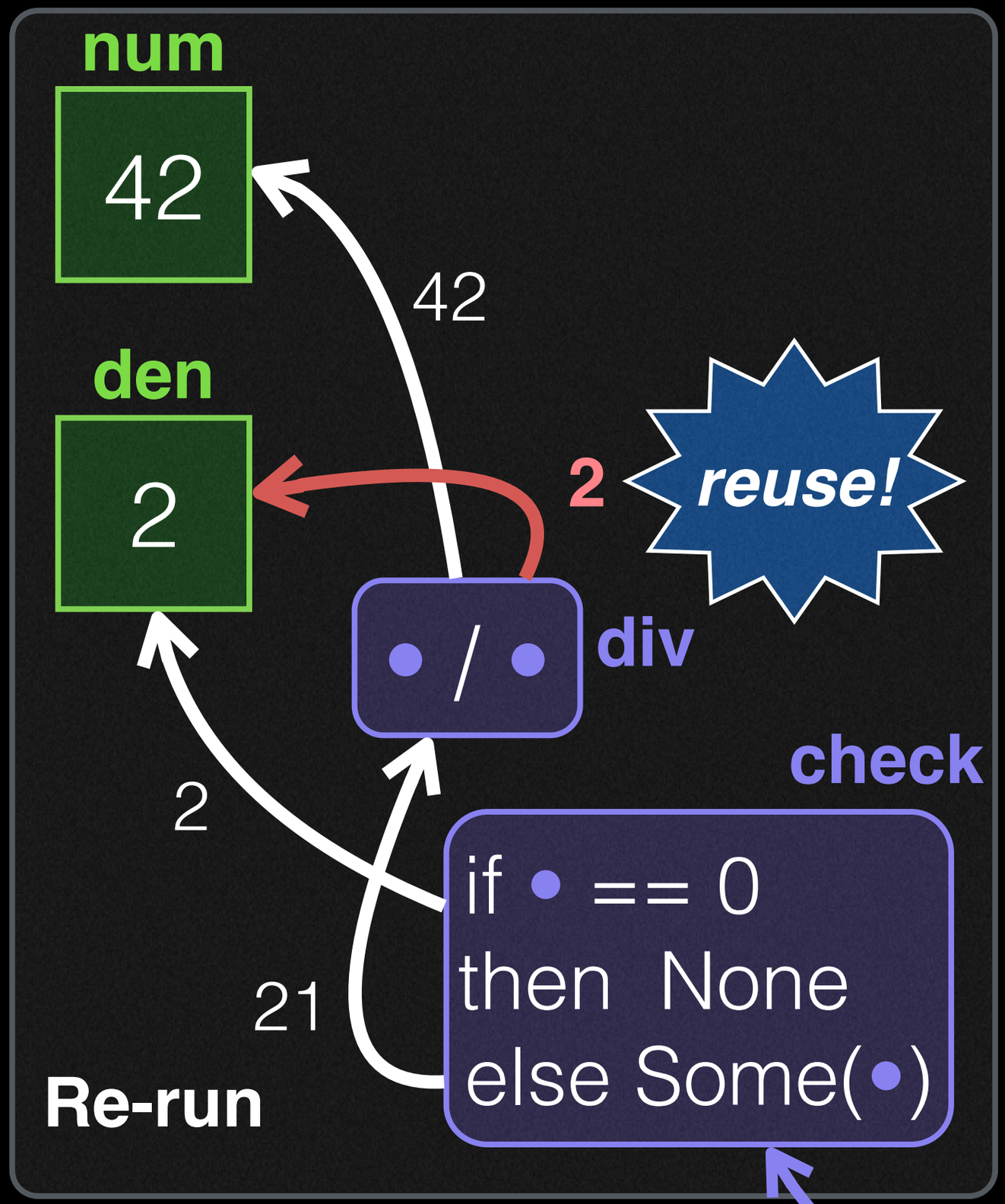
Change propagation (transitive cleaning)




```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

root

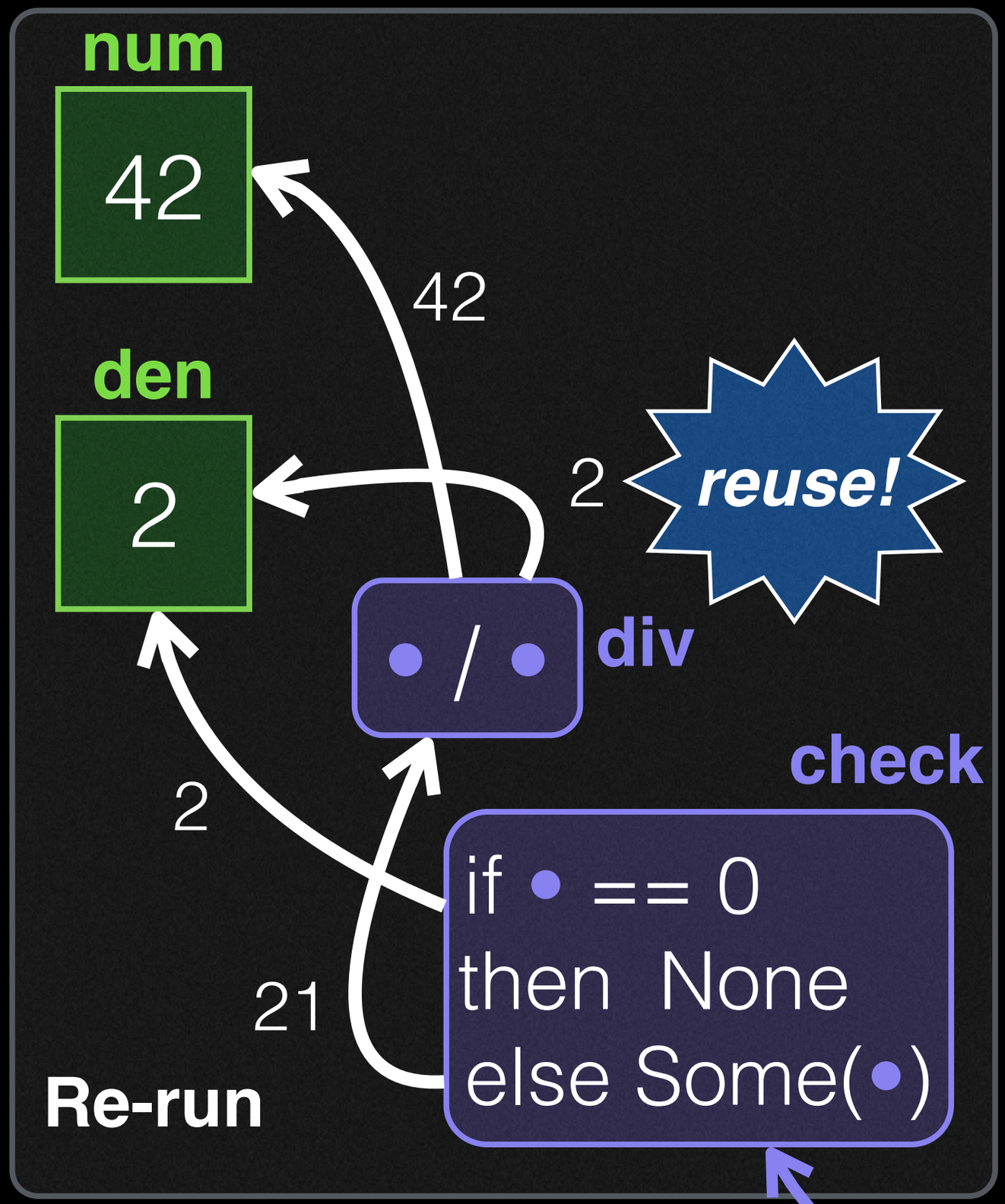
Change propagation (transitive cleaning)




```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

root

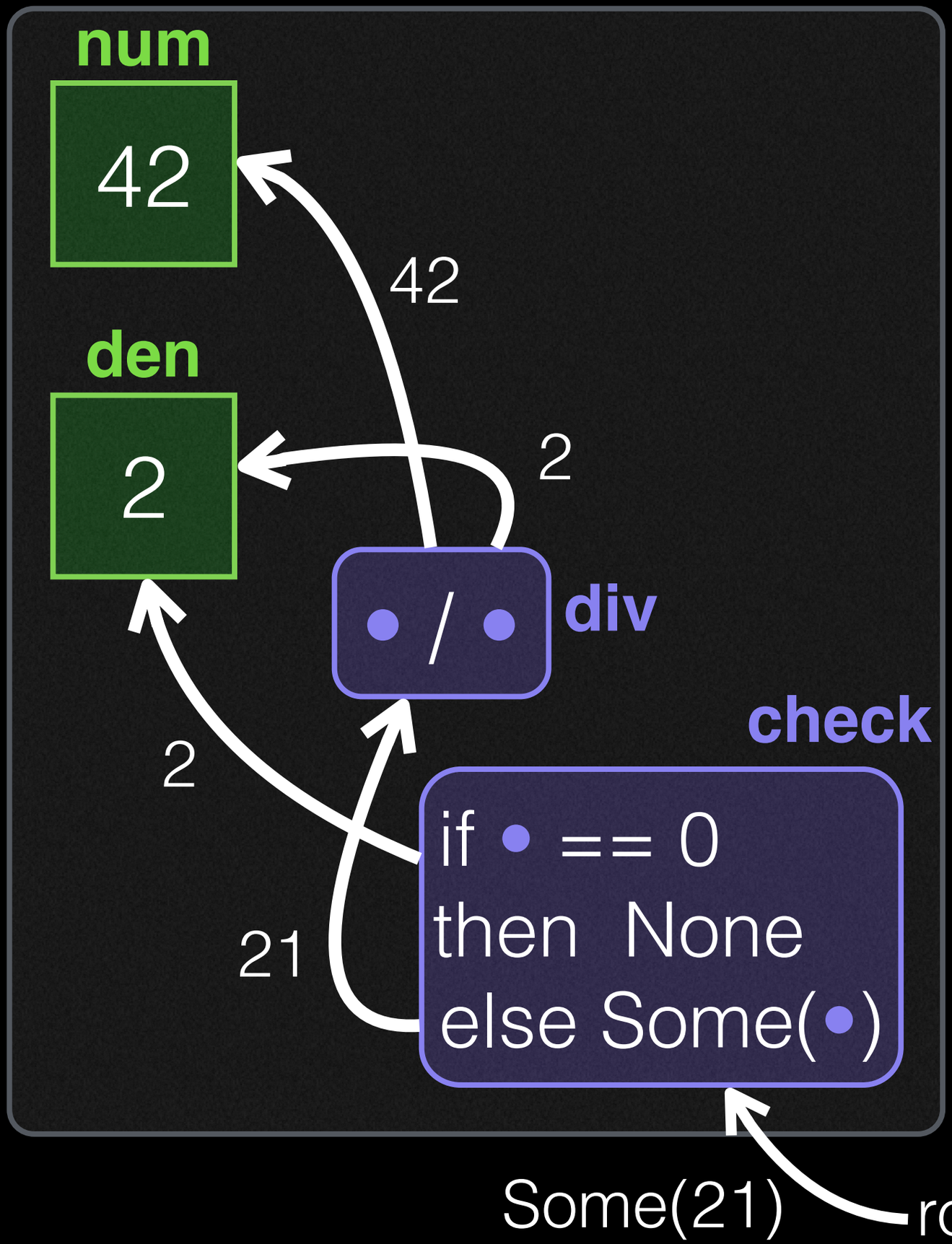
Change propagation (transitive cleaning)



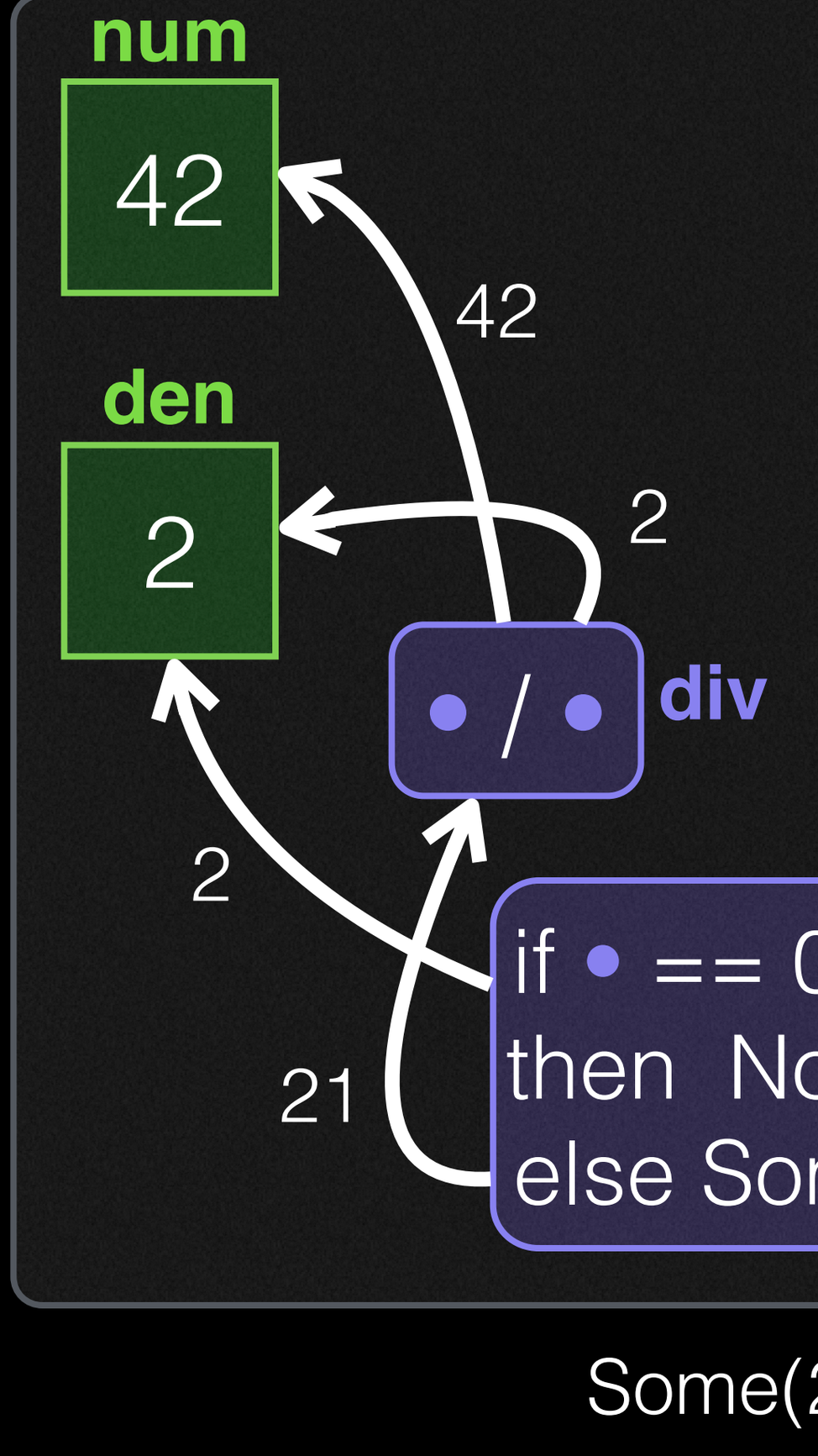

```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```

root

Change propagation (transitive cleaning)




```
let num = cell(42)
let den = cell(2)
let div = thunk [
  get(num) / get(den)
 ]
let check = thunk [
  if get(den) == 0
  then None
  else Some(get(div))
 ]

get(check)
set(den, 0); get(check)
set(den, 2); get(check)
```



Summary:

- dirtying is eager

- **cleaning** is **demand-driven**
  c.f. self-adjusting computation

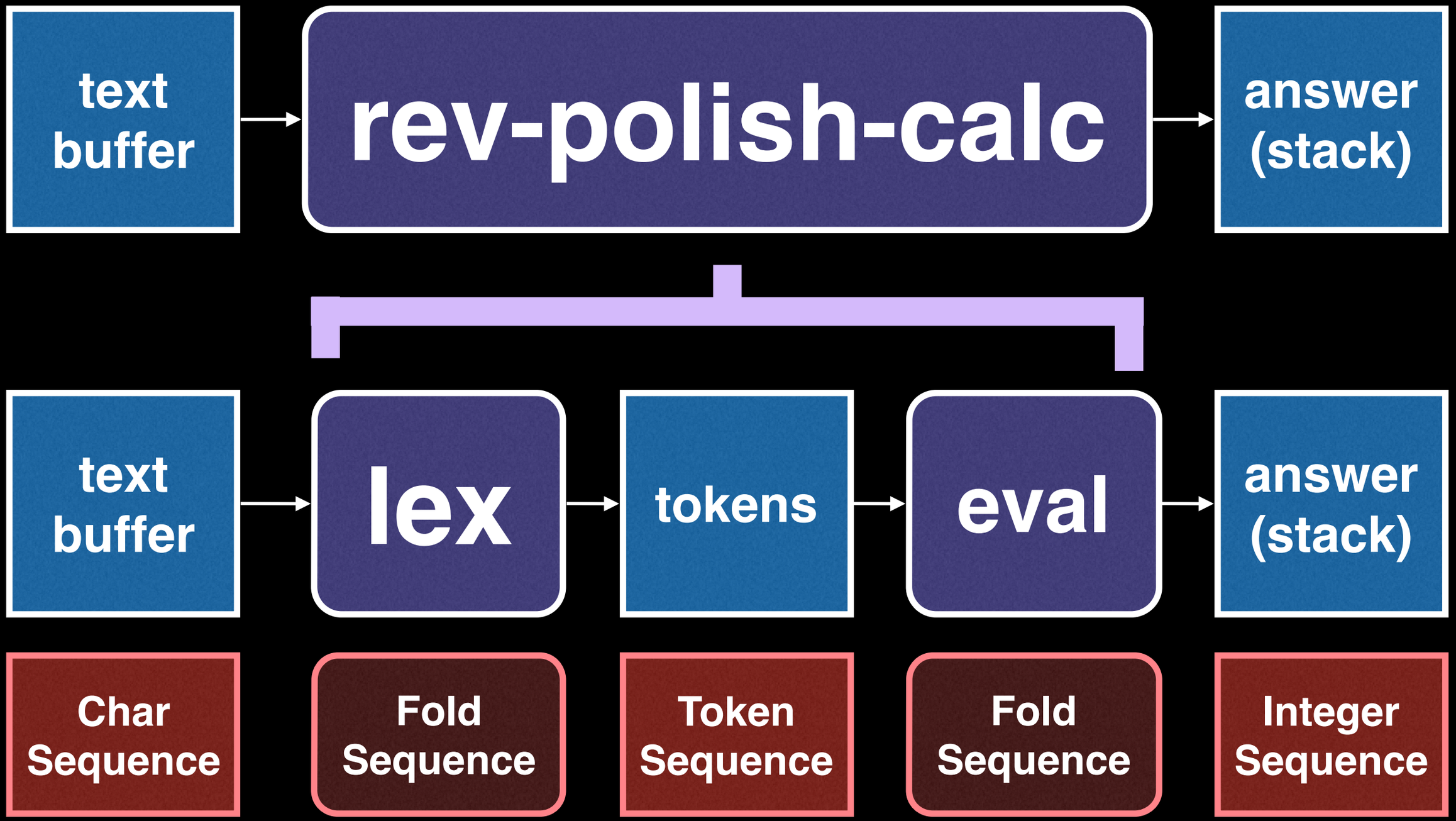
- **cleaning** is **consistent / sound**
  ("from-scratch consistent", FSC)
  c.f. "height-based" change prop

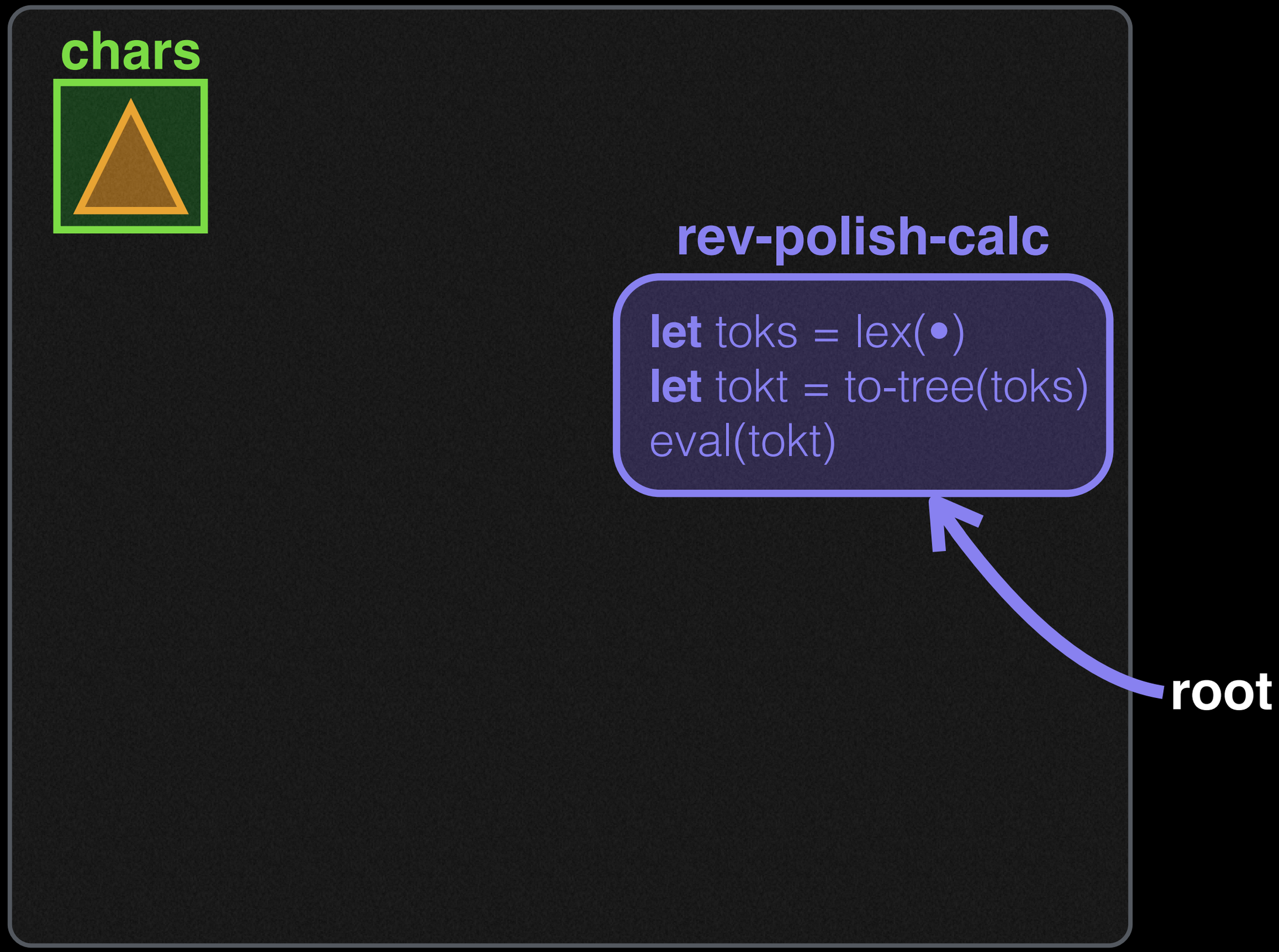
get(check)
set(den, 0); **get**(check)
set(den, 2); **get**(check)

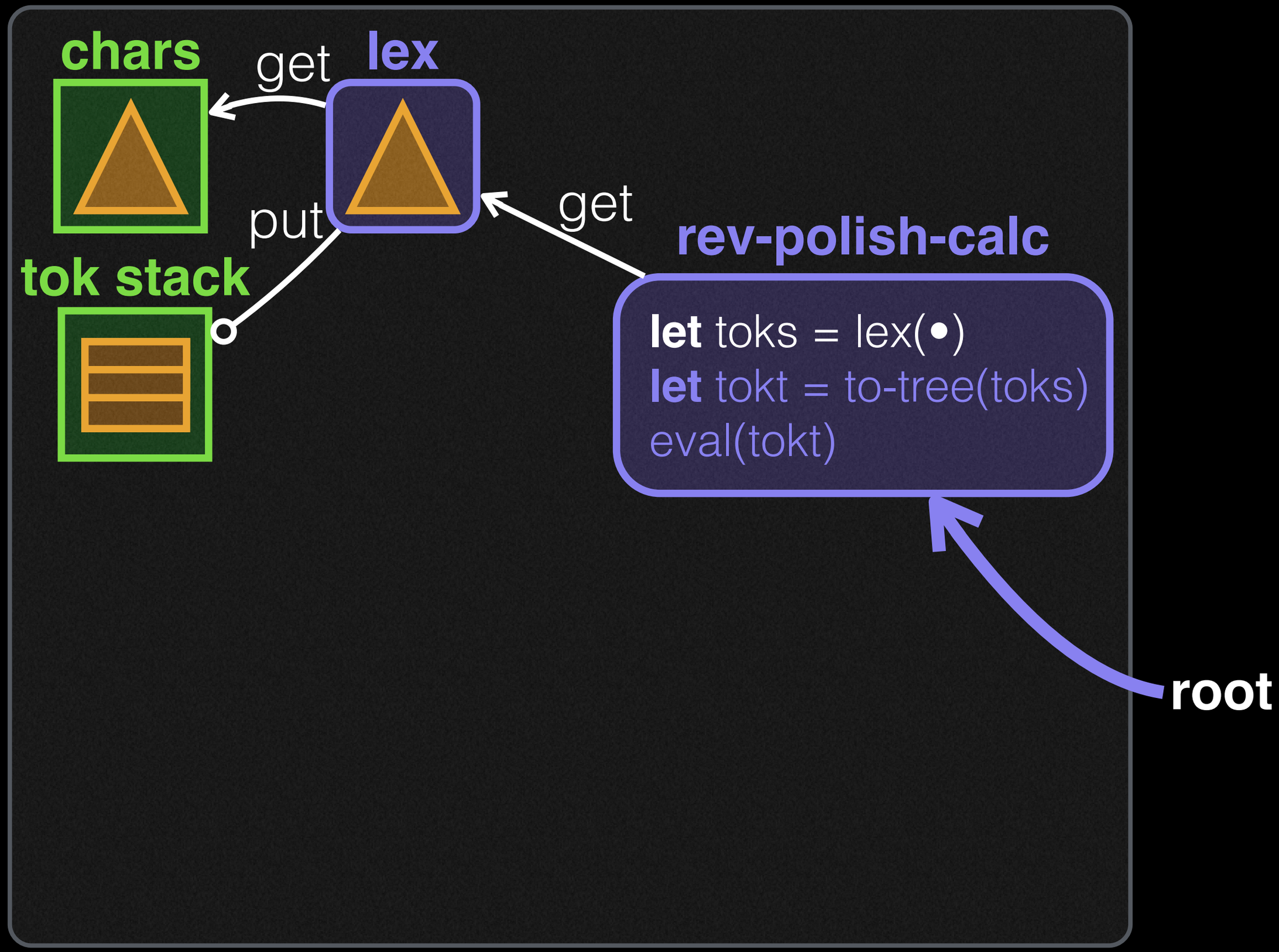
Large, Dynamic DCGs

