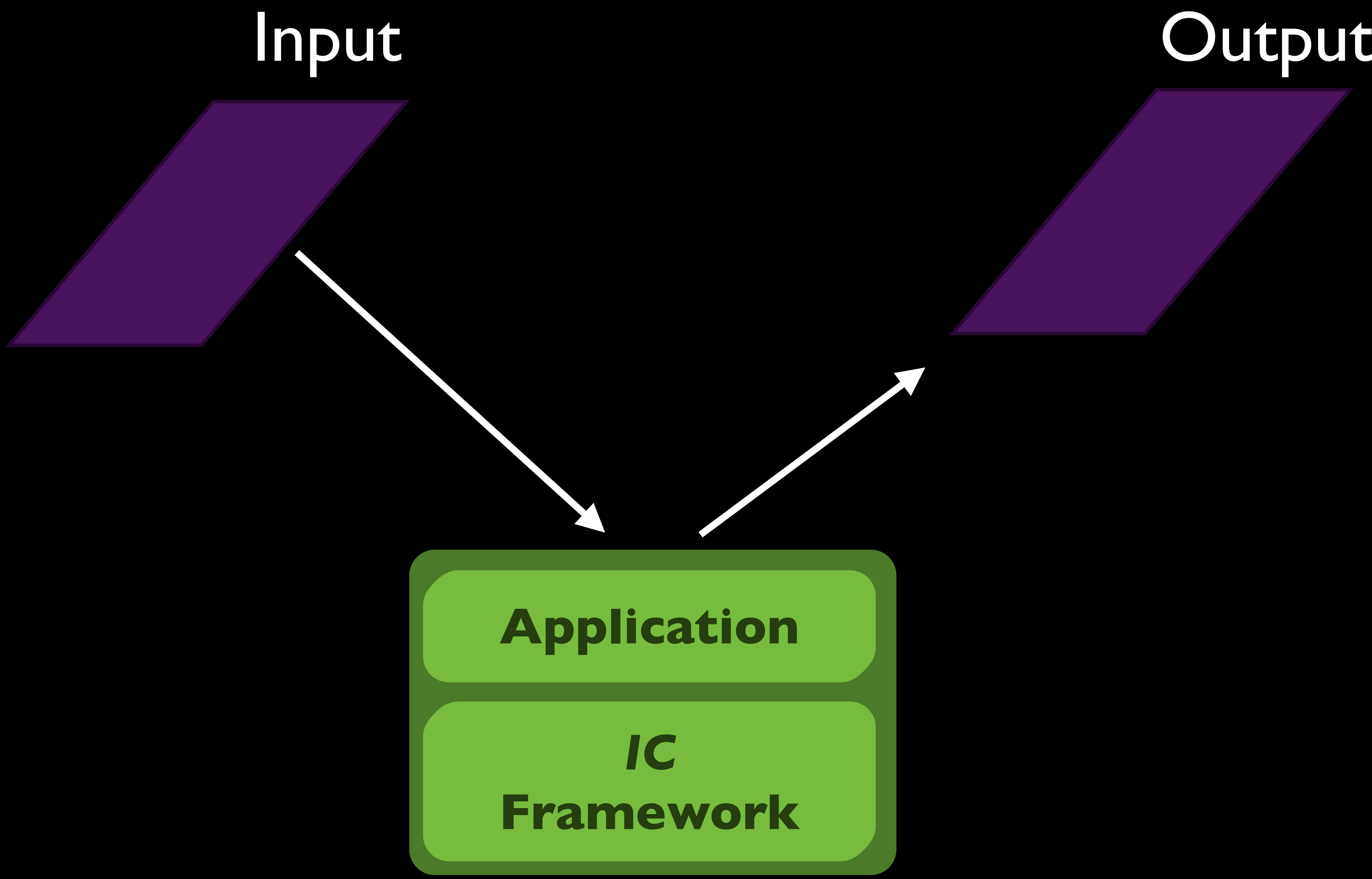
Adapton:
Composable Demand-Driven Incremental Computation

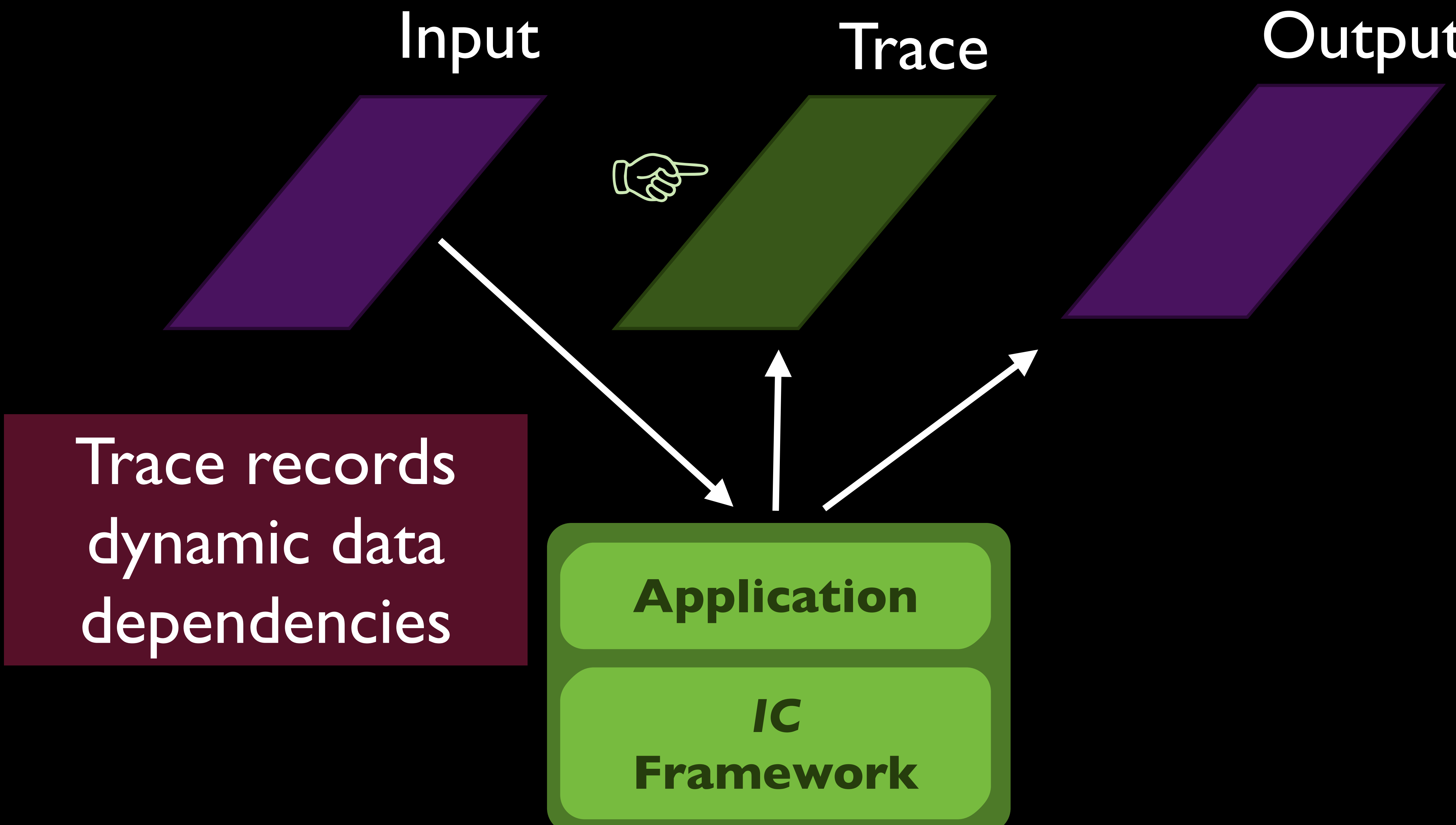
Matthew A. Hammer
Khoo Yit Phang, Michael Hicks *and* Jeffrey S. Foster