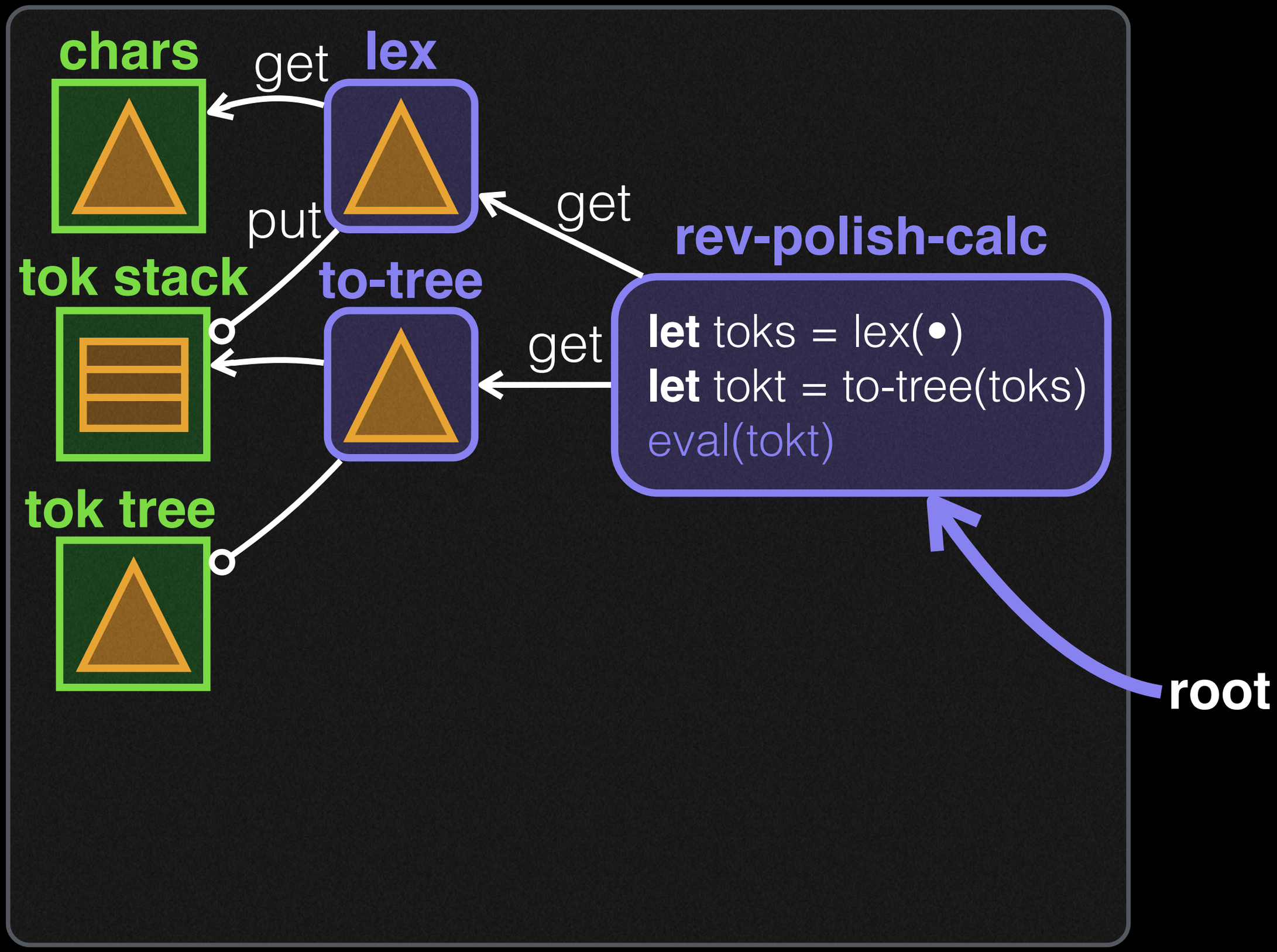
Adapton + Dynamic Collections

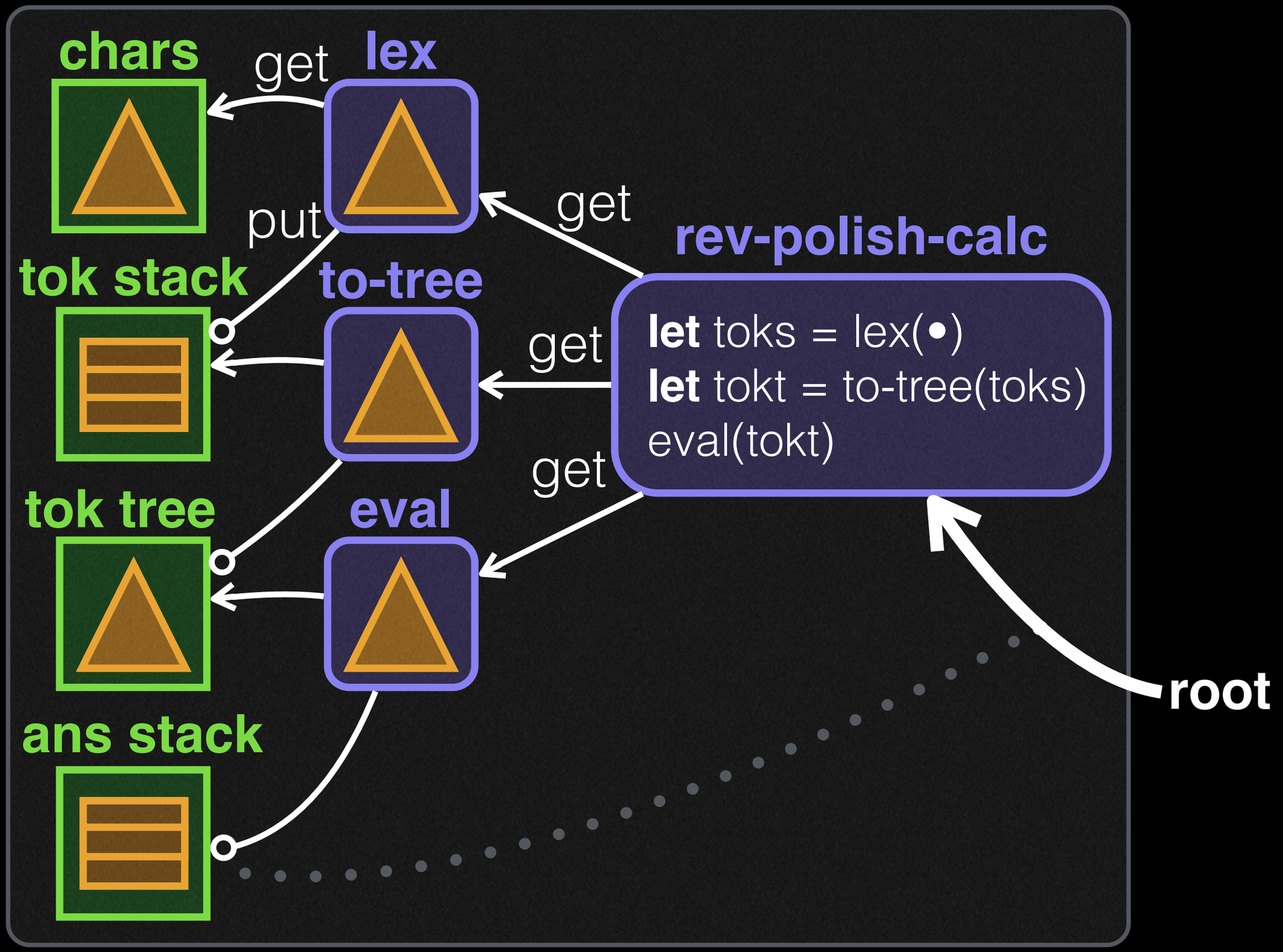
Large, Dynamic DCGs

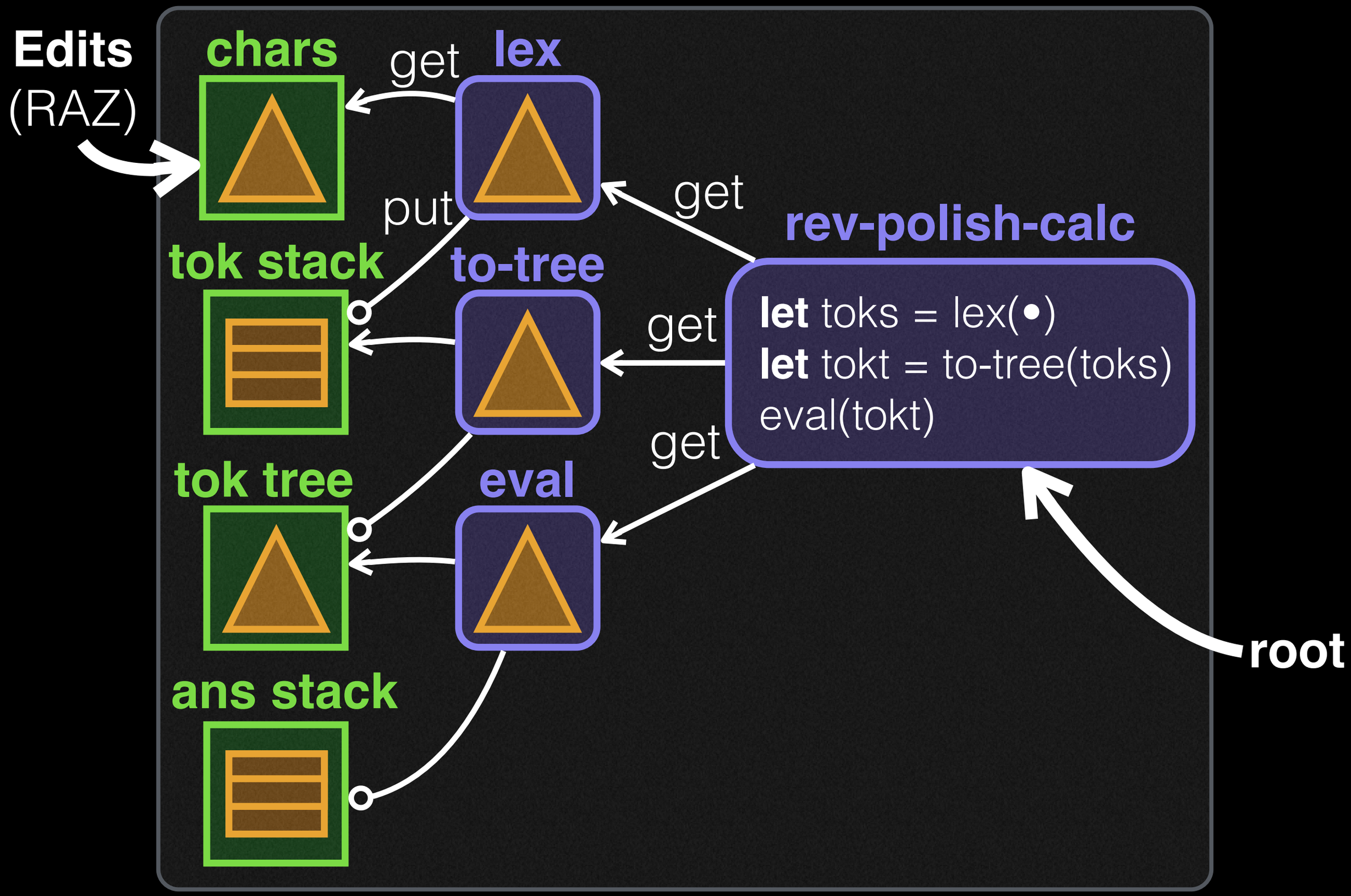


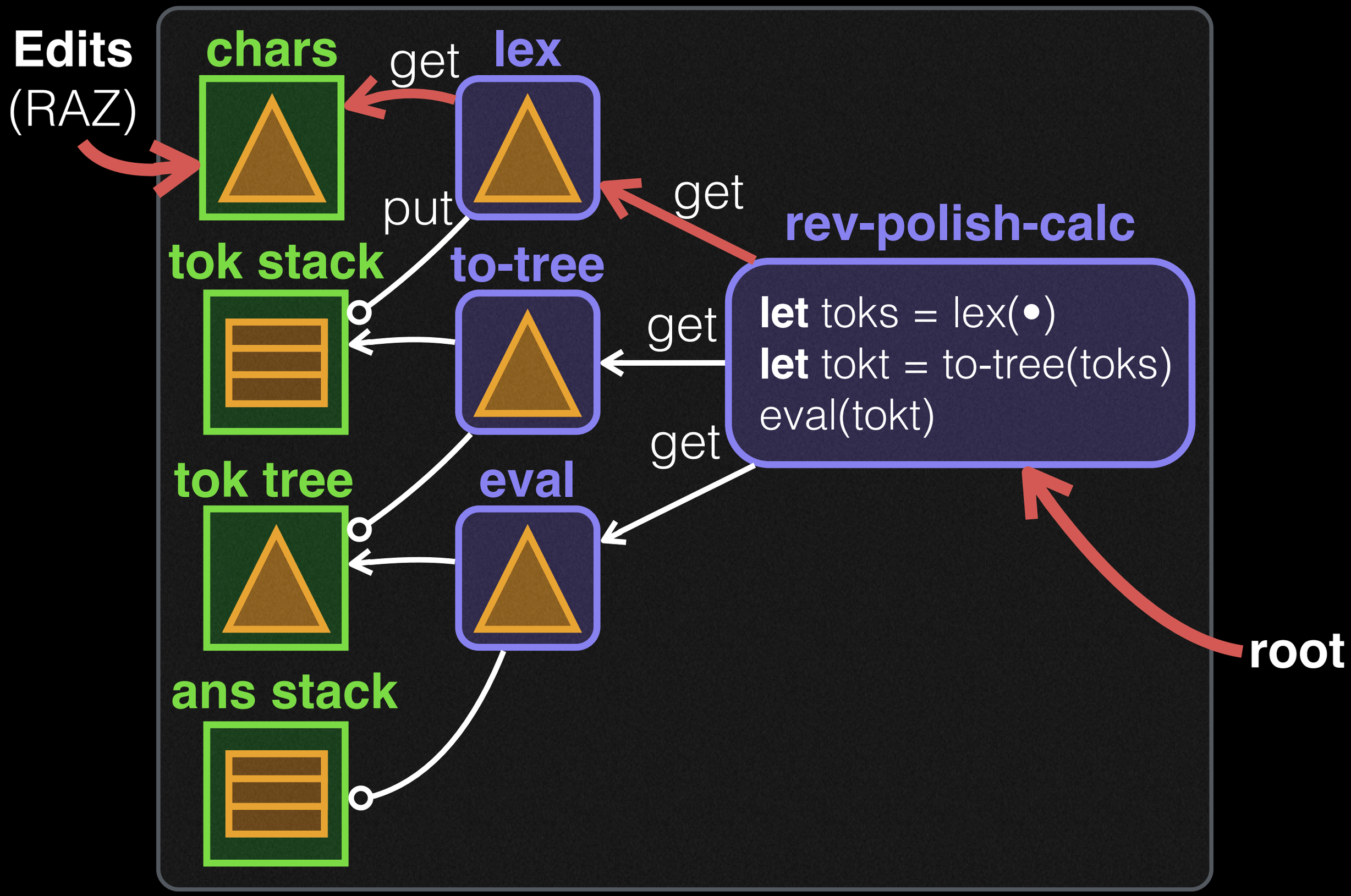

chars
rev-polish-calc
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
root


chars
get
lex
put
tok stack
get
rev-polish-calc
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
root

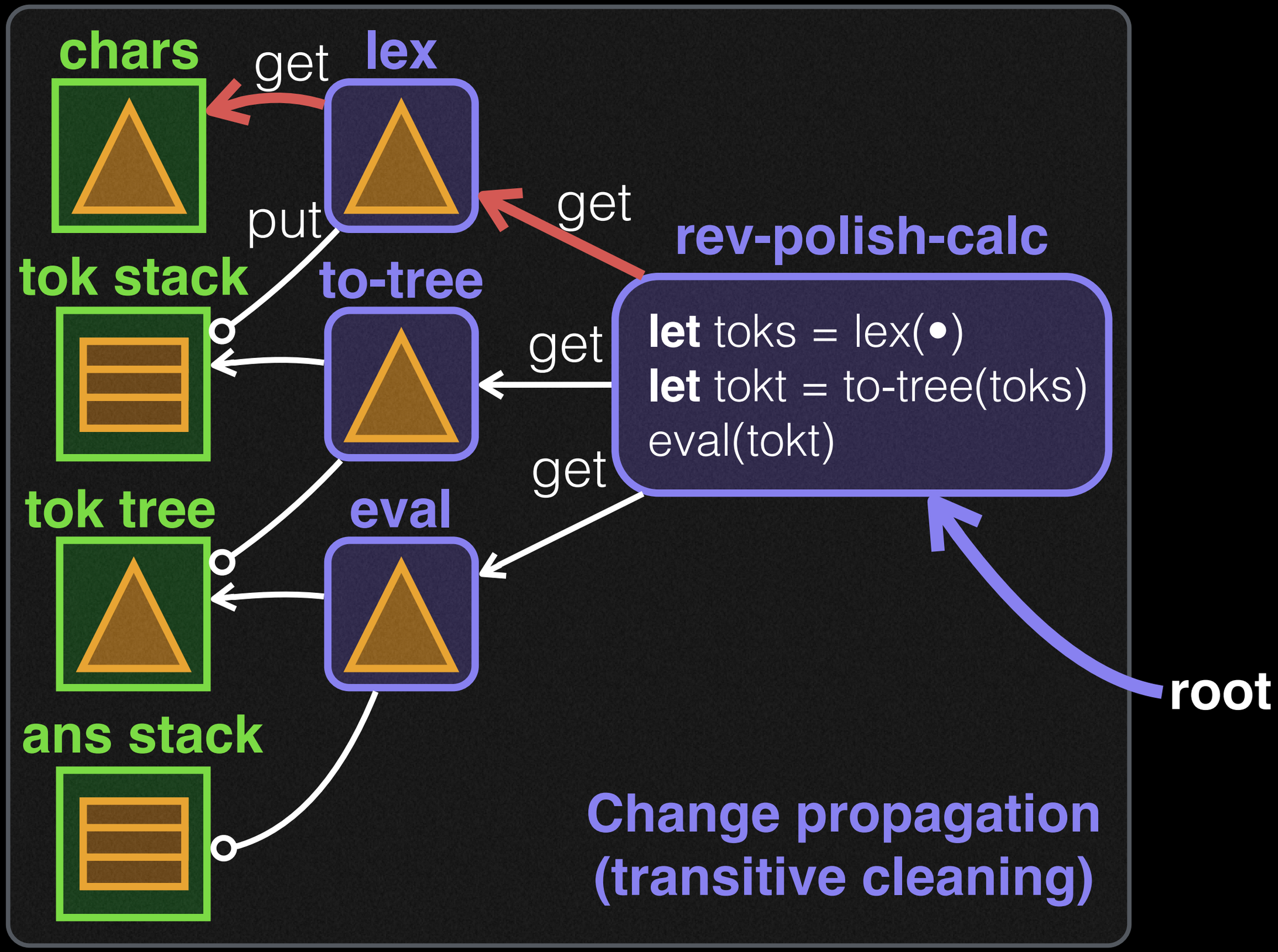

chars
get
lex
put
get
rev-polish-calc
tok stack
to-tree
get
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
tok tree
root


chars
get
lex
put
get
rev-polish-calc
tok stack
to-tree
get
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
get
tok tree
eval
root
ans stack

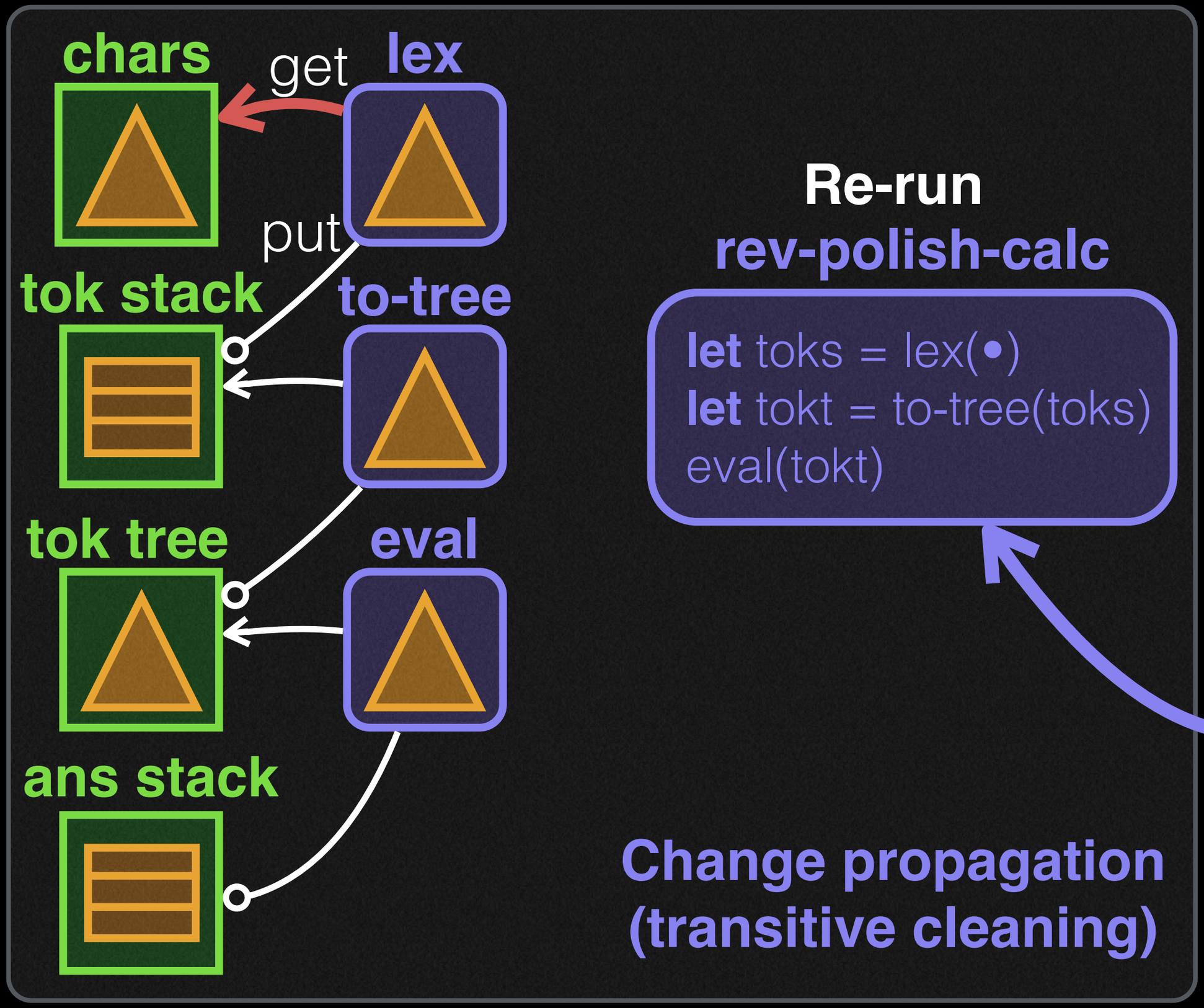

Edits
(RAZ)
chars
get
lex
put
get
rev-polish-calc
tok stack
to-tree
get
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
get
tok tree
eval
root
ans stack


Edits
(RAZ)
chars
get
lex
put
get
rev-polish-calc
tok stack
to-tree
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
get
get
tok tree
eval
root
ans stack

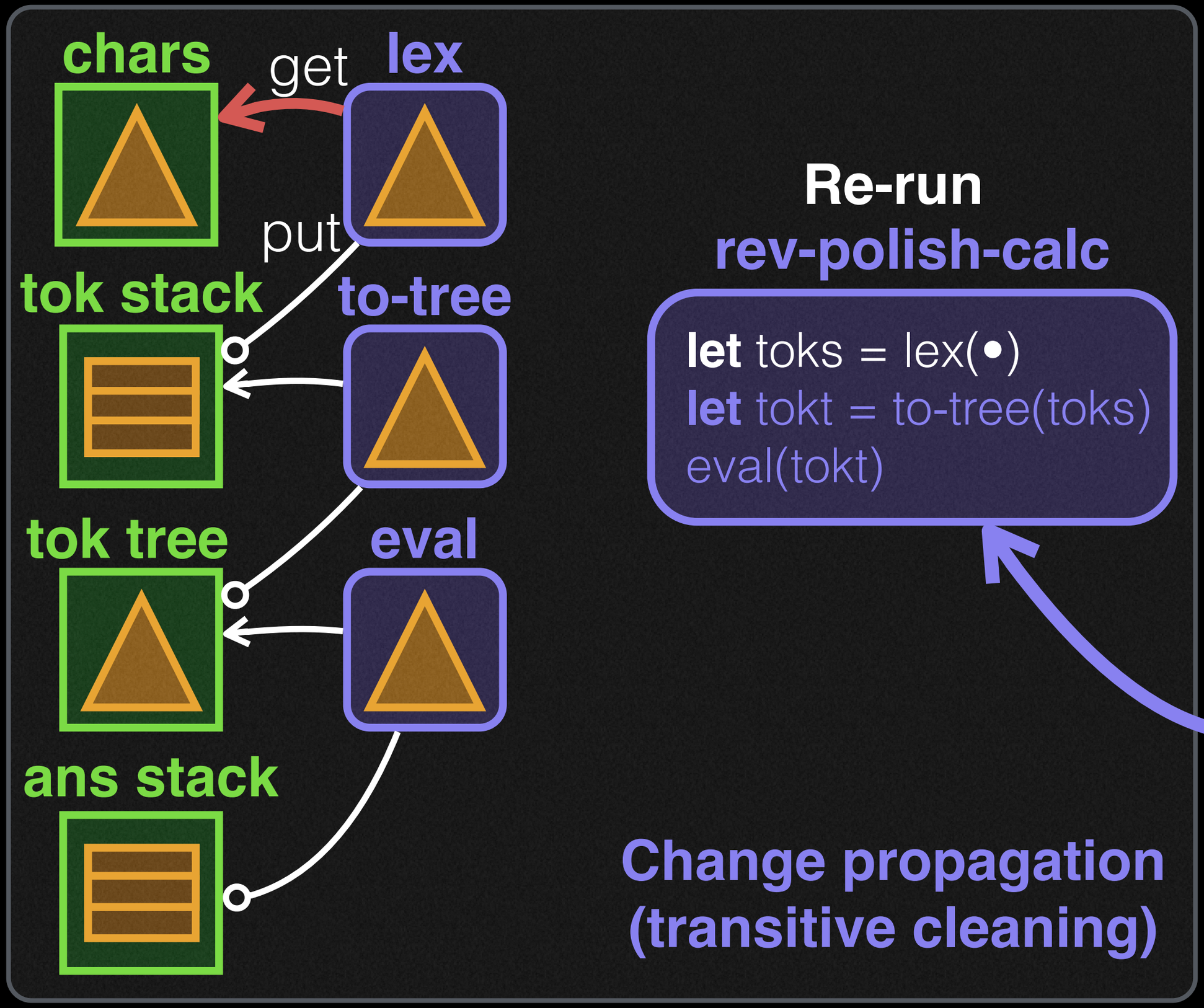
Edits
(RAZ)


chars
get
lex
put
get
rev-polish-calc
tok stack
to-tree
get
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
get
tok tree
eval
root
ans stack
Change propagation
(transitive cleaning)

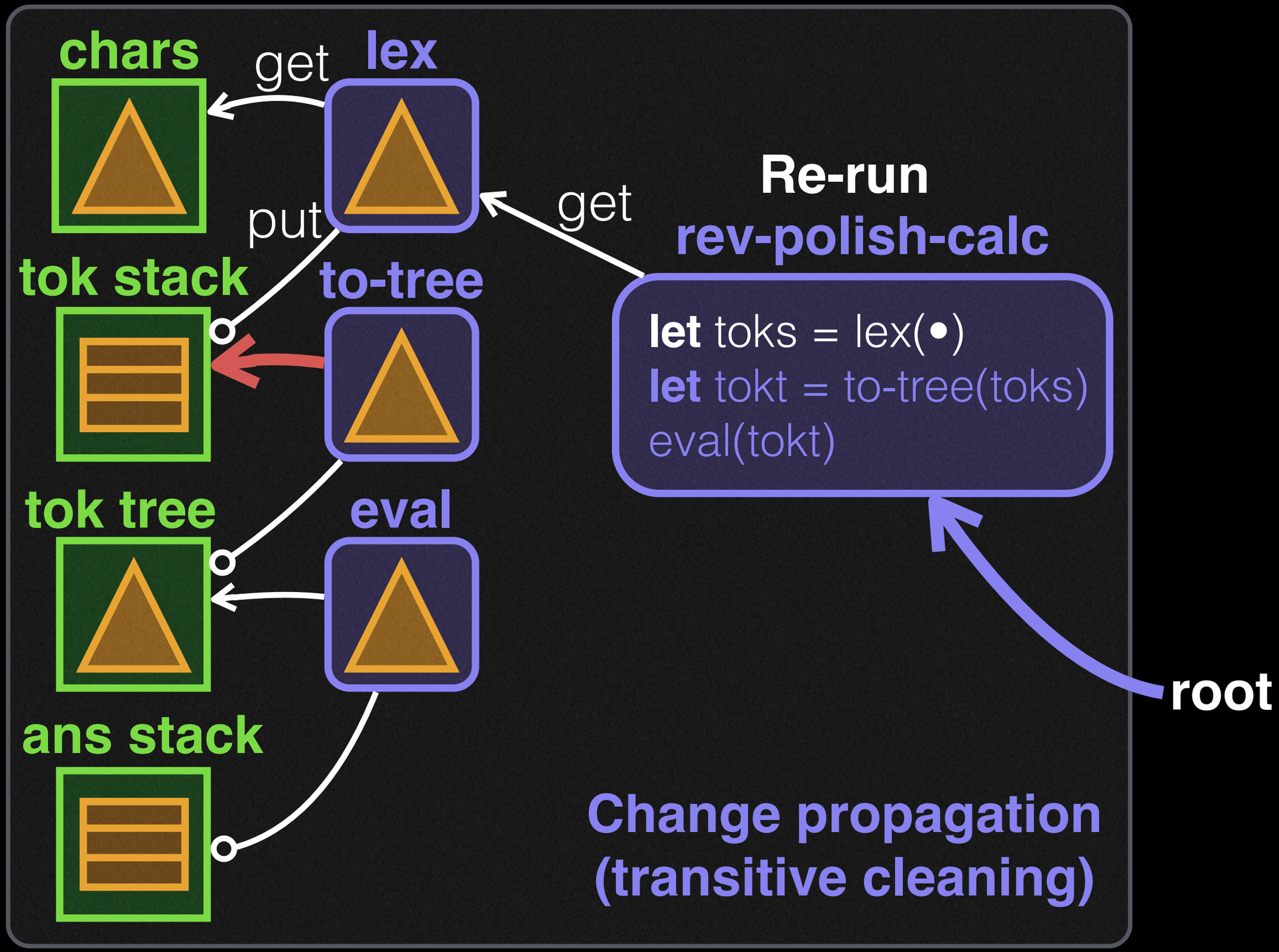
Edits
(RAZ)


chars
get
lex
put
tok stack
to-tree
tok tree
eval
ans stack
Re-run
rev-polish-calc
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
root
Change propagation
(transitive cleaning)

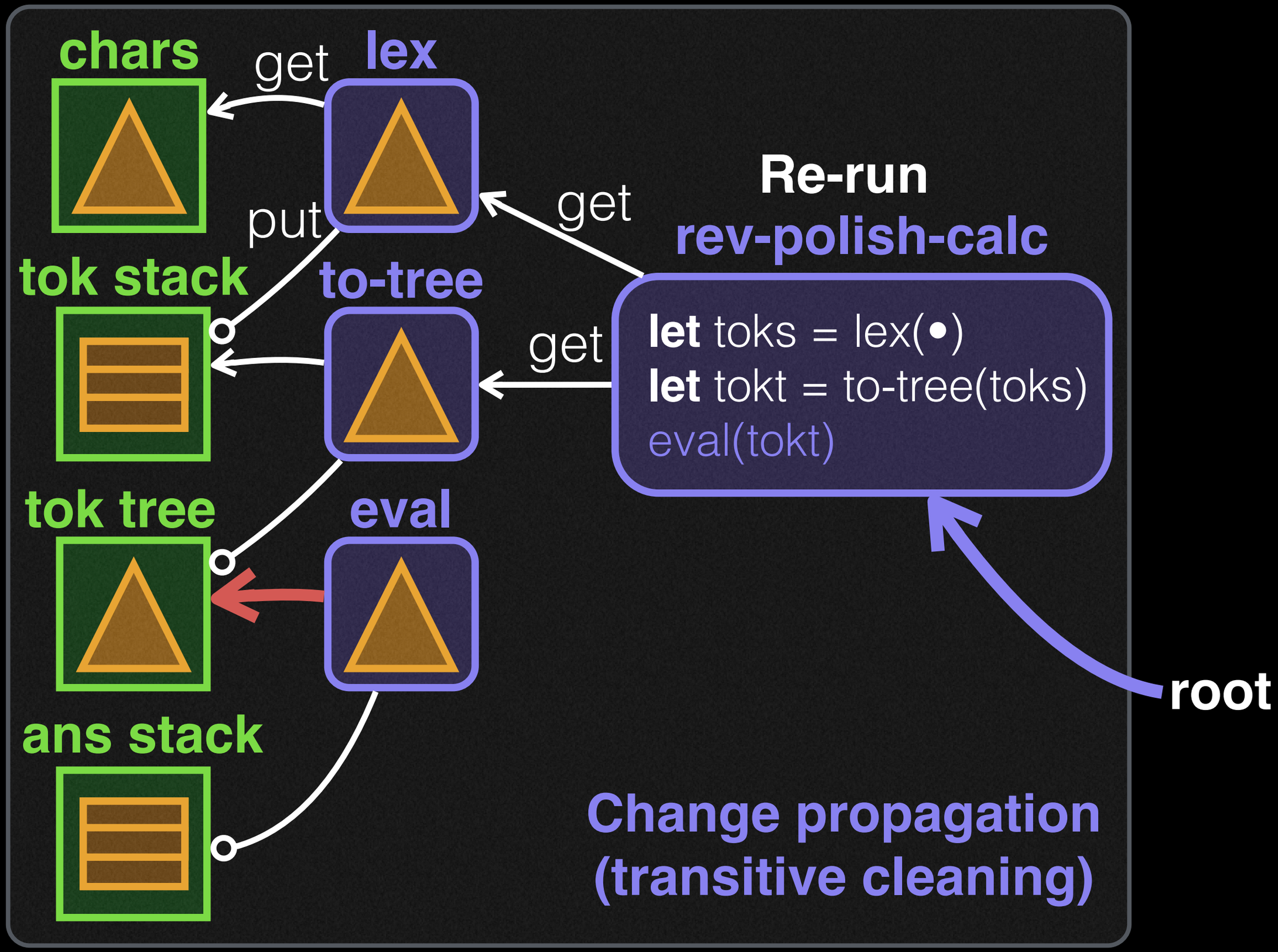
Edits
(RAZ)


chars
get
lex
put
tok stack
to-tree
tok tree
eval
ans stack
Re-run
rev-polish-calc
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
root
Change propagation
(transitive cleaning)

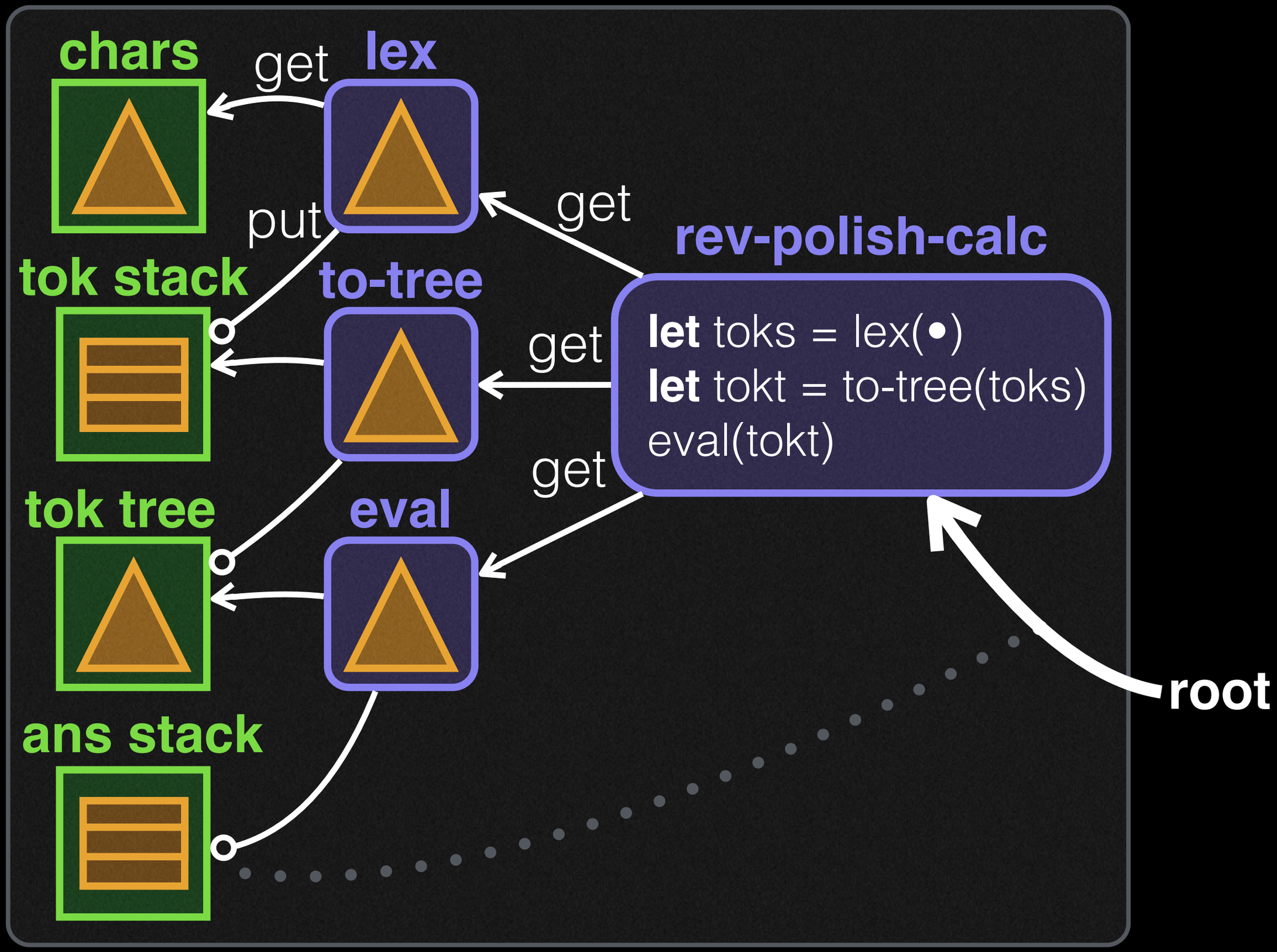
Edits
(RAZ)


chars
get
lex
put
get
Re-run
rev-polish-calc
tok stack
to-tree
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
tok tree
eval
root
ans stack
Change propagation
(transitive cleaning)

Edits
(RAZ)


chars
get
lex
put
get
Re-run
rev-polish-calc
tok stack
to-tree
get
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
tok tree
eval
root
ans stack
Change propagation
(transitive cleaning)

Edits
(RAZ)


chars
get
lex
put
get
rev-polish-calc
tok stack
to-tree
get
let toks = lex(•)