PROGRAMMING LANGUAGES UNIVERSITY OF MARYLAND

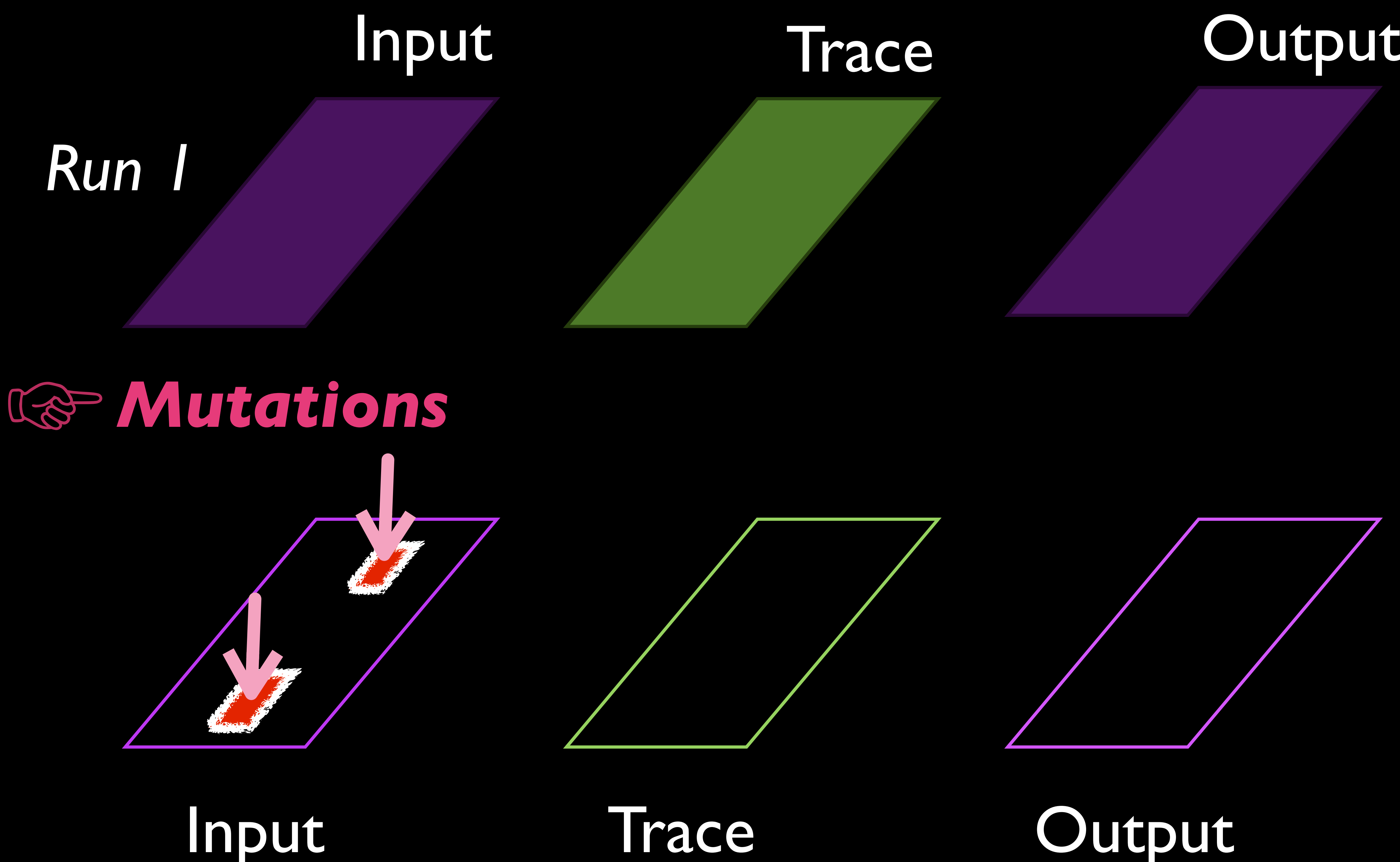
Incremental Computation



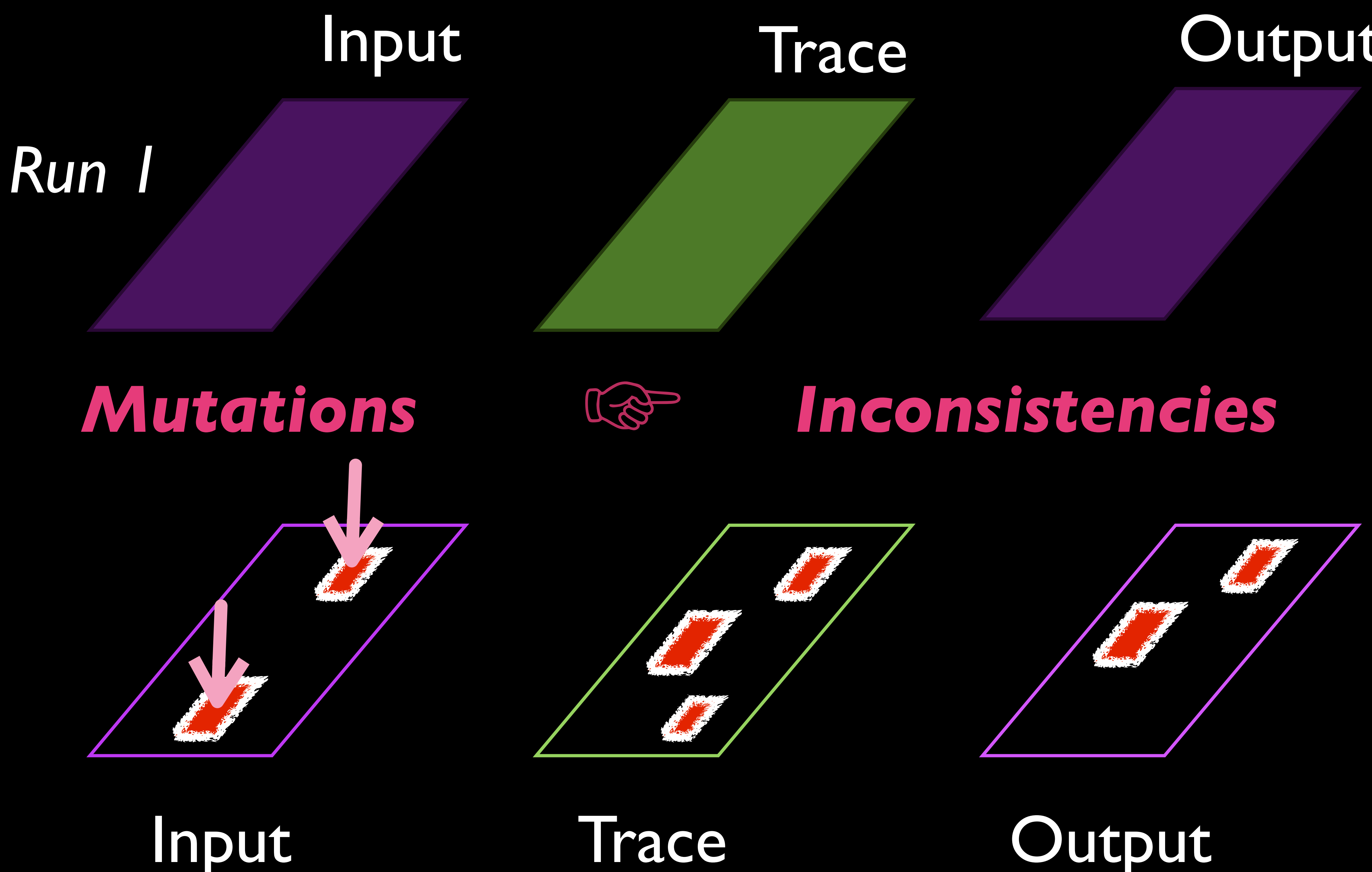
Incremental Computation



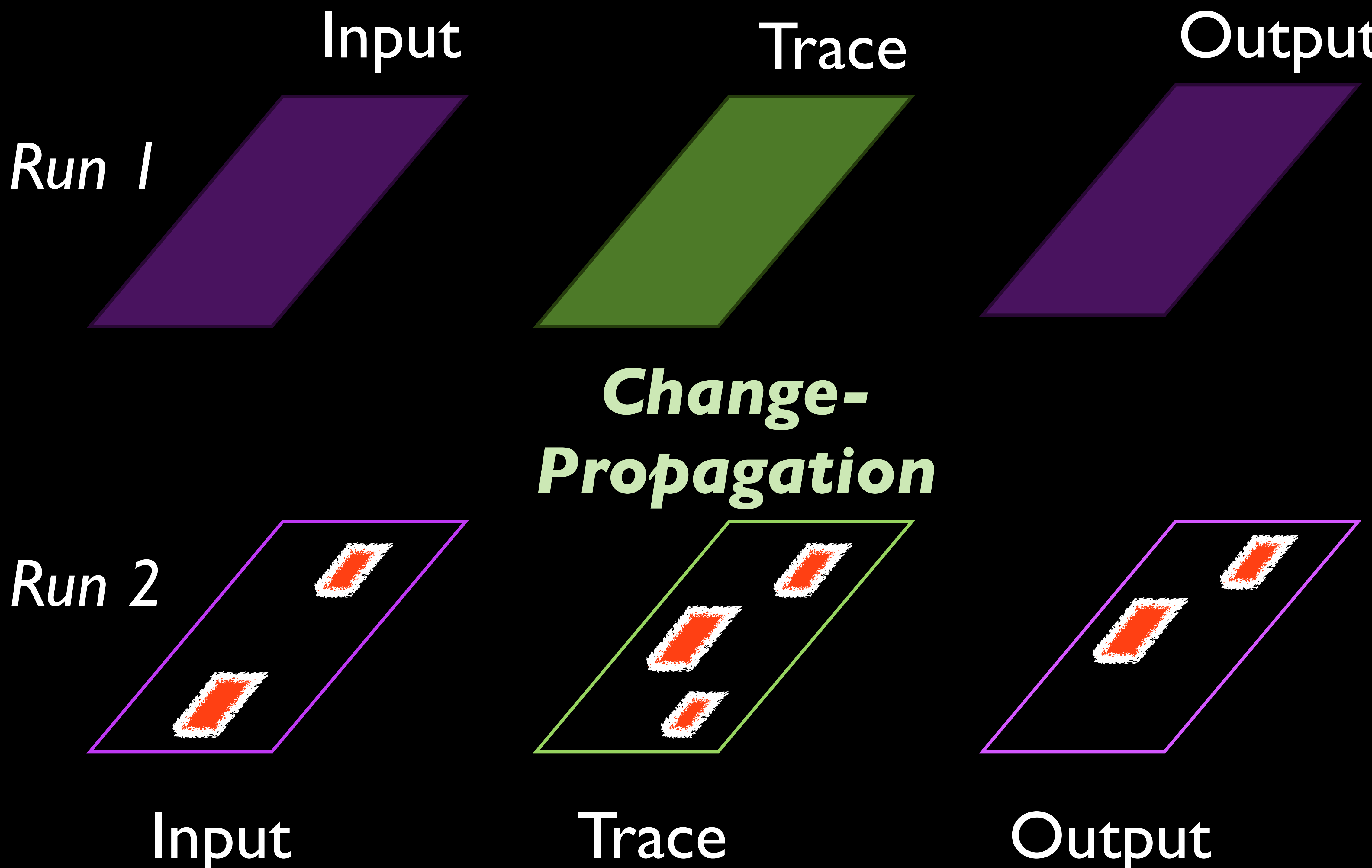
Incremental Computation



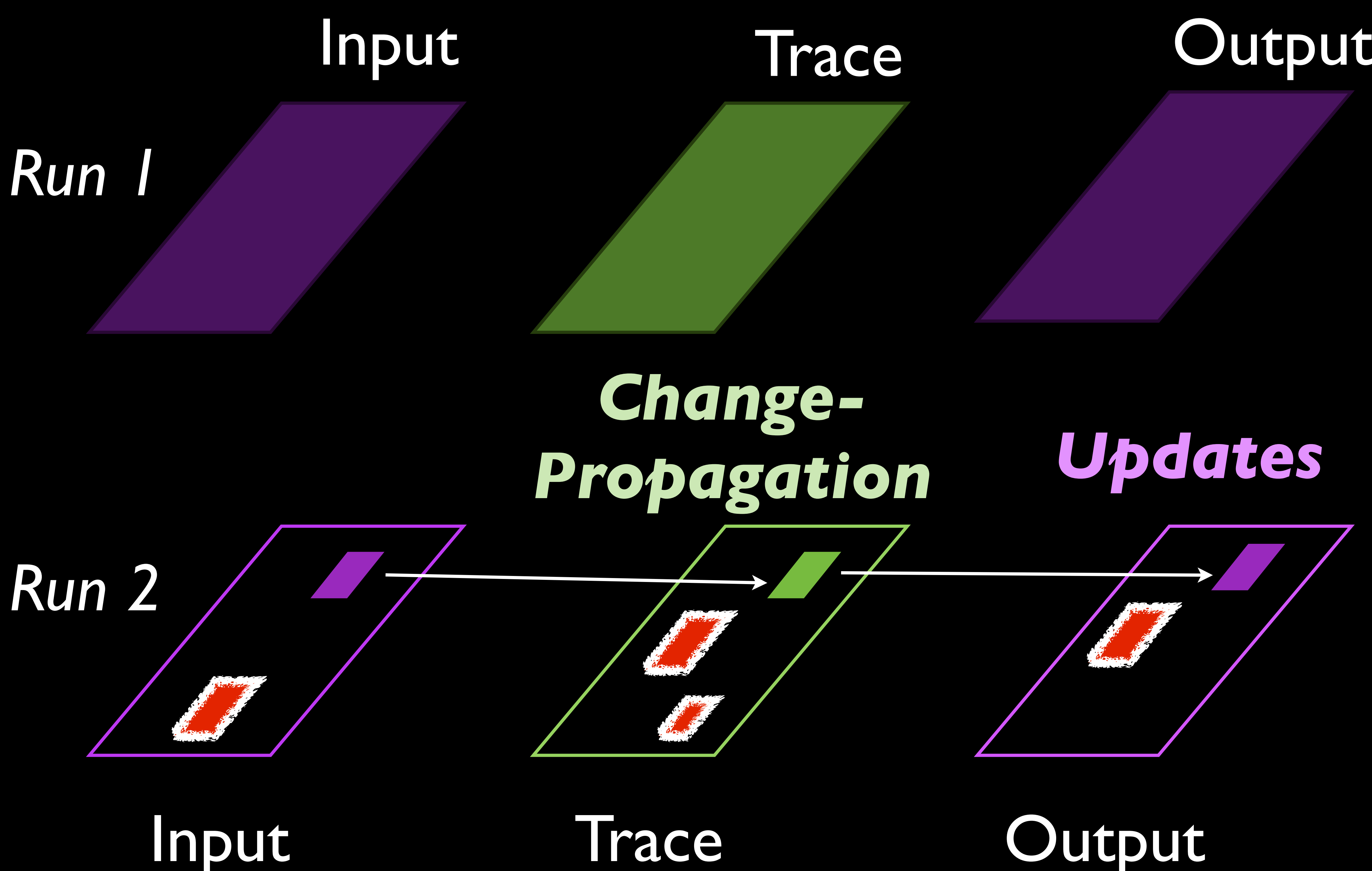
Incremental Computation



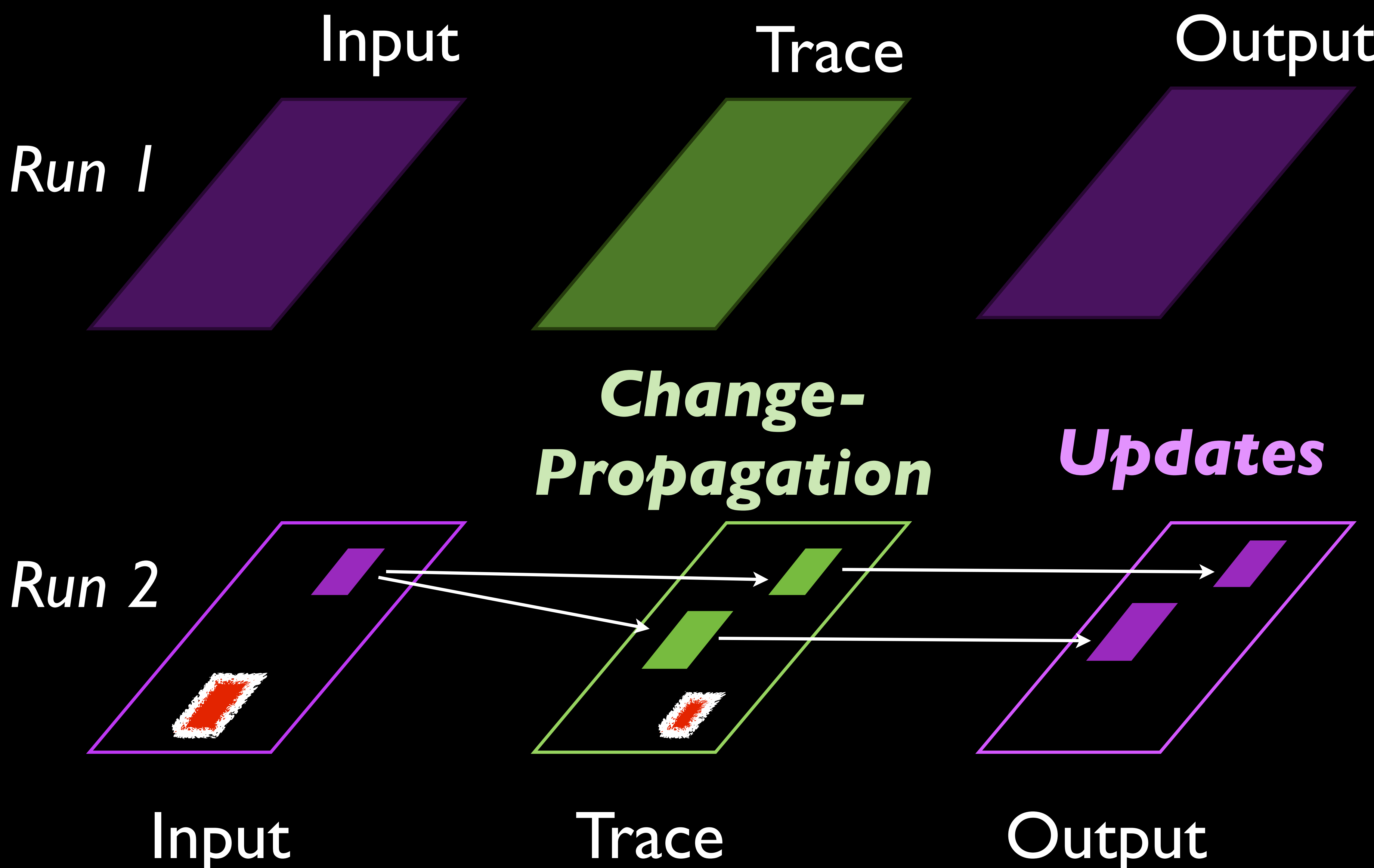
Incremental Computation



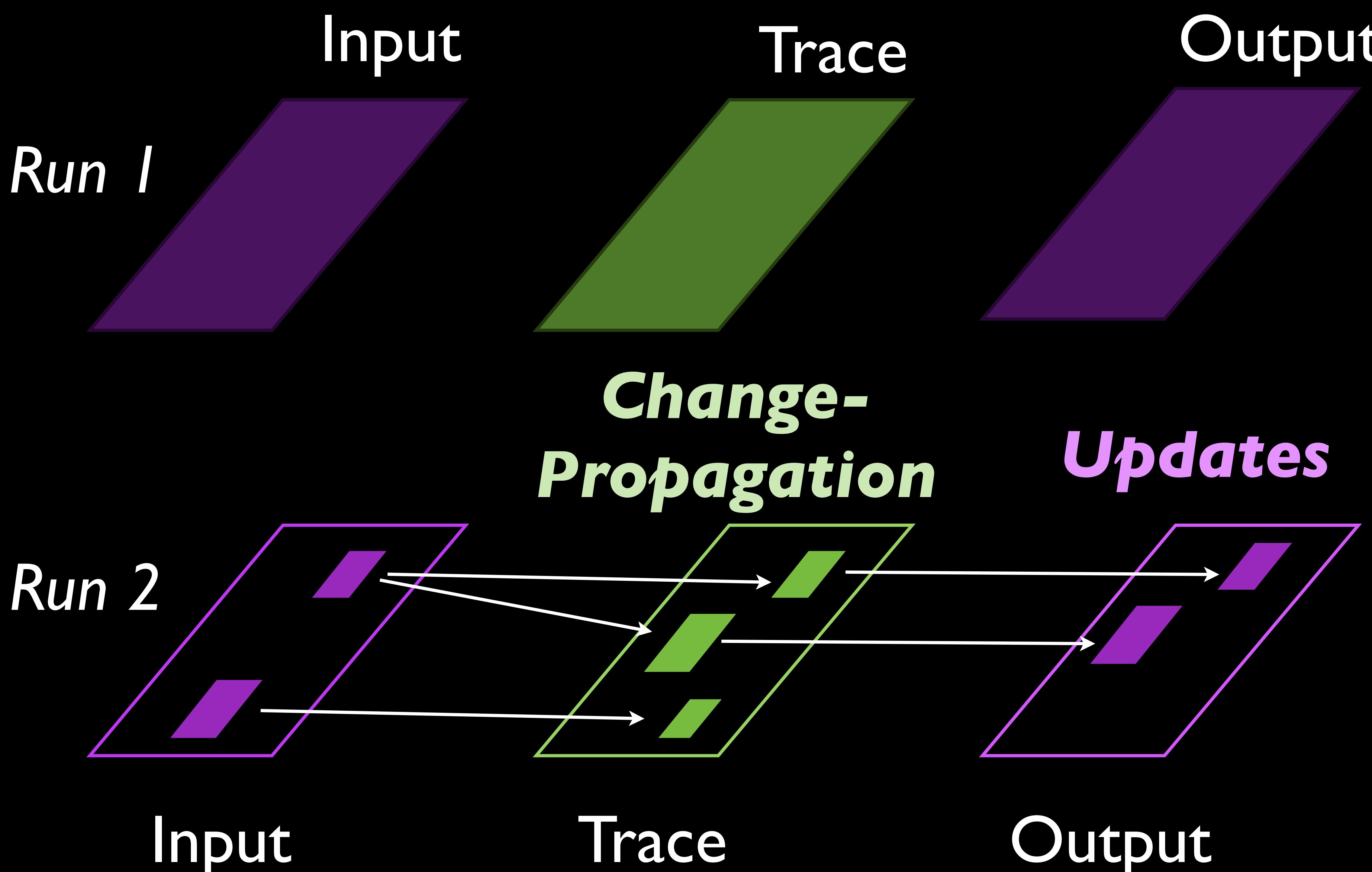
Incremental Computation



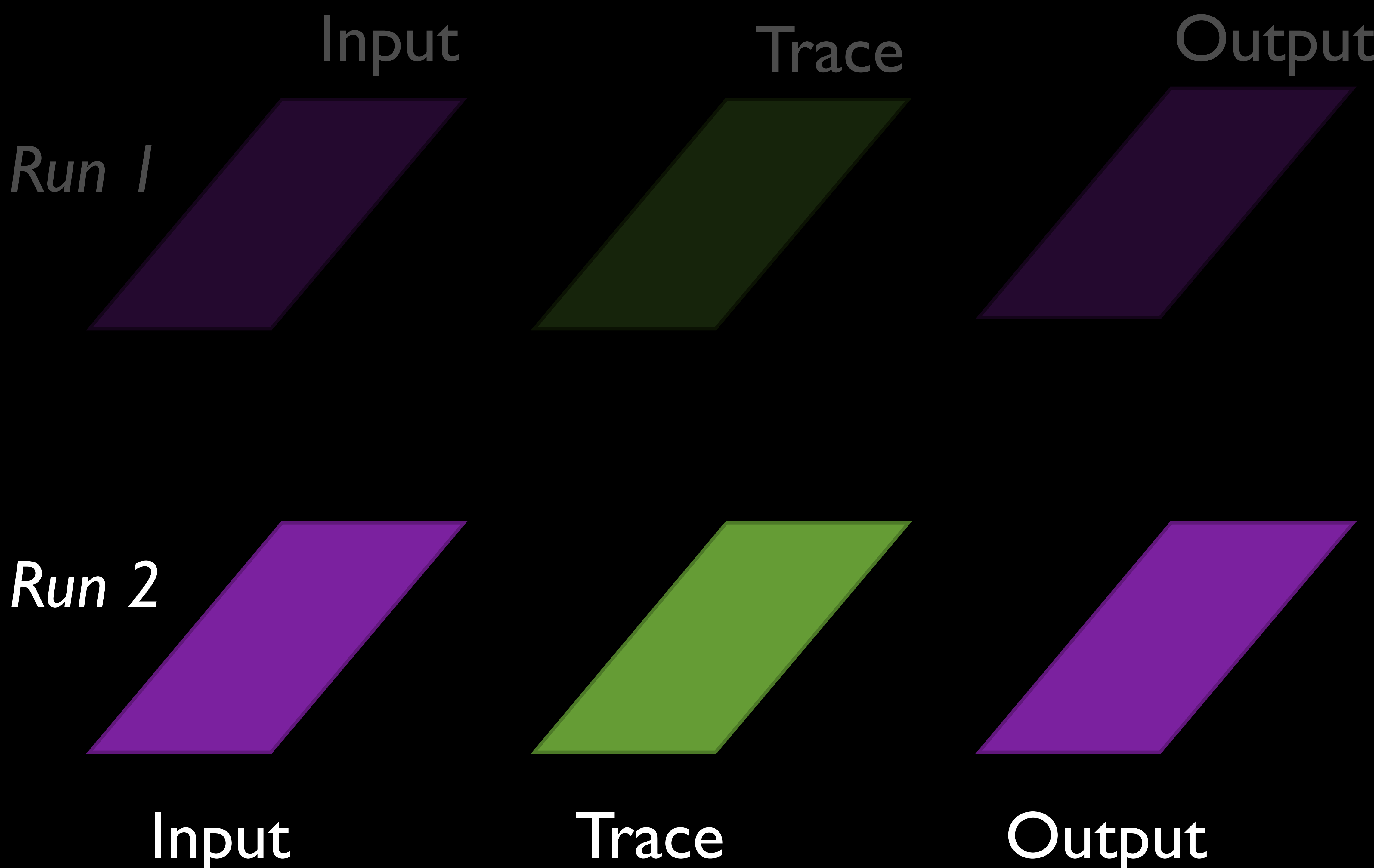
Incremental Computation



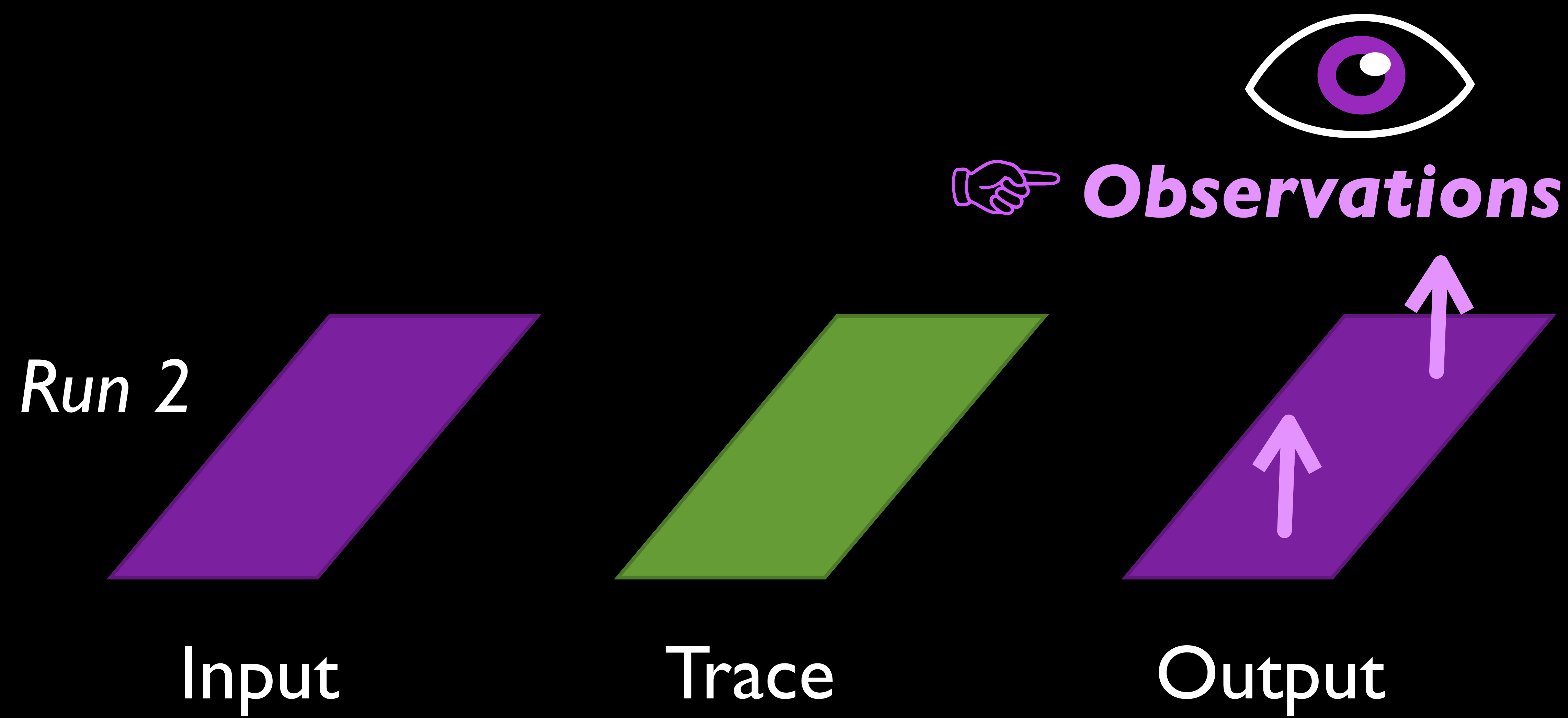
Incremental Computation



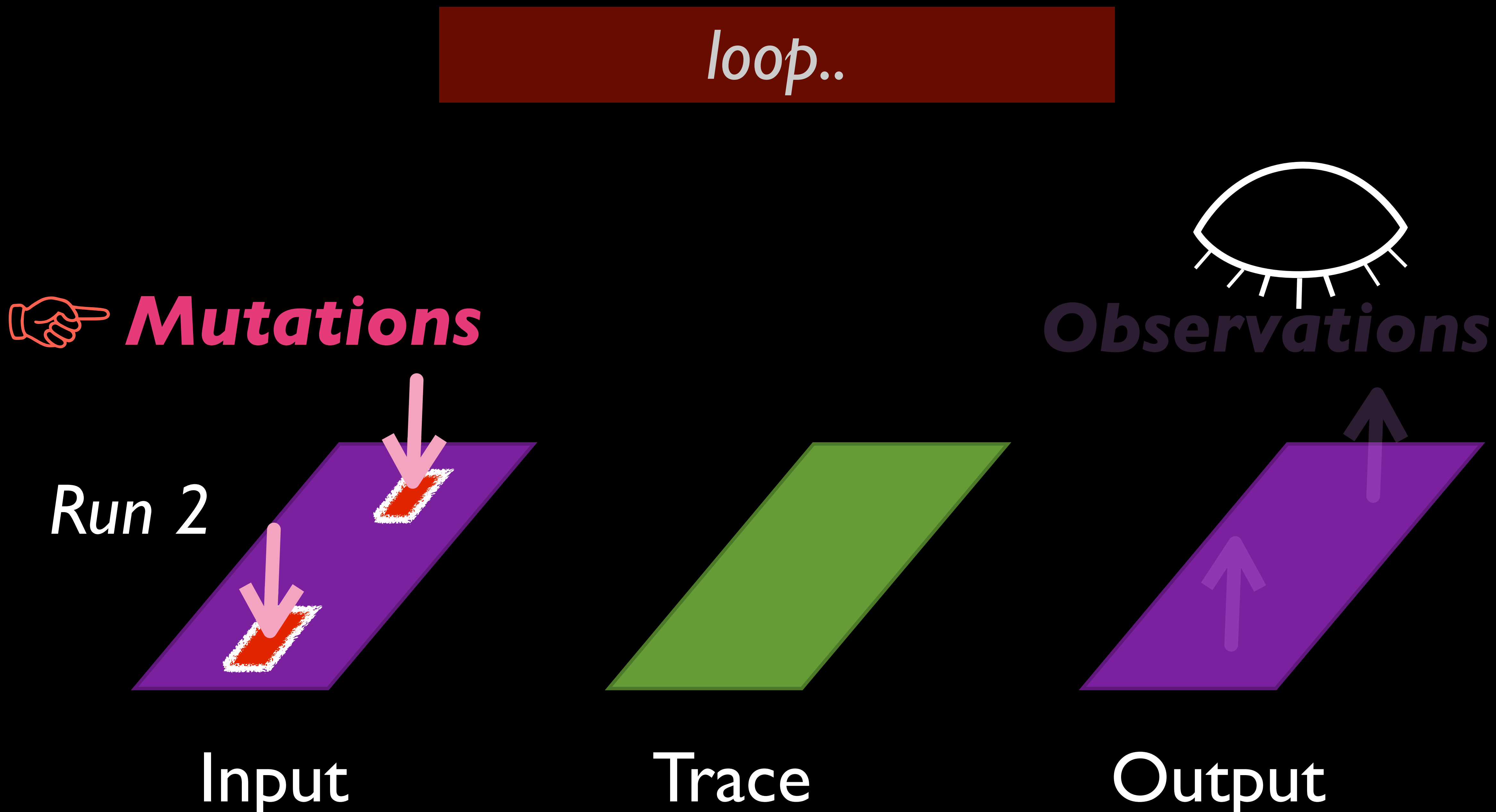
Incremental Computation



Incremental Computation



Incremental Computation



Incremental Computation

Propagation respects program semantics:

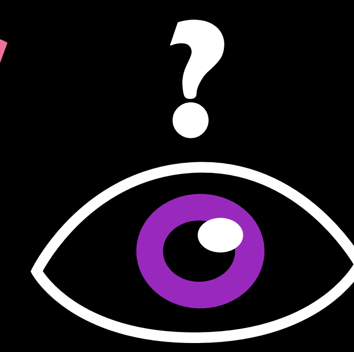
Theorem:
Trace and output are ***"from-scratch"-consistent***

Equivalently:
Change propagation is ***History independent***

Existing Limitations
(self-adjusting computation)

- **Change propagation is eager**

  Not driven by output observations