let tokt = to-tree(toks)
eval(tokt)
get
tok tree
eval
root
ans stack

Incremental **Collections**

Avoid deep call graphs:
Avoid tail recursion on lists,
Favor divide-and-conquer on **balanced trees**

Avoid long data dependencies:
Avoid traditional binary search tree (BST) designs
Favor persistent tries, with **nominal insertion**

Incremental **Collections**

Avoid deep call graphs:
Avoid tail recursion on lists,
Favor divide-and-conquer on **balanced trees**

Avoid long data dependencies:
Avoid traditional binary search tree (BST) designs
Favor persistent tries, with **nominal insertion**

Sequences — **Random-Access Zippers**, not Lists
Sets — **Hash Tries**, not AVL- or splay trees
Maps — **Hash Tries**, not AVL- or splay trees

Incremental **Collections**

Avoid deep call graphs:
Avoid tail recursion on lists,
Favor divide-and-conquer on **balanced trees**

Avoid long data dependencies:
Avoid traditional binary search tree (BST) designs
Favor persistent tries, with **nominal insertion**

Sequences — **Random-Access Zippers**, not Lists
Sets — **Hash Tries**, not AVL- or splay trees
Maps — **Hash Tries**, not AVL- or splay trees

RAZ: See TFP 2016 Paper for details!

Incremental **Sequences**

Levels
CTZ(hash(•))

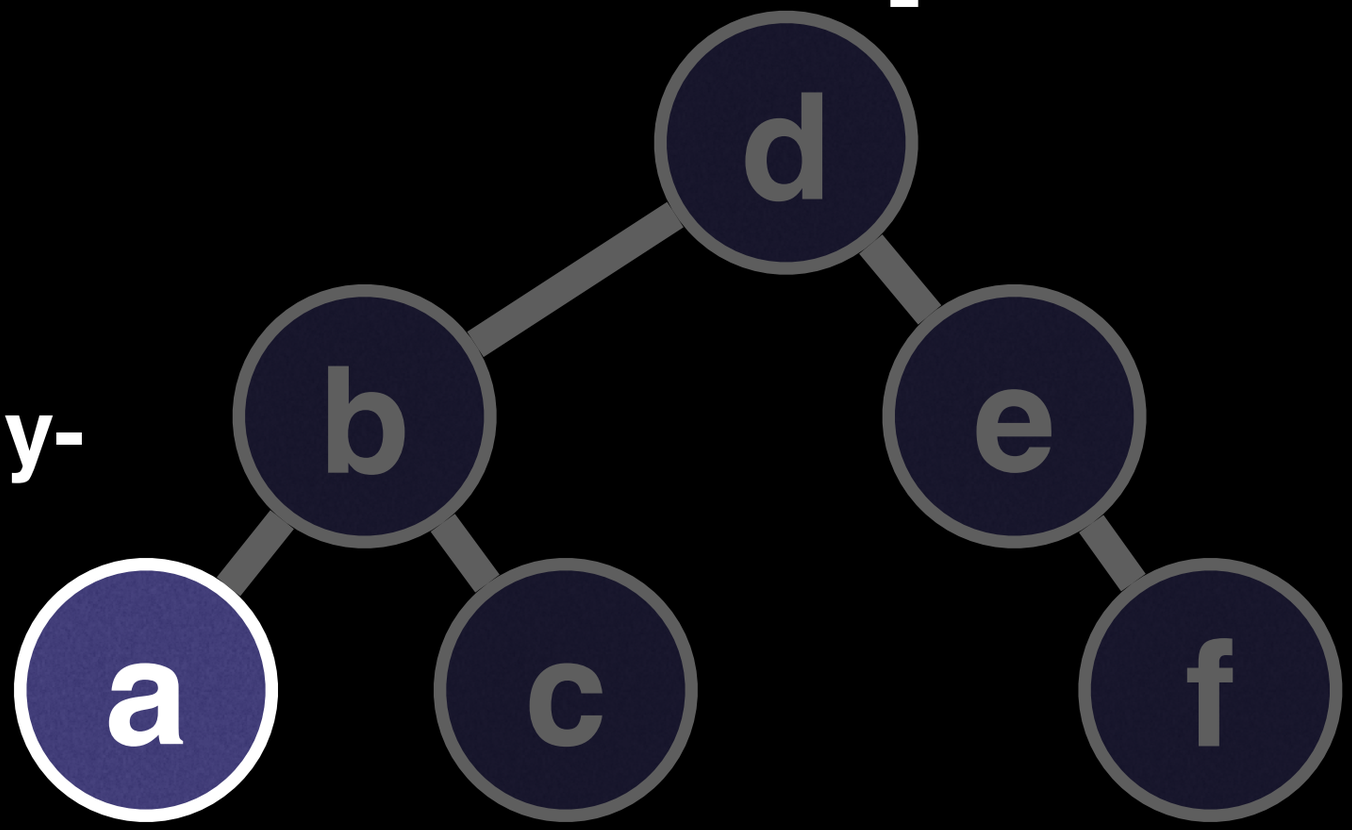
[0, 1, 0, 2, 1, 0]

Linked List

[a, b, c, d, e, f]

Incremental **Sequences**

Probabilistically-Balanced Binary Tree



Levels
CTZ(hash(•))

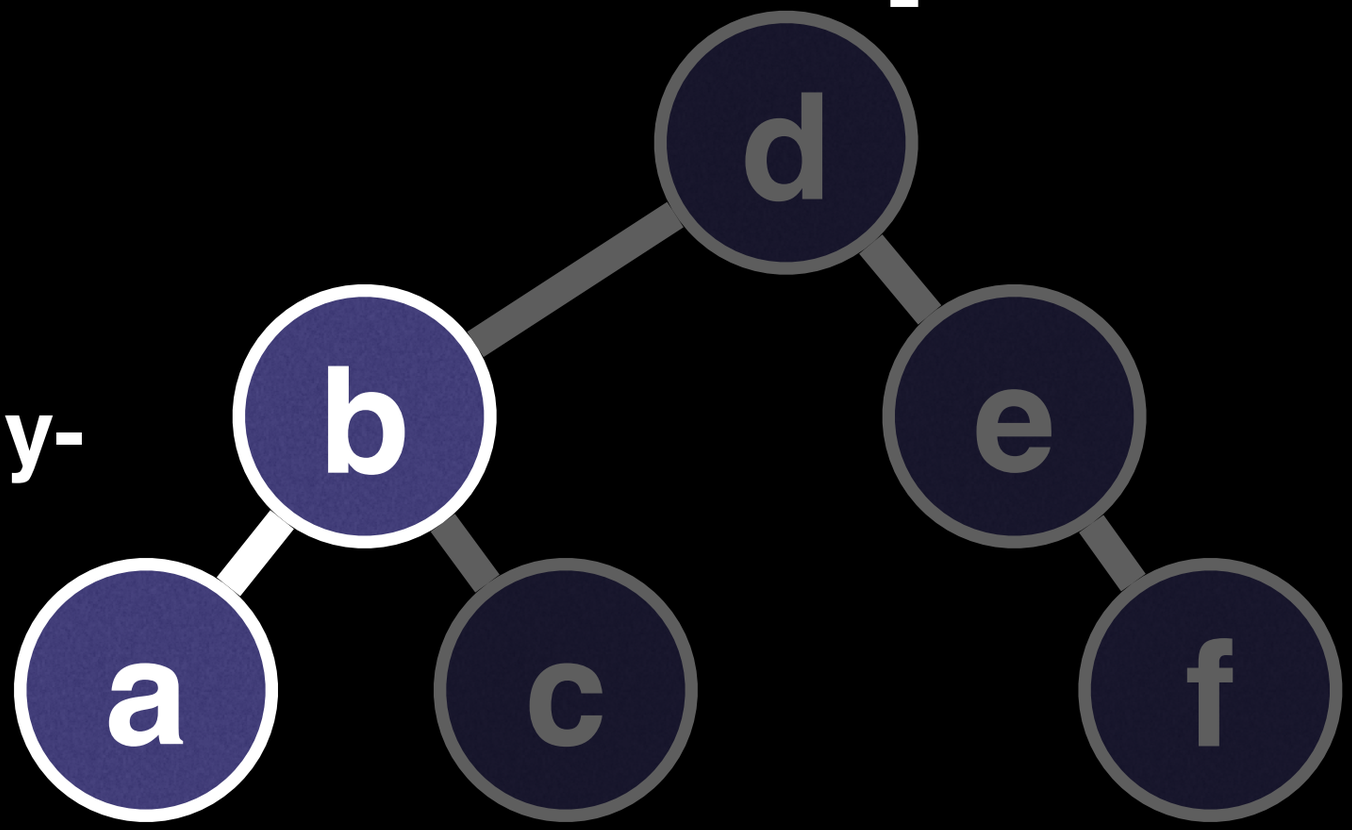
[0, 1, 0, 2, 1, 0]

Linked List

[a, b, c, d, e, f]

Incremental **Sequences**

Probabilistically-Balanced Binary Tree



Levels
CTZ(hash(•))

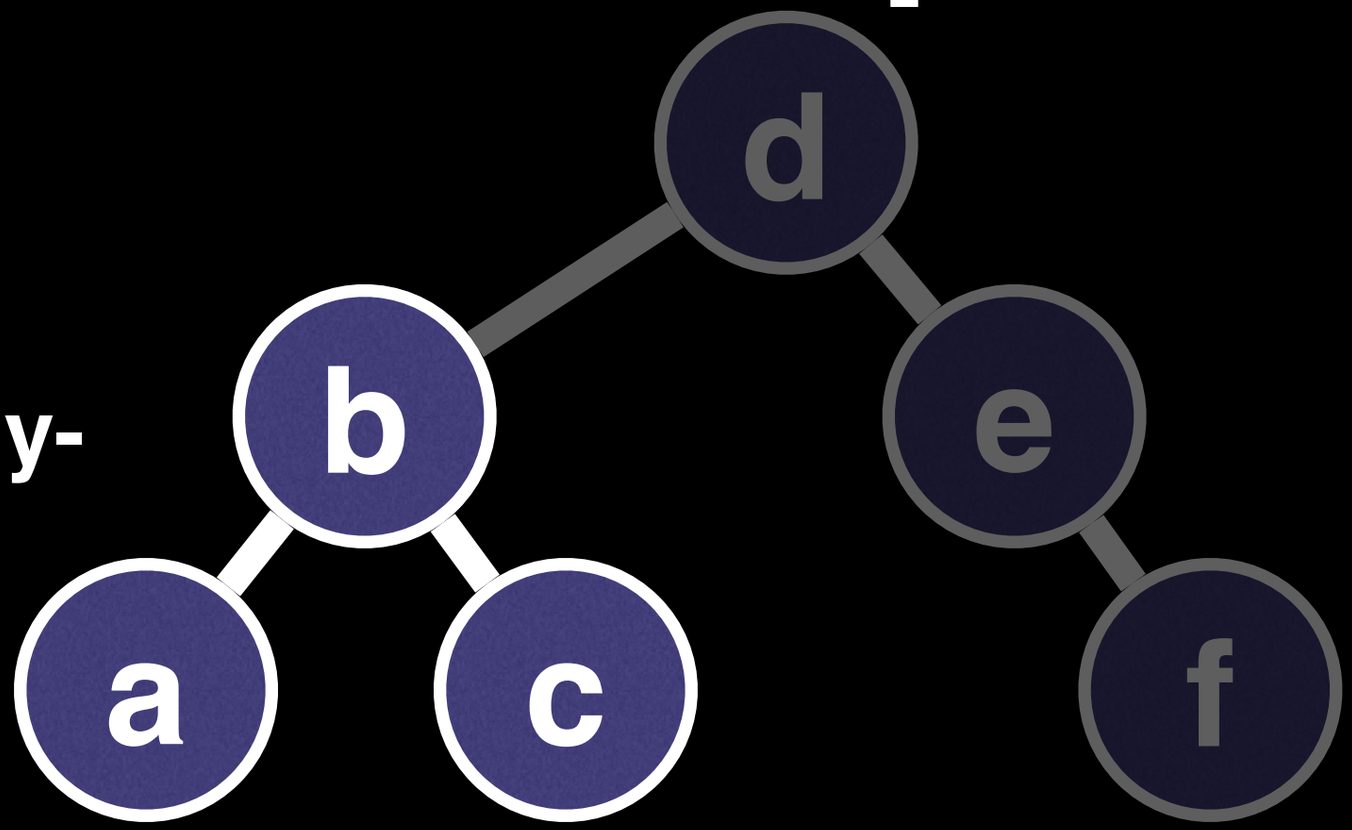
[0, 1, 0, 2, 1, 0]

Linked List

[a, b, c, d, e, f]

Incremental **Sequences**

Probabilistically-Balanced Binary Tree



Levels
CTZ(hash(•))

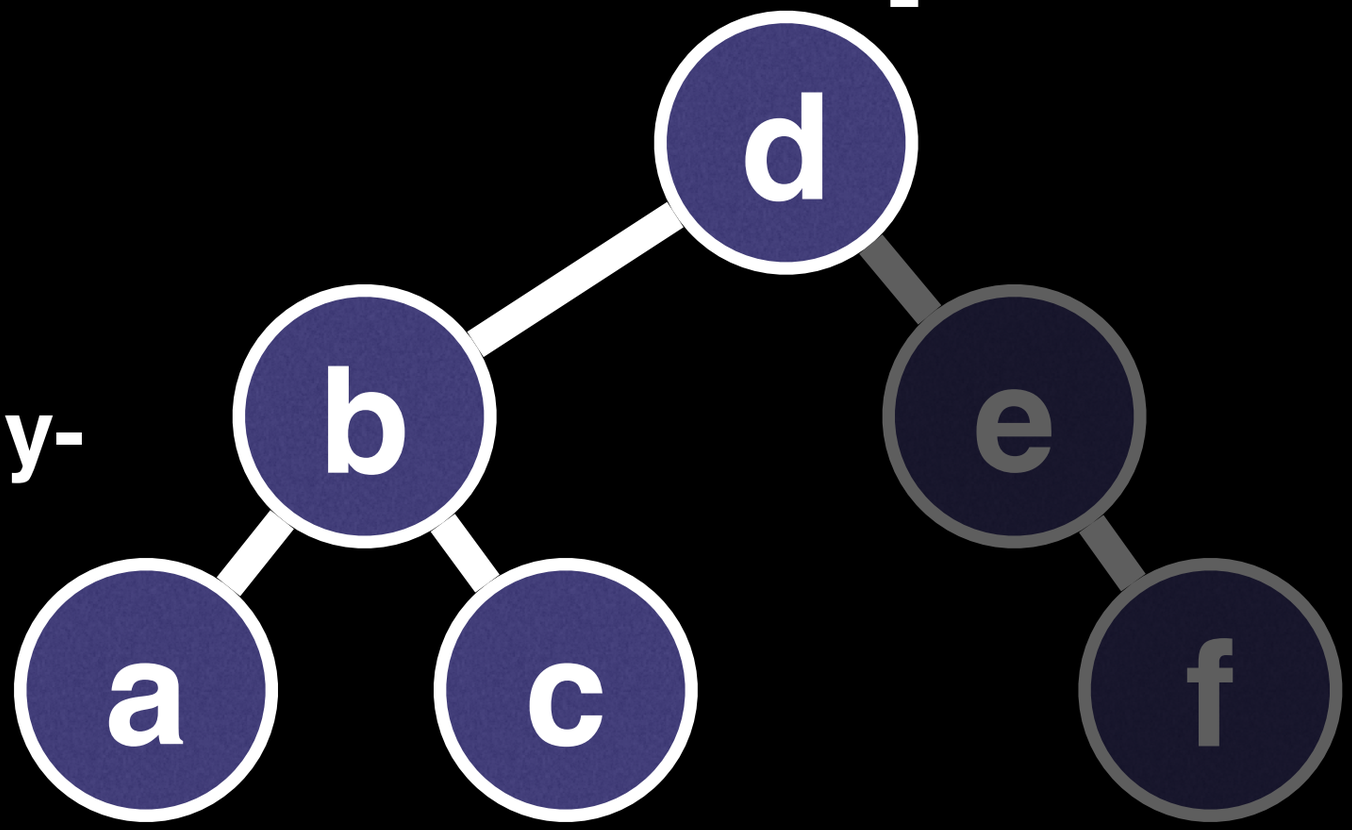