- **Trace representation = Total ordering**

  Limits reuse, excluding certain patterns

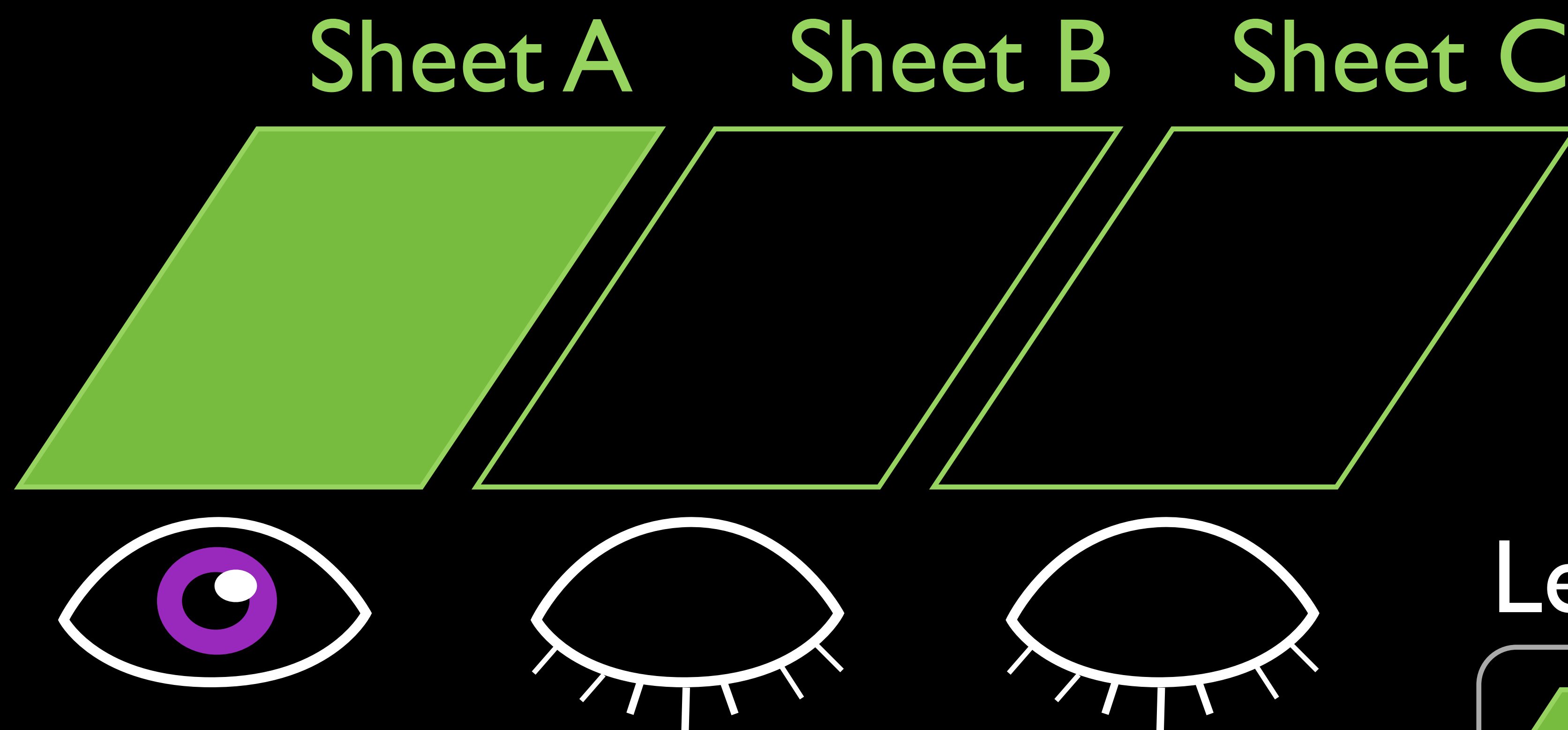
Interactive settings suffer in particular

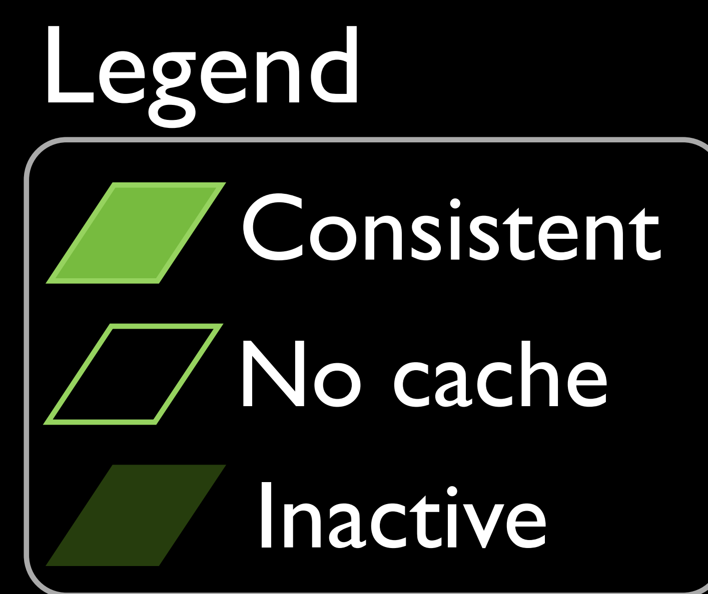
Adapton: Composable, Demand-Driven IC

- Key concepts:
  - **Lazy thunks:** programming interface
  - ***Demanded Computation Graph*** (DCG)**:** represents execution trace
- Formal semantics, proven sound
- Implemented in OCaml (and Python)
- Speedups for all patterns (unlike SAC)
- Freely available at http://ter.ps/adapton