[0, 1, 0, 2, 1, 0]

Linked List

[a, b, c, d, e, f]

Incremental **Sequences**

Probabilistically-Balanced Binary Tree



Levels
CTZ(hash(•))

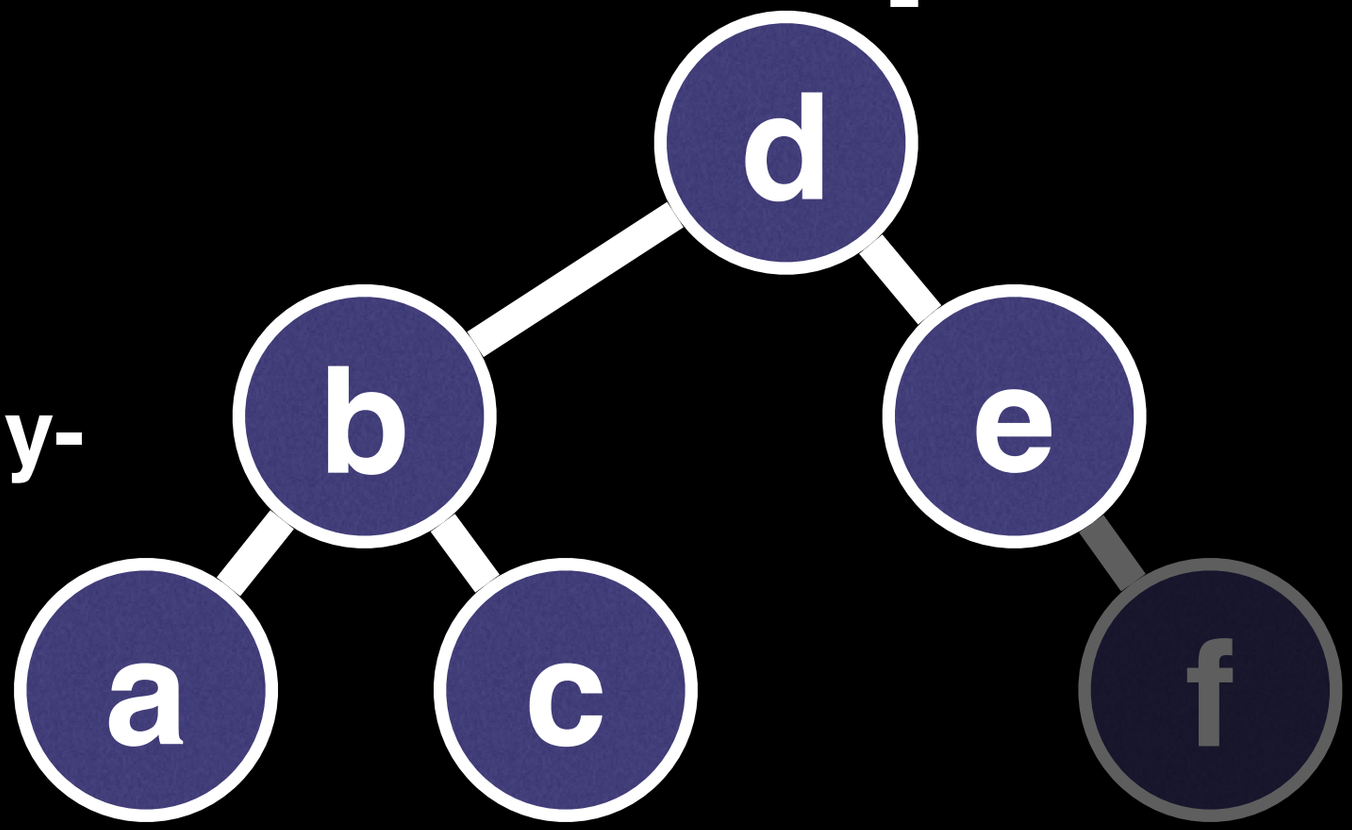
[0, 1, 0, 2, 1, 0]

Linked List

[a, b, c, d, e, f]

Incremental **Sequences**

Probabilistically-Balanced Binary Tree



Levels
CTZ(hash(•))

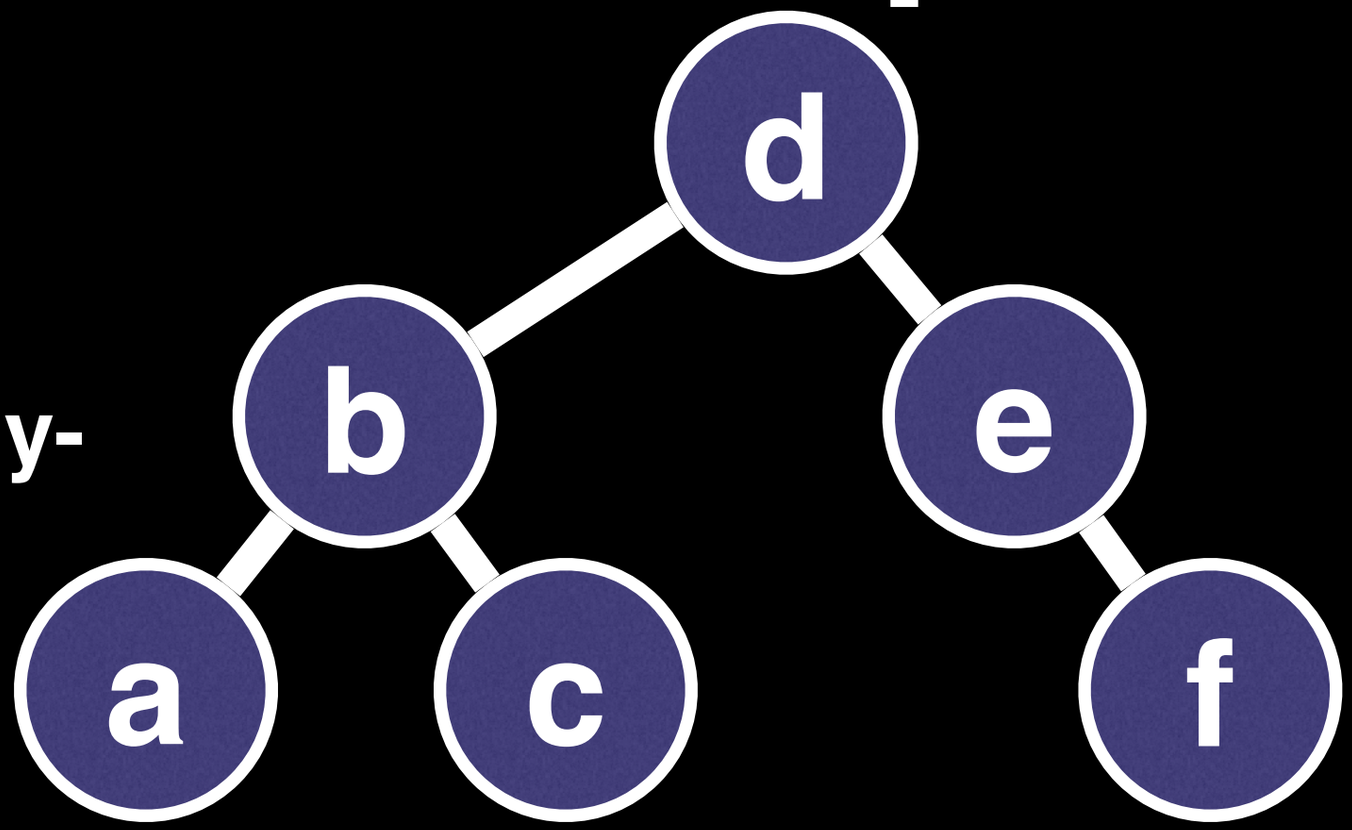
[0, 1, 0, 2, 1, 0]

Linked List

[a, b, c, d, e, f]

Incremental **Sequences**

Probabilistically-Balanced Binary Tree



Levels
CTZ(hash(•))

[0, 1, 0, 2, 1, 0]

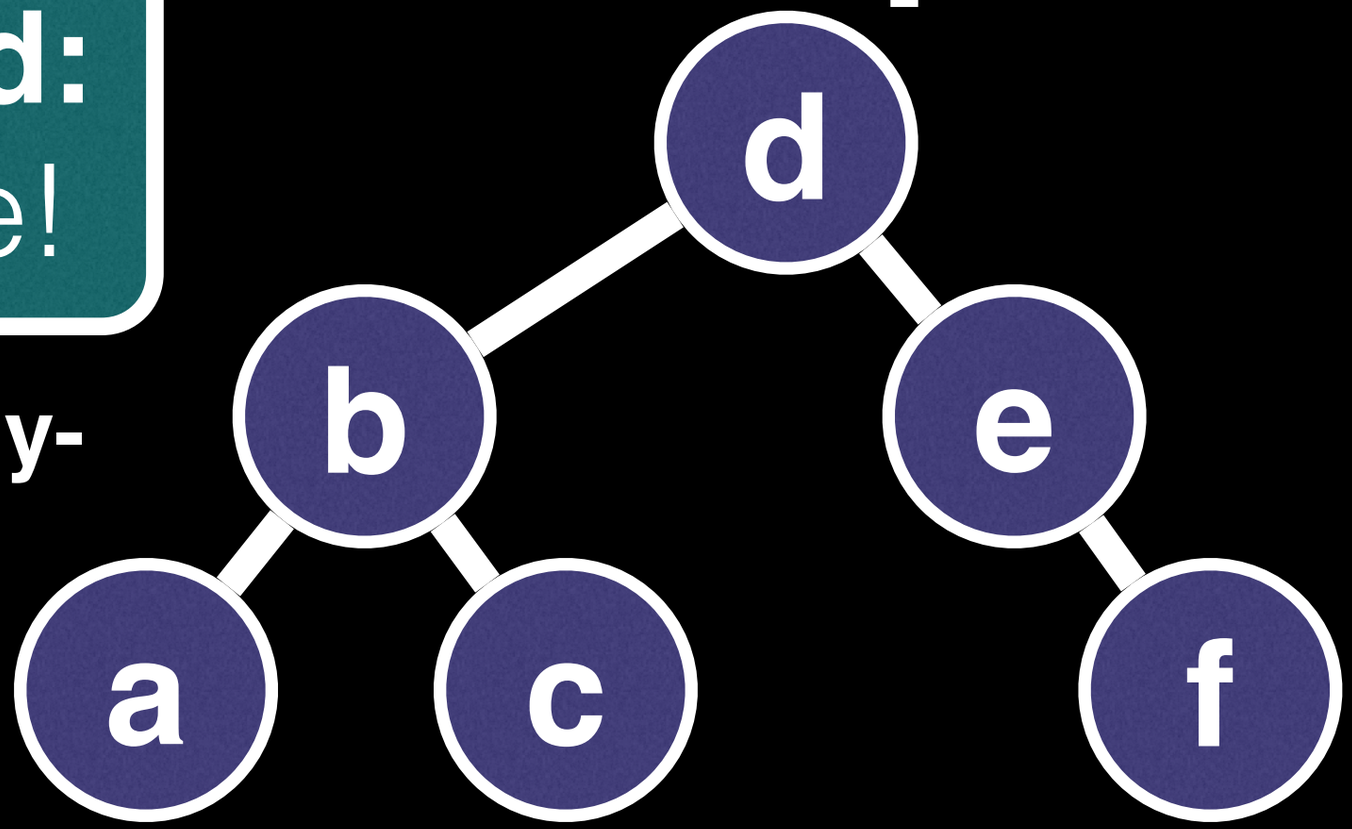
Linked List

[a, b, c, d, e, f]

Incremental **Sequences**

Full Build:
O(n) time!

Probabilistically-Balanced
Binary Tree



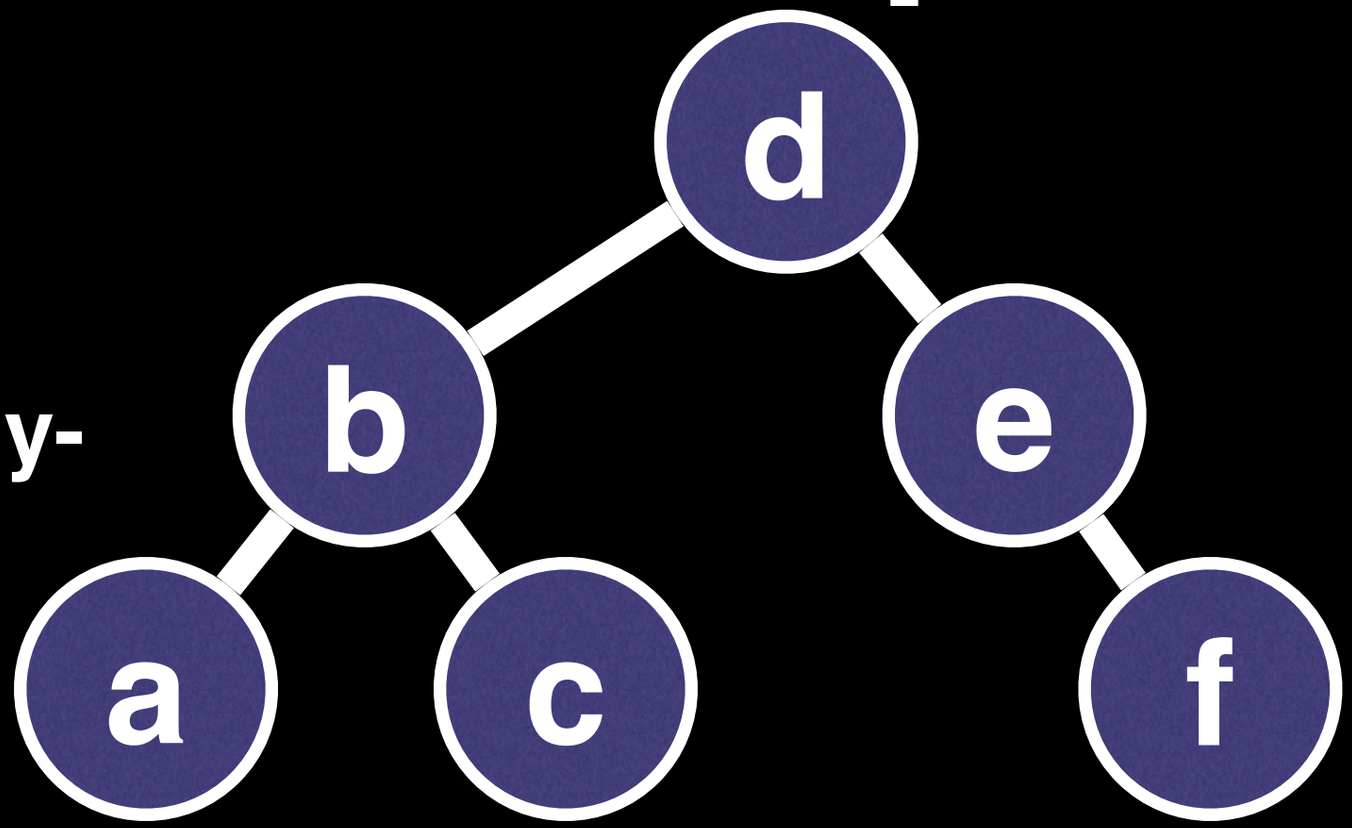
Linked List

[0, 1, 0, 2, 1, 0]

[a, b, c, d, e, f]

Incremental **Sequences**

Probabilistically-Balanced
Binary Tree



Δ
O(1)

Linked List

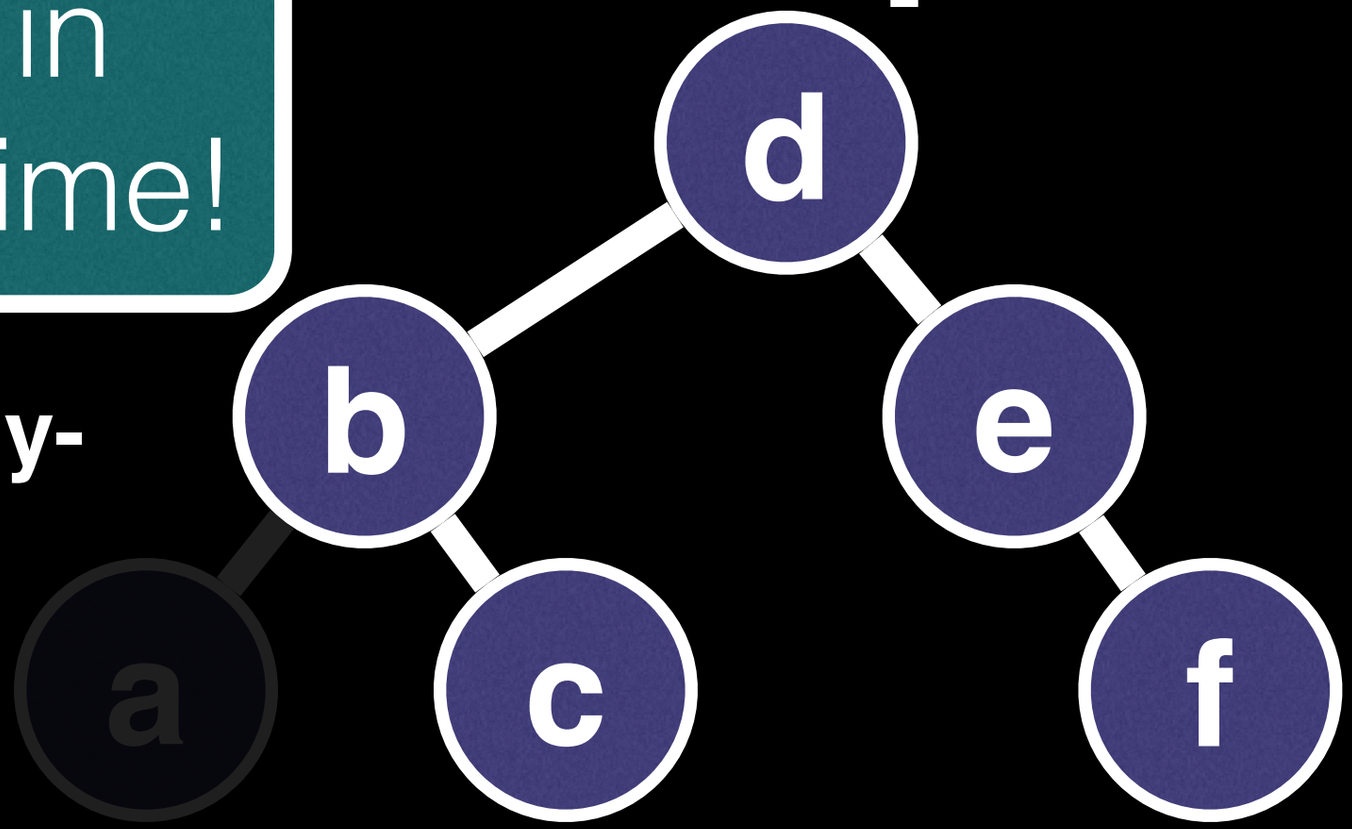
[0, 1, 0, 2, 1, 0]

[a, b, c, d, e, f]

Incremental **Sequences**

Update in
O(log n) time!

Probabilistically-Balanced
Binary Tree



Δ
O(1)

Linked List

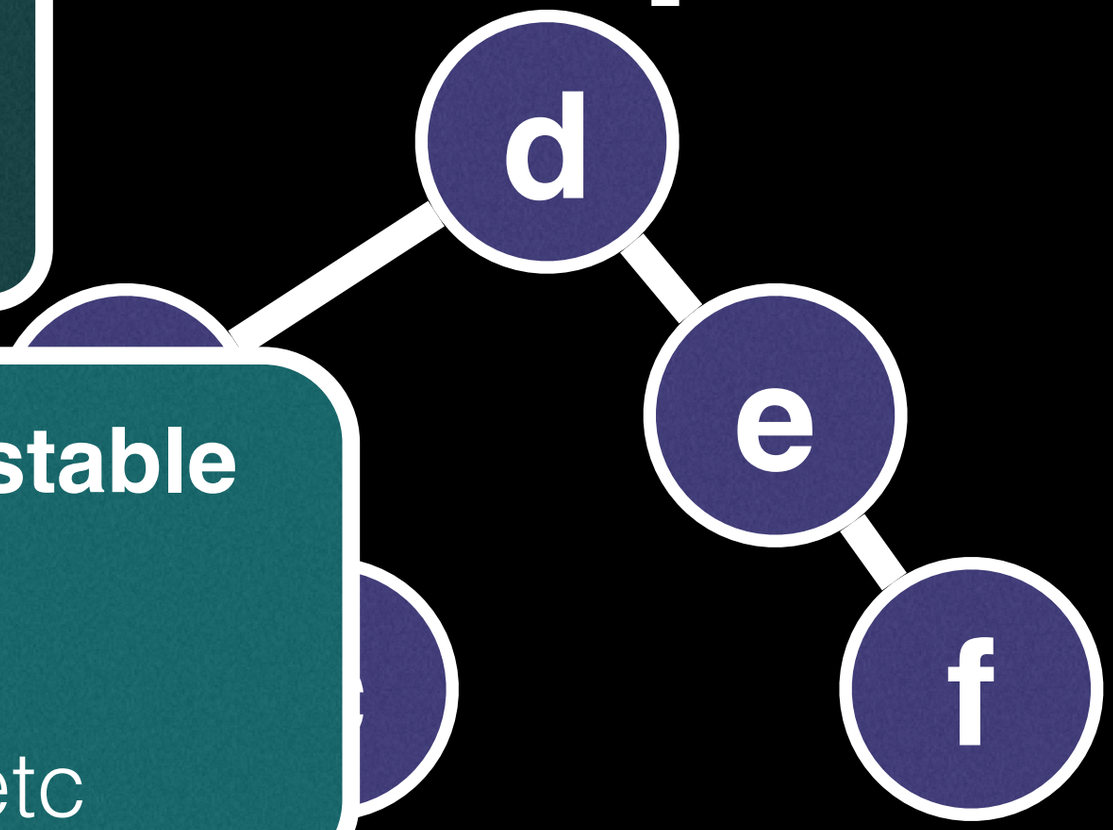
[0, 1, 0, 2, 1, 0]

[a, b, c, d, e, f]

Incremental **Sequences**

Update in
O(log n) time!

Textbook trees **not stable**
Splay trees,
AVL trees,
Red-Black trees, etc



△
O(1)

Linked List

[0, 1, 0, 2, 1, 0]

[a, b, c, d, e, f]

Current Work: "**Auto-Tuning Knobs**" (Overhead-Speedup Tradeoff)

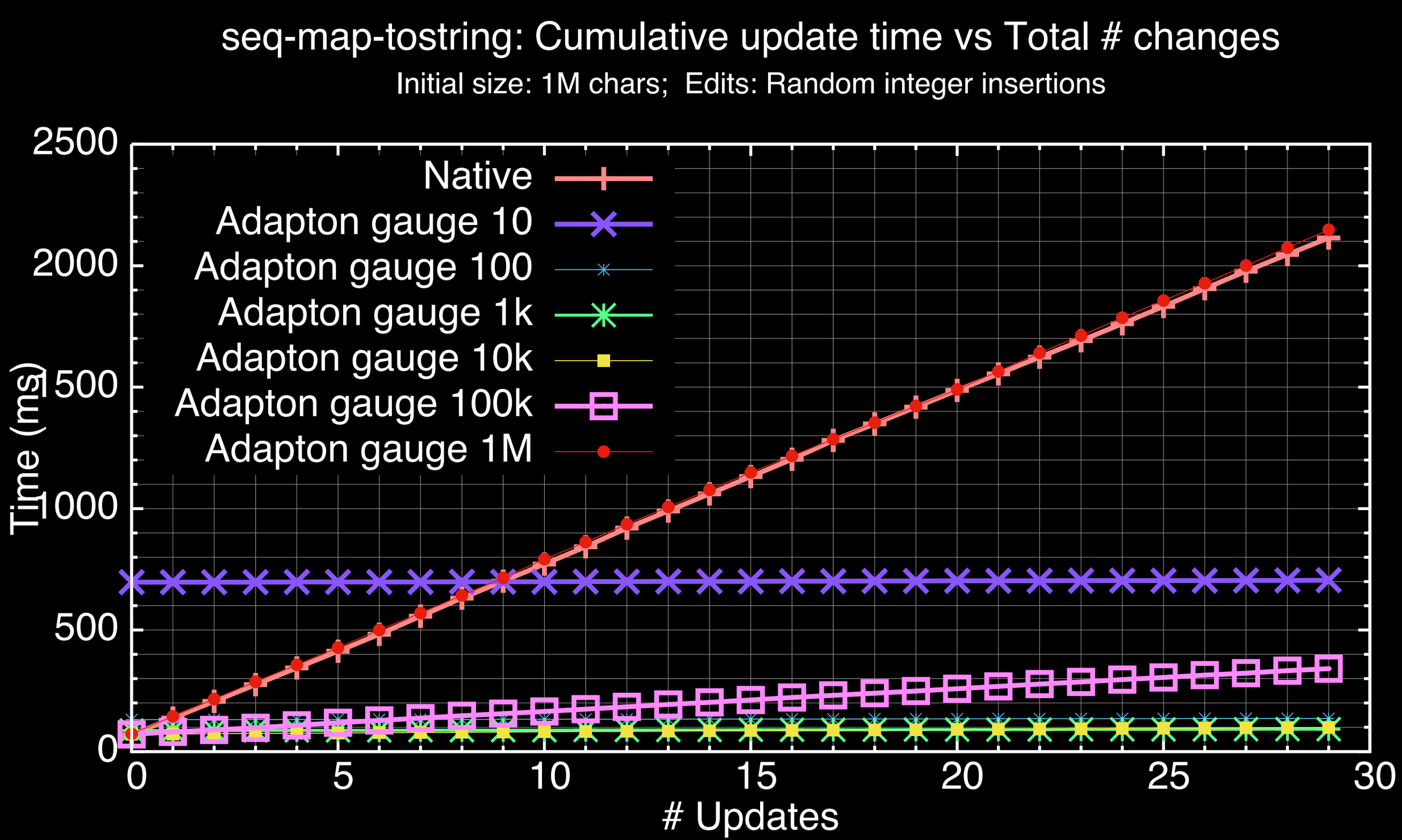
Native Rust:

```
vec.iter().map(|num|format!("{}",num) )
```

Adapton in Rust:

```
level_tree.map(|num|format!("{}",num))
```

Comparison as we vary tree's leaf vector sizes?



Native Rust:

```
vec.iter().map(|num|format!("{}",num) )
```

Incremental Computing with **Adapton**

adapton.org

Summary of Adapton

- **Simple programming model** (cells, thunks)
- Change prop **consistent with from-scratch evaluation**
- **Dynamic graphs**, from **dynamic collections**
- E.g.
  RAZ: Random Access Zipper
  (Persistent sequences)

Ongoing / Future work

- **Auto-tuning collection library**
  (Finite maps, Sets, Graphs)
- DSL for **implicit** incremental computing
- *More applications!*
  - Incremental **Rust compiler**
  - **VM** for **online static analysis**

Incremental Computing with **Adapton**

adapton.org

Summary of Adapton

- **Simple programming model** (cells, thunks)
- Change prop **consistent with from-scratch evaluation**
- **Dynamic graphs**, from **dynamic collections**
- E.g.
  RAZ: Random Access Zipper
  (Persistent sequences)

Ongoing / Future work

- **Auto-tuning collection library**
  (Finite maps, Sets, Graphs)
- DSL for **implicit** incremental computing
- *More applications!*
  - Incremental **Rust compiler**
  - **VM** for **online static analysis**

Incremental Computing with **Adapton**

adapton.org

Thank you!
Questions?

Summary of Adapton

- **Simple programming model** (cells, thunks)
- Change prop **consistent with from-scratch evaluation**
- **Dynamic graphs**, from **dynamic collections**
- E.g.
  RAZ: Random Access Zipper
  (Persistent sequences)

Ongoing / Future work

- **Auto-tuning collection library**
  (Finite maps, Sets, Graphs)
- DSL for **implicit** incremental computing
- *More applications!*
  - Incremental **Rust compiler**
  - **VM** for **online static analysis**

Abridged History of Incremental Computation

Memoization, Dep Graphs, SAC and Adapton

A Brief History of IC

1960s—1990s	2000s	Present, Future
Dynamic Programming / Memo tables **Attribute Grammars** / Dependency Graphs **Special Languages**	**Self-Adjusting Computation***	**Adapton**
	Dynamic Dependency Graphs	Demanded Computation Graphs
Special Domains	**General, Batch Model**	**Demand-Driven, Interactive**
		[PLDI 2014, OOPSLA 2015]

*Acar, Blelloch, Harper, et al

A Brief History of IC

1960s—1990s

Dynamic Programming / Memo tables

Attribute Grammars / Dependency Graphs

Special Languages

Special Domains

2000s

Self-Adjusting Computation*

Dynamic Dependency Graphs

General, Batch Model

Present, Future

Adapton

Demanded Computation Graphs

Demand-Driven, Interactive

[PLDI 2014, OOPSLA 2015]

*Acar, Blelloch, Harper, et al

A Brief History of IC

1960s—1990s

Dynamic Programming / Memo tables

Attribute Grammars / Dependency Graphs

Special Languages

Special Domains

2000s

Self-Adjusting Computation*

Dynamic Dependency Graphs

General, Batch Model

Present, Future

Adapton

Demanded Computation Graphs

Demand-Driven, Interactive

[PLDI 2014, OOPSLA 2015]

*Acar, Blelloch, Harper, et al

Adapton's

Definitions & Meta Theory

Theory

$\boxed{G_1 \vdash^p_\omega e \Downarrow G_2; t}$ Under graph G_1, evaluating expression e as part of thunk p in namespace ω yields G_2 and t.