Interaction Pattern: **Laziness**

Do not (re)compute obscured sheets

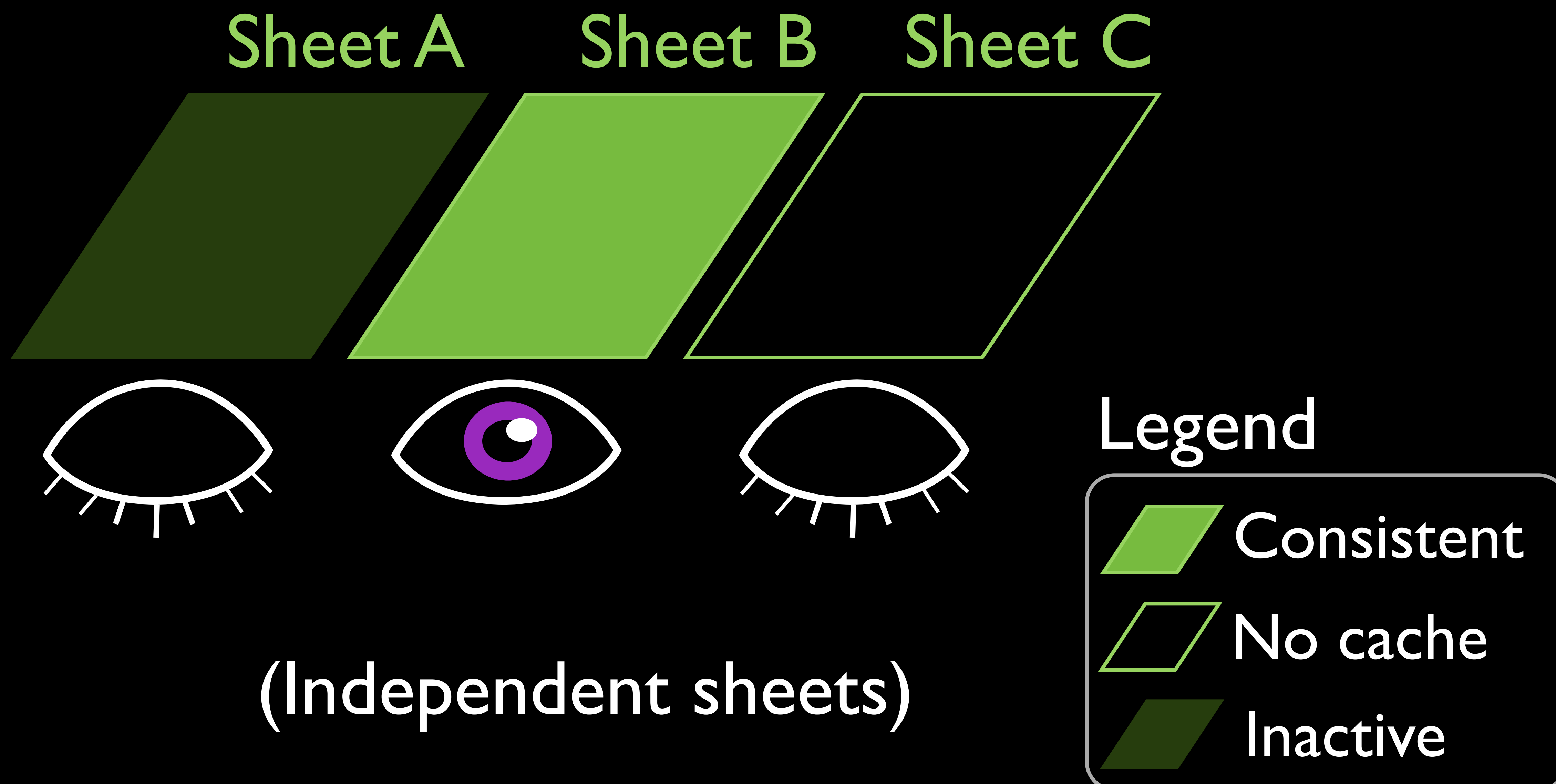




(Independent sheets)

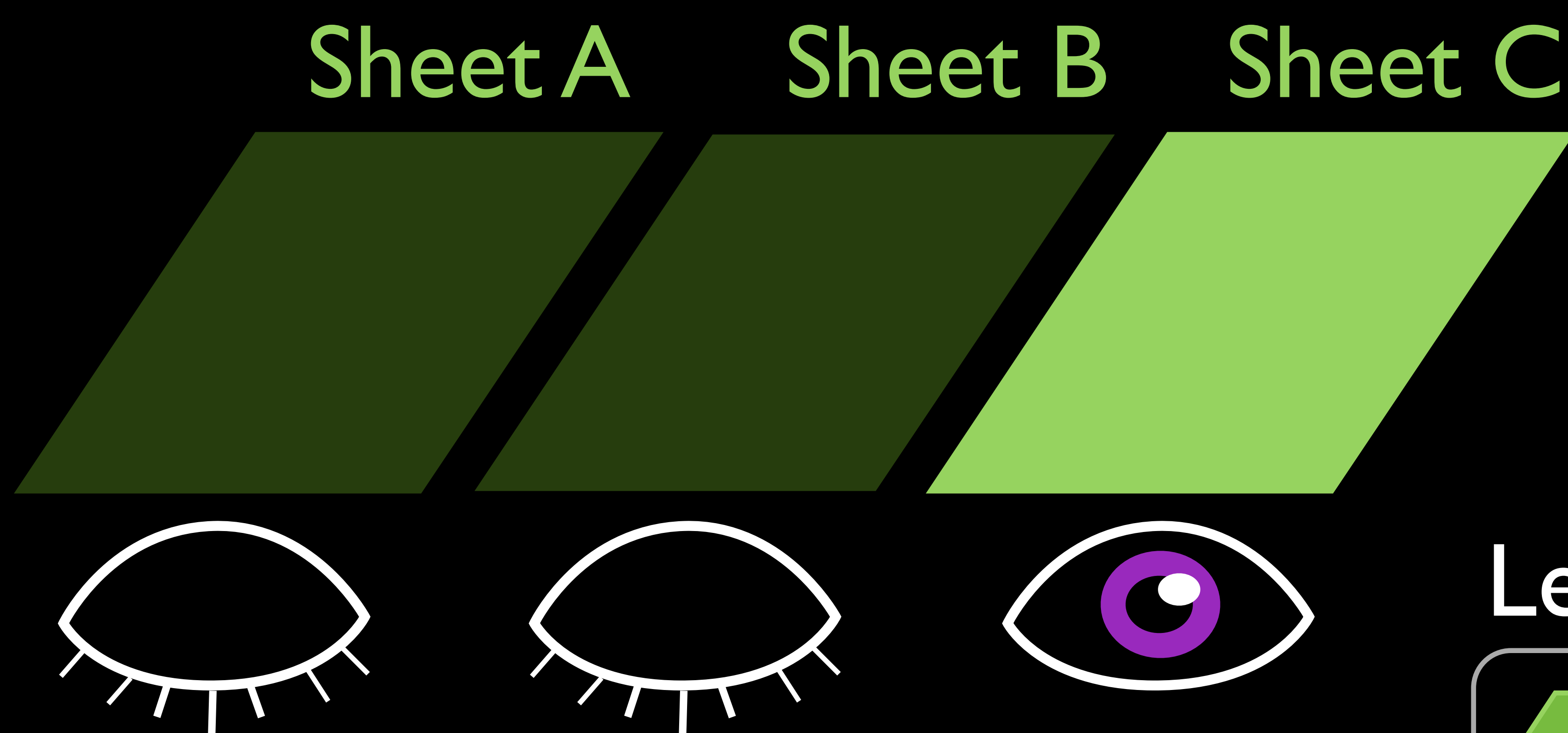
Interaction Pattern: **Laziness**

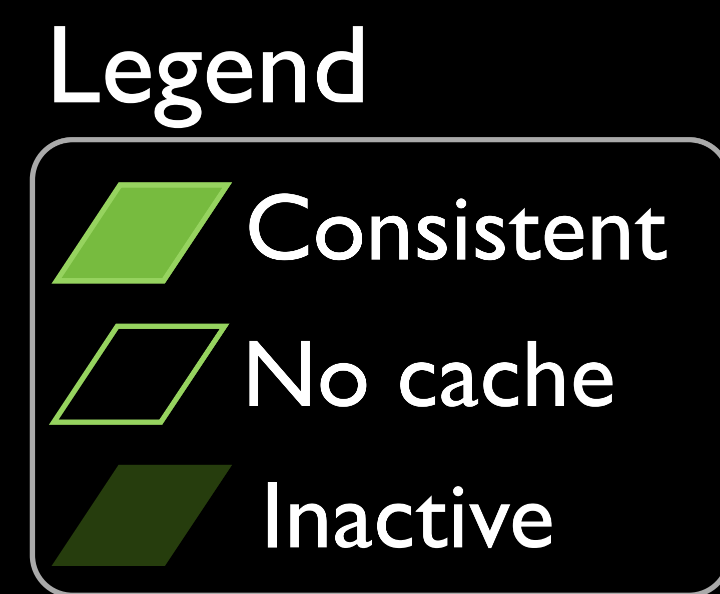
Do not (re)compute obscured sheets



Interaction Pattern: **Laziness**

Do not (re)compute obscured sheets

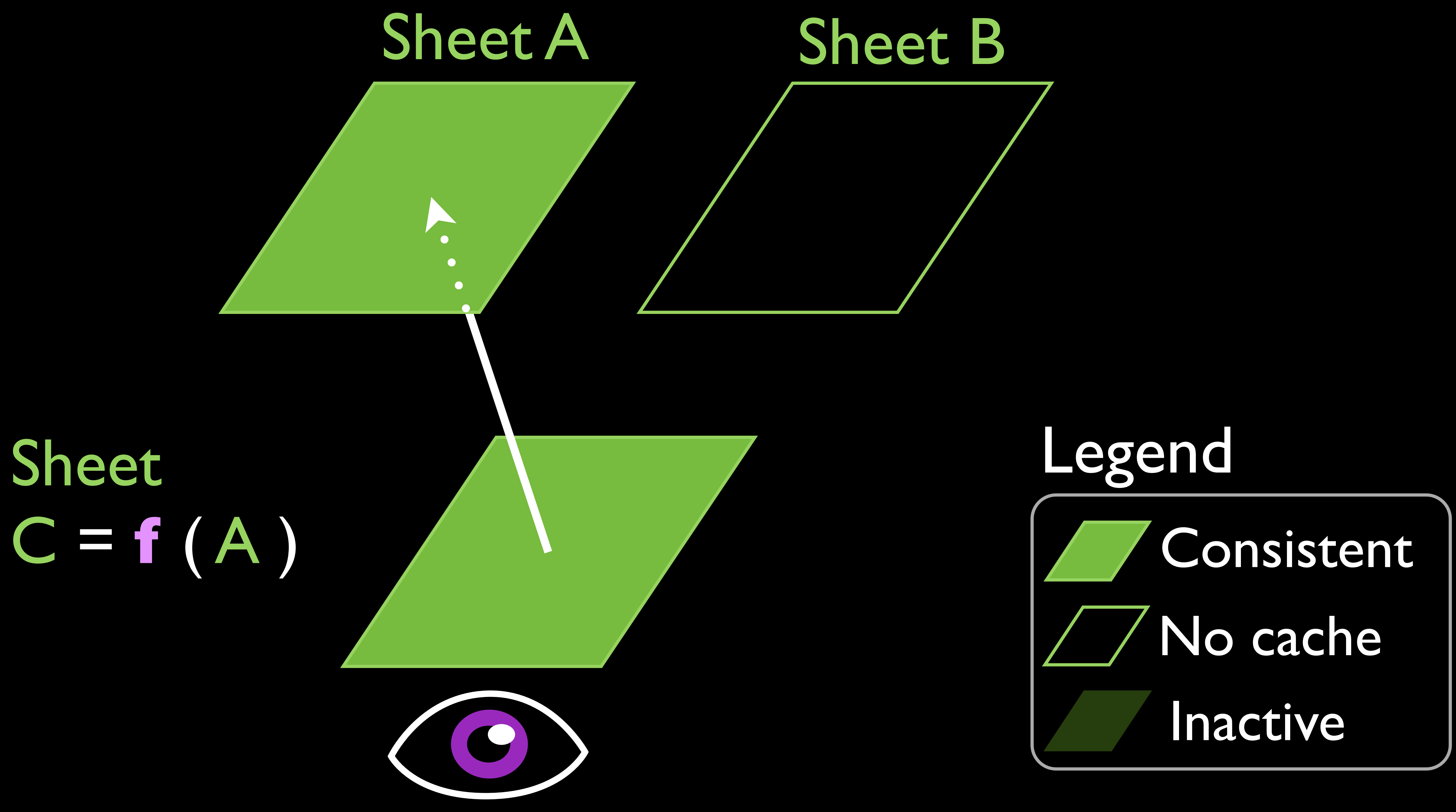




(Independent sheets)

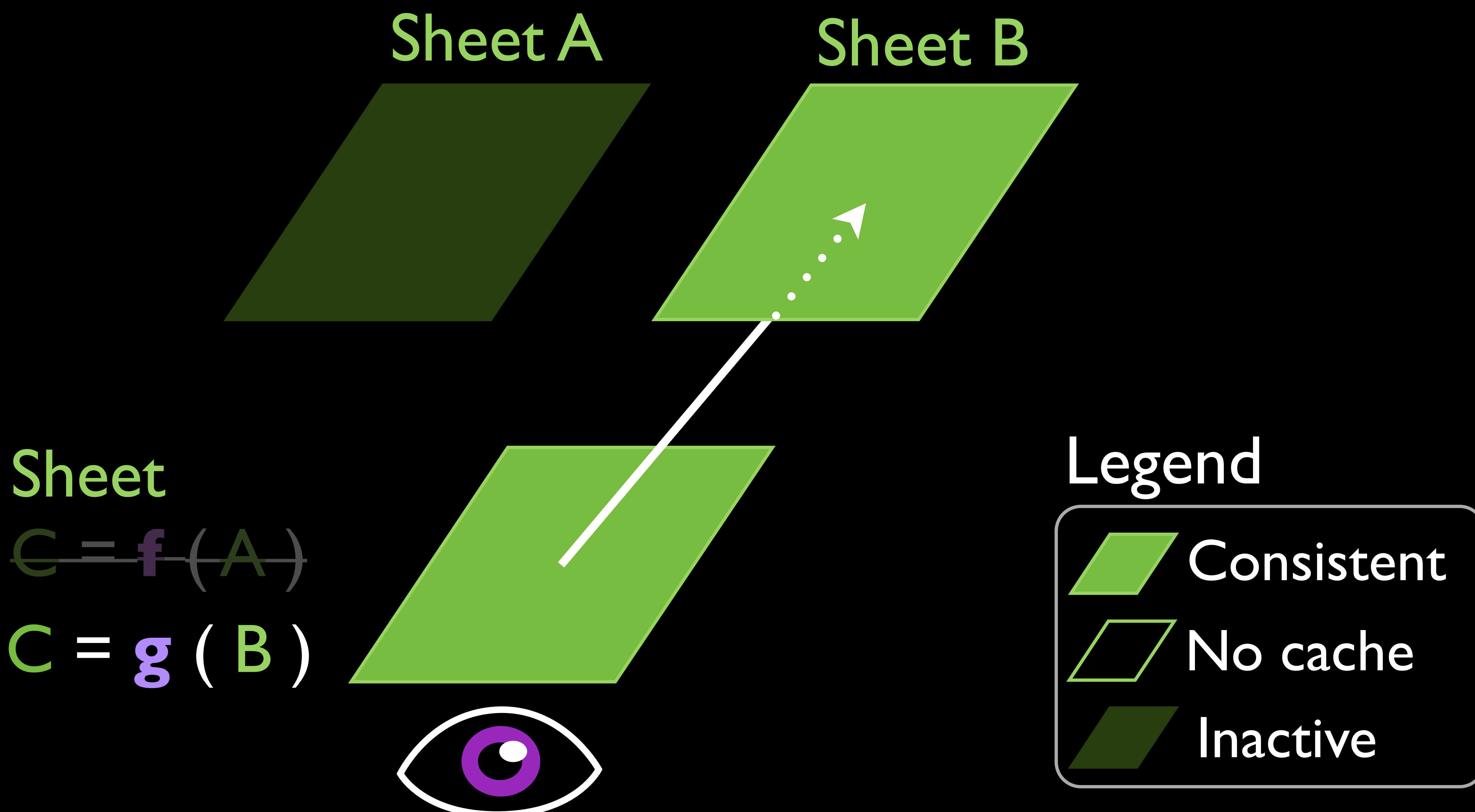
Interactive Pattern: **Switching**

Demand / control-flow change



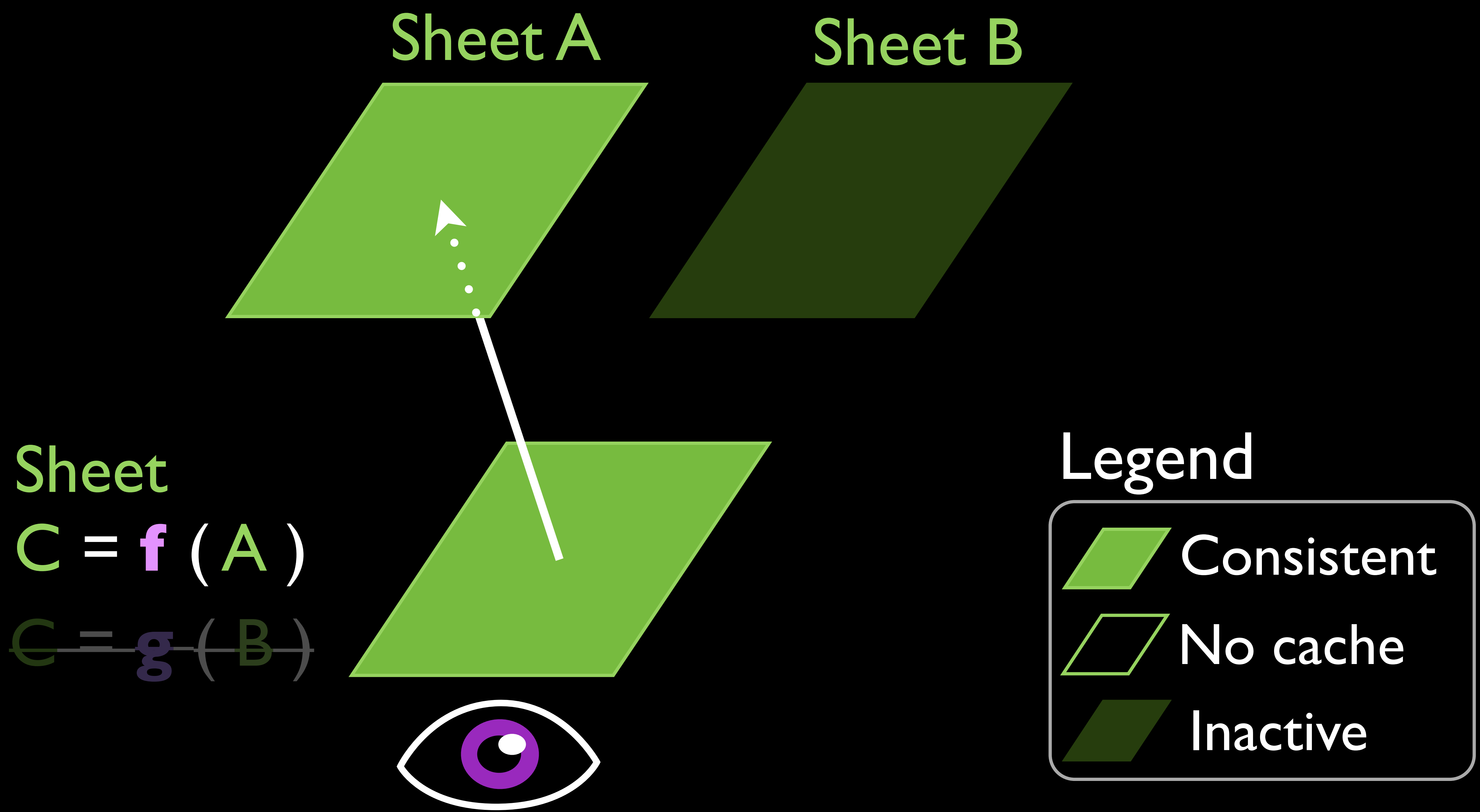
Interactive Pattern: **Switching**

Demand / control-flow change



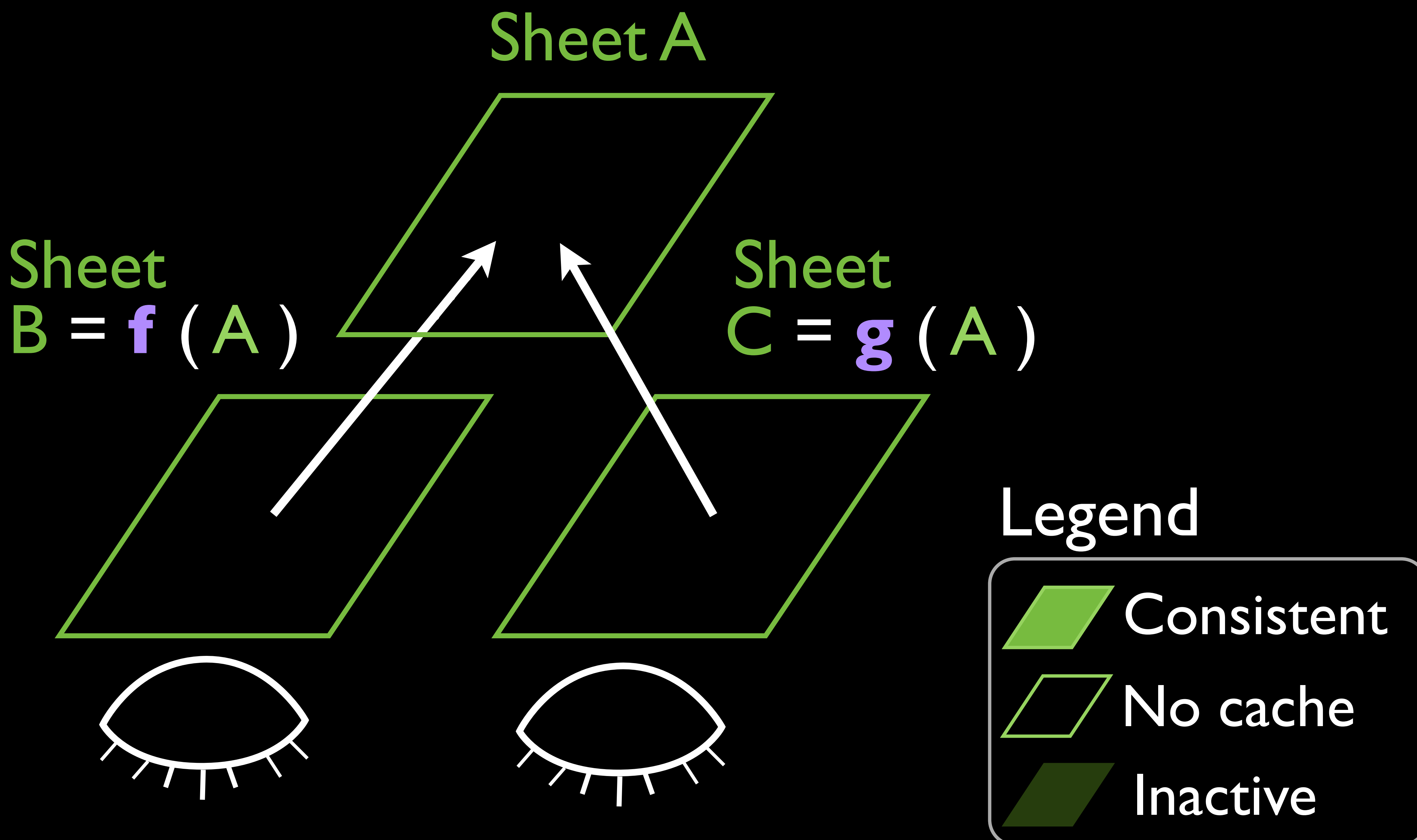
Interactive Pattern: **Switching**

Demand / control-flow change



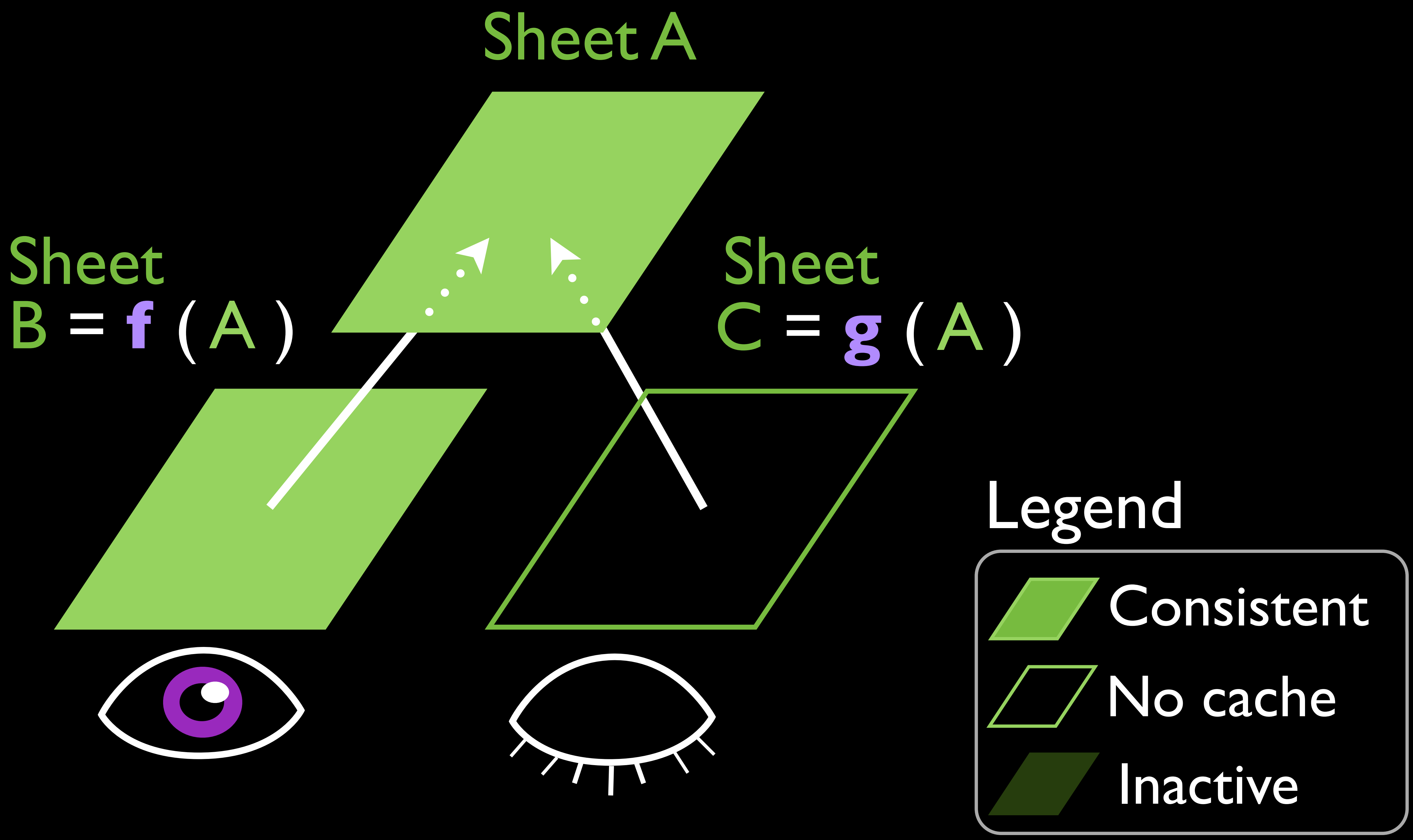
Interaction Pattern: **Sharing**

B and C share work for A



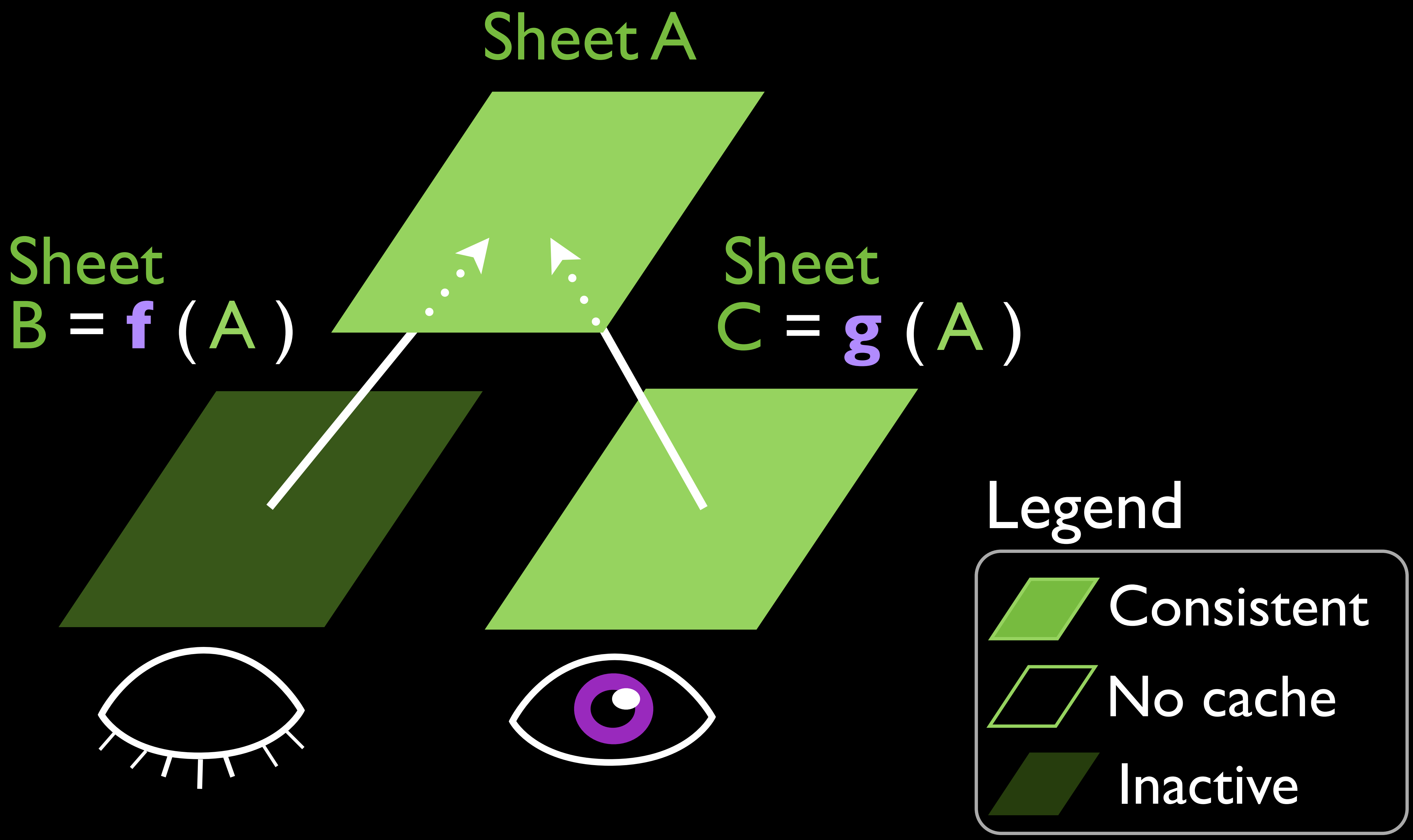
Interaction Pattern: **Sharing**

B and C share work for A



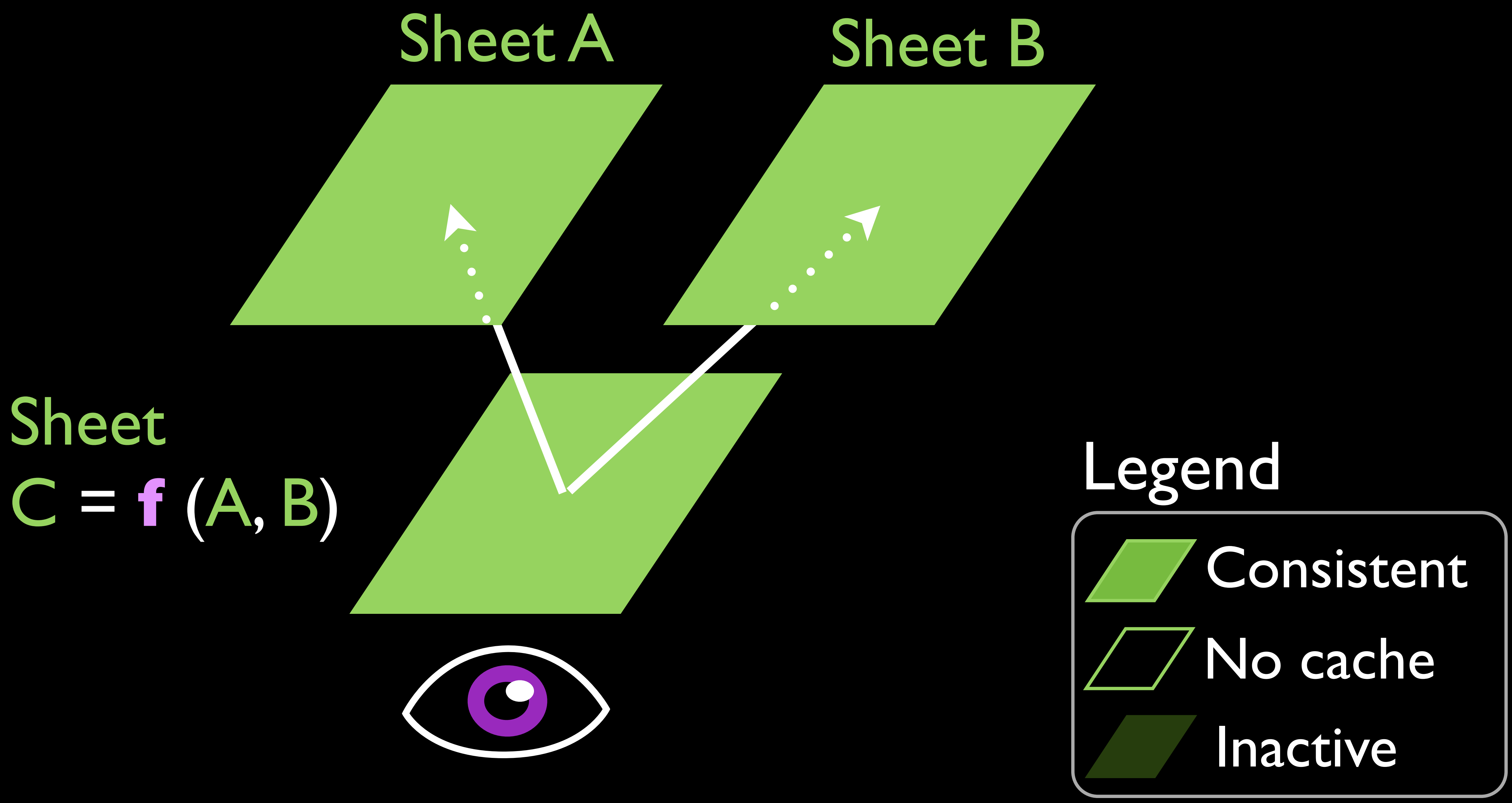
Interaction Pattern: **Sharing**

B and C share work for A



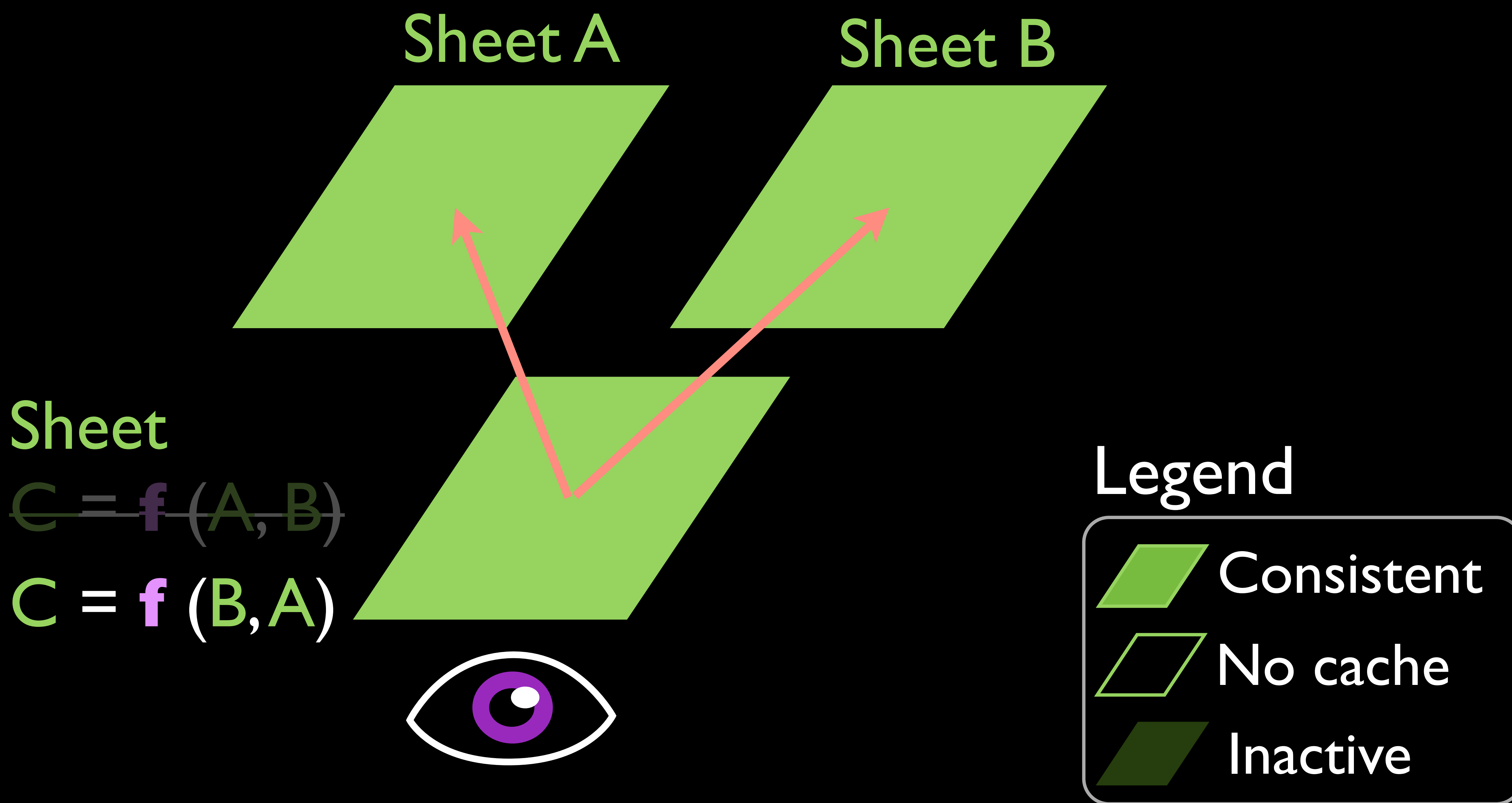