Rules **common** to the non-incremental and incremental systems:

$$\frac{}{G \vdash^p_\omega t \Downarrow G; t}\ \text{Eval-term} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \lambda x.e_2 \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega e_1\, v \Downarrow G_3; t}\ \text{Eval-app}$$

$$\frac{G_1 \vdash^p_\omega [(\mathbf{fix}\, f.e)/f]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{fix}\, f.e \Downarrow G_2; t}\ \text{Eval-fix} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \mathbf{ret}\, v \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{let}\, x \leftarrow e_1\, \mathbf{in}\, e_2 \Downarrow G_3; t}\ \text{Eval-bind}$$

$$\frac{G_1 \vdash^p_\omega [v/x_i]e_i \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{case}\, (\mathbf{inj}_i\, v, x_1.e_1, x_2.e_2) \Downarrow G_2; t}\ \text{Eval-case} \qquad \frac{G_1 \vdash^p_\omega [v_1/x_1][v_2/x_2]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{split}\, ((v_1, v_2), x_1.x_2.e) \Downarrow G_2; t}\ \text{Eval-split}$$

$$\frac{}{G \vdash^p_\omega \mathbf{fork}(\mathbf{nm}\, k) \Downarrow G; \mathbf{ret}\, (\mathbf{nm}\, k{\cdot}1, \mathbf{nm}\, k{\cdot}2)}\ \text{Eval-fork}$$

$$\frac{G_1 \vdash^p_\omega [\mathbf{ns}\, \omega.k/x]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{ns}\, (\mathbf{nm}\, k, x.e) \Downarrow G_2; t}\ \text{Eval-namespace} \qquad \frac{G_1 \vdash^p_\mu e_1 \Downarrow G_2; \mathbf{ret}\, v \qquad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{nest}(\mathbf{ns}\, \mu, e_1, x.e_2) \Downarrow G_3; t}\ \text{Eval-nest}$$

Rules **specific** to the (non-incremental ‖ incremental) systems:

$$\frac{q = k@\omega \quad q \notin \mathrm{dom}(G_1) \quad G_1\{q \mapsto v\} = G_2}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_2; \mathbf{ret}\, \mathbf{ref}\, q}\ \text{Eval-refPlain}$$

$$\frac{q = k@\omega \quad G_1\{q \mapsto v\} = G_2 \quad \text{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_3, (p, \mathbf{alloc}\, v, \mathrm{clean}, q); \mathbf{ret}\, \mathbf{ref}\, q}\ \text{Eval-refDirty}$$

$$\frac{q = k@\omega \quad G(q) = v}{G \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G, (p, \mathbf{alloc}\, v, \mathrm{clean}, q); \mathbf{ret}\, \mathbf{ref}\, q}\ \text{Eval-refClean}$$

$$\frac{q = k@\omega \quad q \notin \mathrm{dom}(G_1) \quad G_1\{q \mapsto e\} = G_2}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_2; \mathbf{ret}\, (\mathbf{thk}\, q)}\ \text{Eval-thunkPlain}$$

$$\frac{q = k@\omega \quad G_1\{q \mapsto e\} = G_2 \quad \text{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_3, (p, \mathbf{alloc}\, e, \mathrm{clean}, q); \mathbf{ret}\, (\mathbf{thk}\, q)}\ \text{Eval-thunkDirty}$$

$$\frac{q = k@\omega \quad \exp(G, q) = e}{G \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G, (p, \mathbf{alloc}\, e, \mathrm{clean}, q); \mathbf{ret}\, (\mathbf{thk}\, q)}\ \text{Eval-thunkClean}$$

$$\frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\, (\mathbf{ref}\, q) \Downarrow G; \mathbf{ret}\, v}\ \text{Eval-getPlain} \qquad \frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\, (\mathbf{ref}\, q) \Downarrow G, (p, \mathbf{obs}\, v, \mathrm{clean}, q); \mathbf{ret}\, v}\ \text{Eval-getClean}$$

$$\frac{G_1(q) = e \quad G_1 \vdash^q_{\mathrm{namespace}(q)} e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, q) \Downarrow G_2; t}\ \text{Eval-forcePlain}$$

$$\frac{G(q) = (e, t) \quad \text{all-clean-out}(G, q)}{G \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, q) \Downarrow G, (p, \mathbf{obs}\, t, \mathrm{clean}, q); t}\ \text{Eval-forceClean}$$

$$\frac{\begin{array}{l}\text{all-clean-out}((G_1, G_2), q_2) \\ \text{consistent-action}((G_1, G_2), a, q_2) \\ G_1, (q_1, a, \mathrm{clean}, q_2), G_2 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t\end{array}}{G_1, (q_1, a, \mathrm{dirty}, q_2), G_2 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t}\ \text{Eval-scrubEdge}$$

$$\frac{\begin{array}{ll}\exp(G_1, q) = e' & G_2\{q \mapsto (e', t')\} = G_2' \\ \text{del-edges-out}(G_1\{q \mapsto e'\}, q) = G_1' & \text{all-clean-out}(G_2', q) \\ G_1' \vdash^q_{\mathrm{namespace}(q)} e' \Downarrow G_2; t' & G_2' \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t\end{array}}{G_1 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t}\ \text{Eval-computeDep}$$

Figure 5: Evaluation rules of λ_{NomA}; vertical bars separate non-incremental rules (left, shaded) from incremental rules (right)

One Page of Evaluation Rules

Theory

$\boxed{G_1 \vdash^p_\omega e \Downarrow G_2; t}$ Under graph G_1, evaluating expression e as part of thunk p in namespace ω yields G_2 and t.

Rules **common** to the non-incremental and incremental systems:

$$\frac{}{G \vdash^p_\omega t \Downarrow G; t}\ \text{Eval-term} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \lambda x.e_2 \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega e_1\ v \Downarrow G_3; t}\ \text{Eval-app}$$

$$\frac{G_1 \vdash^p_\omega [(\mathbf{fix}\, f.e)/f]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{fix}\, f.e \Downarrow G_2; t}\ \text{Eval-fix} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \mathbf{ret}\, v \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{let}\, x \leftarrow e_1\ \mathbf{in}\ e_2 \Downarrow G_3; t}\ \text{Eval-bind}$$

$$\frac{G_1 \vdash^p_\omega [v/x_i]e_i \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{case}\,(\mathbf{inj}_i\, v, x_1.e_1, x_2.e_2) \Downarrow G_2; t}\ \text{Eval-case} \qquad \frac{G_1 \vdash^p_\omega [v_1/x_1][v_2/x_2]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{split}\,((v_1, v_2), x_1.x_2.e) \Downarrow G_2; t}\ \text{Eval-split}$$

$$\frac{}{G \vdash^p_\omega \mathbf{fork}(\mathbf{nm}\, k) \Downarrow G; \mathbf{ret}\,(\mathbf{nm}\, k{\cdot}1, \mathbf{nm}\, k{\cdot}2)}\ \text{Eval-fork}$$

$$\frac{G_1 \vdash^p_\omega [\mathbf{ns}\, \omega.k/x]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{ns}\,(\mathbf{nm}\, k, x.e) \Downarrow G_2; t}\ \text{Eval-namespace} \qquad \frac{G_1 \vdash^p_\mu e_1 \Downarrow G_2; \mathbf{ret}\, v \qquad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{nest}(\mathbf{ns}\, \mu, e_1, x.e_2) \Downarrow G_3; t}\ \text{Eval-nest}$$

Rules **specific** to the (non-incremental ‖ incremental) systems:

$$\frac{q = k@\omega \quad q \notin \mathrm{dom}(G_1) \qquad G_1\{q \mapsto v\} = G_2}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_2; \mathbf{ret}\ \mathbf{ref}\ q}\ \text{Eval-refPlain}$$

$$\frac{q = k@\omega \qquad G_1\{q \mapsto v\} = G_2 \qquad \text{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_3, (p, \mathbf{alloc}\, v, \text{clean}, q); \mathbf{ret}\ \mathbf{ref}\ q}\ \text{Eval-refDirty}$$

$$\frac{q = k@\omega \qquad G(q) = v}{G \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G, (p, \mathbf{alloc}\, v, \text{clean}, q); \mathbf{ret}\ \mathbf{ref}\ q}\ \text{Eval-refClean}$$

$$\frac{q = k@\omega \quad q \notin \mathrm{dom}(G_1) \qquad G_1\{q \mapsto e\} = G_2}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_2; \mathbf{ret}\,(\mathbf{thk}\ q)}\ \text{Eval-thunkPlain}$$

$$\frac{q = k@\omega \qquad G_1\{q \mapsto e\} = G_2 \qquad \text{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_3, (p, \mathbf{alloc}\, e, \text{clean}, q); \mathbf{ret}\,(\mathbf{thk}\ q)}\ \text{Eval-thunkDirty}$$

$$\frac{q = k@\omega \qquad \exp(G, q) = e}{G \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G, (p, \mathbf{alloc}\, e, \text{clean}, q); \mathbf{ret}\,(\mathbf{thk}\ q)}\ \text{Eval-thunkClean}$$

$$\frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\,(\mathbf{ref}\ q) \Downarrow G; \mathbf{ret}\, v}\ \text{Eval-getPlain} \qquad \frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\,(\mathbf{ref}\ q) \Downarrow G, (p, \mathbf{obs}\, v, \text{clean}, q); \mathbf{ret}\, v}\ \text{Eval-getClean}$$

$$\frac{G_1(q) = e \qquad G_1 \vdash^q_{\text{namespace}(q)} e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{force}\,(\mathbf{thk}\ q) \Downarrow G_2; t}\ \text{Eval-forcePlain}$$

$$\frac{G(q) = (e, t) \qquad \text{all-clean-out}(G, q)}{G \vdash^p_\omega \mathbf{force}\,(\mathbf{thk}\ q) \Downarrow G, (p, \mathbf{obs}\, t, \text{clean}, q); t}\ \text{Eval-forceClean}$$

$$\frac{\begin{array}{l}\text{all-clean-out}((G_1, G_2), q_2) \\ \text{consistent-action}((G_1, G_2), a, q_2) \\ G_1, (q_1, a, \text{clean}, q_2), G_2 \vdash^p_\omega \mathbf{force}\,(\mathbf{thk}\ p_0) \Downarrow G_3; t\end{array}}{G_1, (q_1, a, \text{dirty}, q_2), G_2 \vdash^p_\omega \mathbf{force}\,(\mathbf{thk}\ p_0) \Downarrow G_3; t}\ \text{Eval-scrubEdge}$$

$$\frac{\begin{array}{ll}\exp(G_1, q) = e' & G_2\{q \mapsto (e', t')\} = G_2' \\ \text{del-edges-out}(G_1\{q \mapsto e'\}, q) = G_1' & \text{all-clean-out}(G_2', q) \\ G_1' \vdash^q_{\text{namespace}(q)} e' \Downarrow G_2; t' & G_2' \vdash^p_\omega \mathbf{force}\,(\mathbf{thk}\ p_0) \Downarrow G_3; t\end{array}}{G_1 \vdash^p_\omega \mathbf{force}\,(\mathbf{thk}\ p_0) \Downarrow G_3; t}\ \text{Eval-computeDep}$$

Figure 5: Evaluation rules of λ_{NomA}; vertical bars separate non-incremental rules (left, shaded) from incremental rules (right)

From-Scratch Evaluation

Spec for Incremental Eval.

No Memoization,

No Dependency Graphs

Incremental Evaluation

Spec for Impl. Algorithms:

Spec for Memoization

Spec Dependency Graphs

Theory

$\boxed{G_1 \vdash^p_\omega e \Downarrow G_2; t}$ Under graph G_1, evaluating expression e as part of thunk p in namespace ω yields G_2 and t.

Rules **common** to the non-incremental and incremental systems:

$$\frac{}{G \vdash^p_\omega t \Downarrow G; t}\ \text{Eval-term} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \lambda x.e_2 \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega e_1\, v \Downarrow G_3; t}\ \text{Eval-app}$$

$$\frac{G_1 \vdash^p_\omega [(\mathbf{fix}\, f.e)/f]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{fix}\, f.e \Downarrow G_2; t}\ \text{Eval-fix} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \mathbf{ret}\, v \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{let}\, x \leftarrow e_1\, \mathbf{in}\, e_2 \Downarrow G_3; t}\ \text{Eval-bind}$$

$$\frac{G_1 \vdash^p_\omega [v/x_i]e_i \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{case}\, (\mathbf{inj}_i\, v, x_1.e_1, x_2.e_2) \Downarrow G_2; t}\ \text{Eval-case} \qquad \frac{G_1 \vdash^p_\omega [v_1/x_1][v_2/x_2]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{split}\, ((v_1, v_2), x_1.x_2.e) \Downarrow G_2; t}\ \text{Eval-split}$$

$$\frac{}{G \vdash^p_\omega \mathbf{fork}(\mathbf{nm}\, k) \Downarrow G; \mathbf{ret}\, (\mathbf{nm}\, k{\cdot}1, \mathbf{nm}\, k{\cdot}2)}\ \text{Eval-fork}$$

$$\frac{G_1 \vdash^p_\omega [\mathbf{ns}\, \omega.k/x]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{ns}\, (\mathbf{nm}\, k, x.e) \Downarrow G_2; t}\ \text{Eval-namespace} \qquad \frac{G_1 \vdash^p_\mu e_1 \Downarrow G_2; \mathbf{ret}\, v \qquad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{nest}(\mathbf{ns}\, \mu, e_1, x.e_2) \Downarrow G_3; t}\ \text{Eval-nest}$$

Rules **specific** to the (non-incremental ‖ incremental) systems:

$$\frac{q = k@\omega \quad q \notin \mathrm{dom}(G_1) \quad G_1\{q \mapsto v\} = G_2}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_2; \mathbf{ret}\, \mathbf{ref}\, q}\ \text{Eval-refPlain}$$

$$\frac{q = k@\omega \quad G_1\{q \mapsto v\} = G_2 \quad \text{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_3, (p, \mathbf{alloc}\, v, \text{clean}, q); \mathbf{ret}\, \mathbf{ref}\, q}\ \text{Eval-refDirty}$$

$$\frac{q = k@\omega \quad G(q) = v}{G \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G, (p, \mathbf{alloc}\, v, \text{clean}, q); \mathbf{ret}\, \mathbf{ref}\, q}\ \text{Eval-refClean}$$

$$\frac{q = k@\omega \quad q \notin \mathrm{dom}(G_1) \quad G_1\{q \mapsto e\} = G_2}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_2; \mathbf{ret}\, (\mathbf{thk}\, q)}\ \text{Eval-thunkPlain}$$

$$\frac{q = k@\omega \quad G_1\{q \mapsto e\} = G_2 \quad \text{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_3, (p, \mathbf{alloc}\, e, \text{clean}, q); \mathbf{ret}\, (\mathbf{thk}\, q)}\ \text{Eval-thunkDirty}$$

$$\frac{q = k@\omega \quad \exp(G, q) = e}{G \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G, (p, \mathbf{alloc}\, e, \text{clean}, q); \mathbf{ret}\, (\mathbf{thk}\, q)}\ \text{Eval-thunkClean}$$

$$\frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\, (\mathbf{ref}\, q) \Downarrow G; \mathbf{ret}\, v}\ \text{Eval-getPlain} \qquad \frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\, (\mathbf{ref}\, q) \Downarrow G, (p, \mathbf{obs}\, v, \text{clean}, q); \mathbf{ret}\, v}\ \text{Eval-getClean}$$

$$\frac{G_1(q) = e \quad G_1 \vdash^q_{\text{namespace}(q)} e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, q) \Downarrow G_2; t}\ \text{Eval-forcePlain}$$

$$\frac{G(q) = (e, t) \quad \text{all-clean-out}(G, q)}{G \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, q) \Downarrow G, (p, \mathbf{obs}\, t, \text{clean}, q); t}\ \text{Eval-forceClean}$$

$$\frac{\begin{array}{l}\text{all-clean-out}((G_1, G_2), q_2) \\ \text{consistent-action}((G_1, G_2), a, q_2) \\ G_1, (q_1, a, \text{clean}, q_2), G_2 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t\end{array}}{G_1, (q_1, a, \text{dirty}, q_2), G_2 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t}\ \text{Eval-scrubEdge}$$

$$\frac{\begin{array}{ll}\exp(G_1, q) = e' & G_2\{q \mapsto (e', t')\} = G_2' \\ \text{del-edges-out}(G_1\{q \mapsto e'\}, q) = G_1' & \text{all-clean-out}(G_2', q) \\ G_1' \vdash^q_{\text{namespace}(q)} e' \Downarrow G_2; t' & G_2' \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t\end{array}}{G_1 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t}\ \text{Eval-computeDep}$$

Figure 5: Evaluation rules of λ_{NomA}; vertical bars separate non-incremental rules (left, shaded) from incremental rules (right)

From-Scratch Evaluation

Spec for Incremental Eval.

No Memoization,

No Dependency Graphs

Incremental Evaluation

Spec for Impl. Algorithms:

Spec for Memoization

Spec Dependency Graphs

Implementation Algorithms

Memoization,

Dependency Graphs

Meta Theory

Theorem:
From-Scratch Consistency

From-Scratch Evaluation
Spec for Incremental Eval.
No Memoization,
No Dependency Graphs

Incremental Evaluation
Spec for Impl. Algorithms:
Spec for Memoization
Spec Dependency Graphs

Implementation Algorithms
Memoization,
Dependency Graphs

Meta Theory

Theorem:
From-Scratch Consistency

From-Scratch Evaluation
Spec for Incremental Eval.
No Memoization,
No Dependency Graphs

Incremental Evaluation
Spec for Impl. Algorithms:
Spec for Memoization
Spec Dependency Graphs

Agree
(Informally)

Implementation Algorithms
Memoization,
Dependency Graphs