Interactive Pattern: **Swapping**

Swaps input / evaluation order



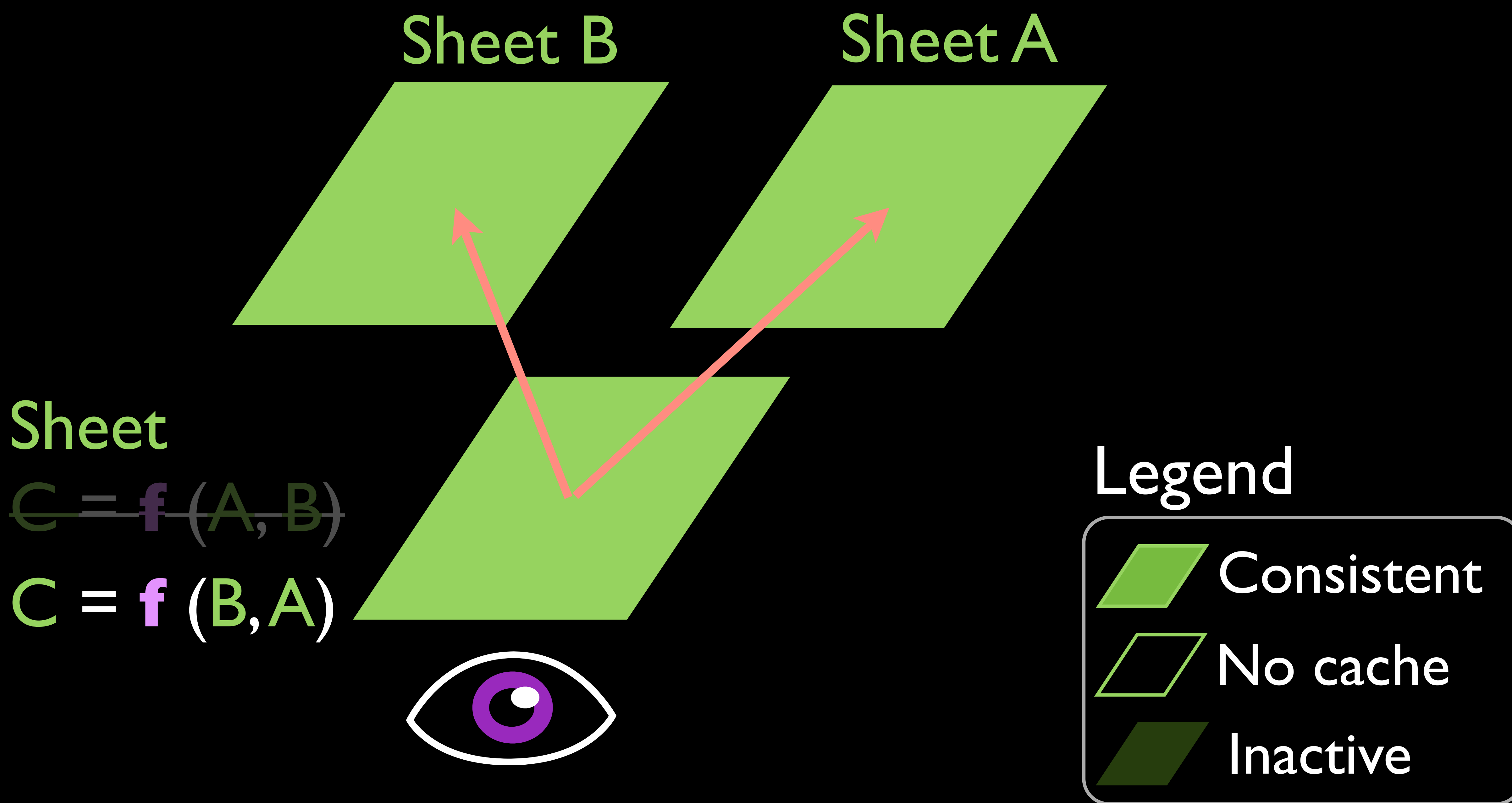
Interactive Pattern: **Swapping**

Swaps input / evaluation order



Interactive Pattern: **Swapping**

Swaps input / evaluation order



Adapton's Approach

- When we **mutate** an **input**, we mark dependent computations as **dirty**
- When we **demand a thunk**:
  - **Memo-match** equivalent thunks
  - **Change-propagation** repairs inconsistencies, **on demand**

Spread Sheet Evaluator

```
type cell = formula ref

and formula =
      | Leaf of int
      | Plus of cell * cell
```

Spread Sheet Evaluator

Mutable

```
type cell = formula ref

and formula =
     | Leaf of int
     | Plus of cell * cell
```

Depends on cells

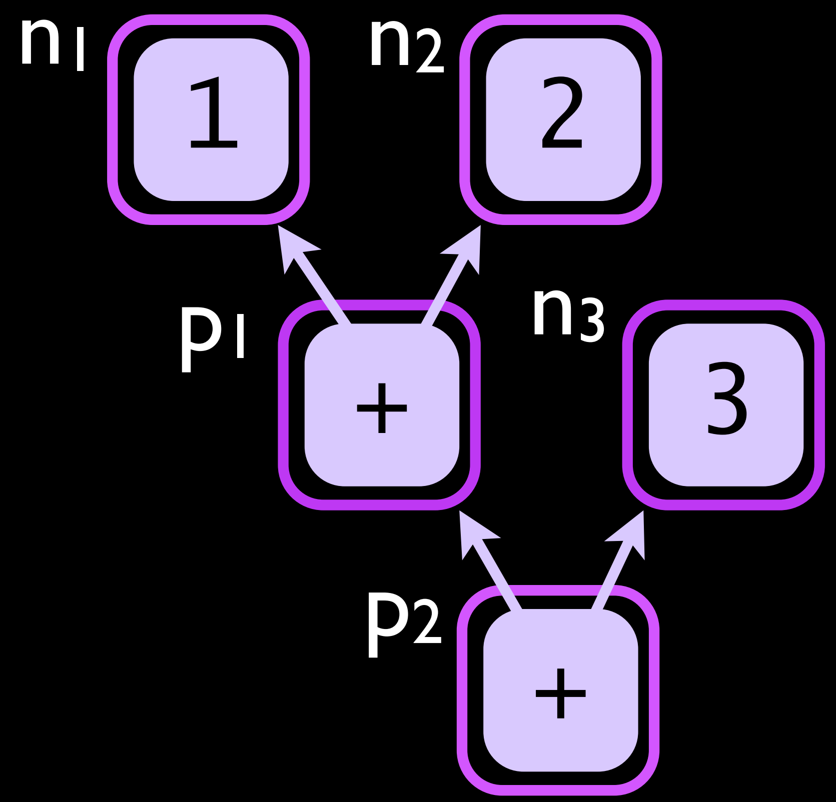
Spread Sheet Evaluator

Example

```
let n1 = ref (Leaf 1)

let n2 = ref (Leaf 2)

let n3 = ref (Leaf 3)

let p1 = ref (Plus (n1, n2))

let p2 = ref (Plus (p1, n3))
```

```
type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
```

Spread Sheet Evaluator



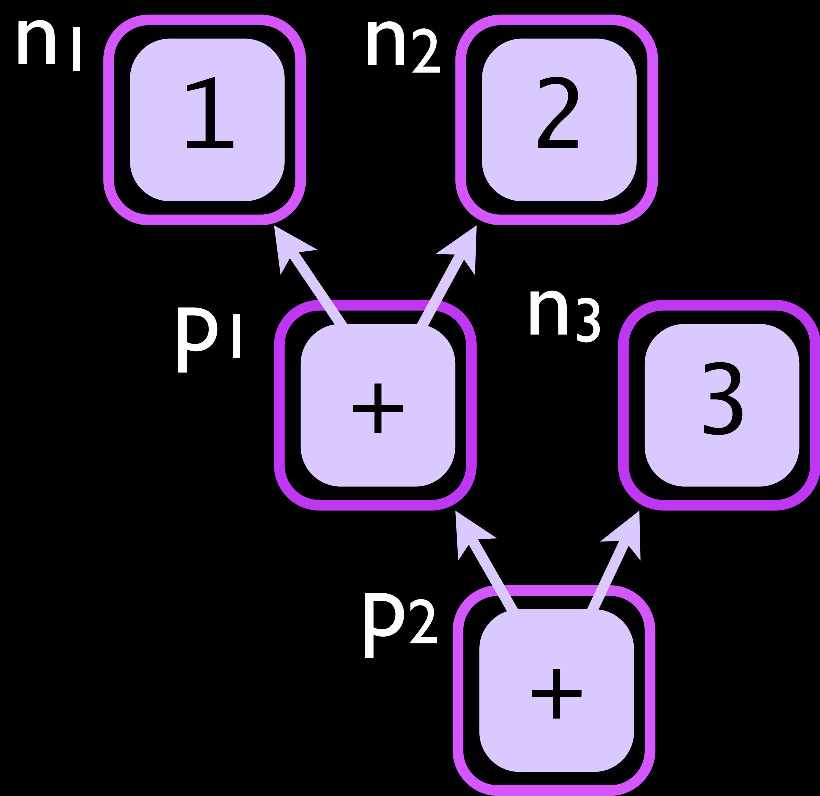
Example

```
let n1 = ref (Leaf 1)
let n2 = ref (Leaf 2)
let n3 = ref (Leaf 3)
let p1 = ref (Plus (n1, n2))
let p2 = ref (Plus (p1, n3))
```

"User interface" (REPL)

```
type cell = formula ref
and formula =
  | Leaf of int
  | Plus of cell * cell
```

Spread Sheet Evaluator

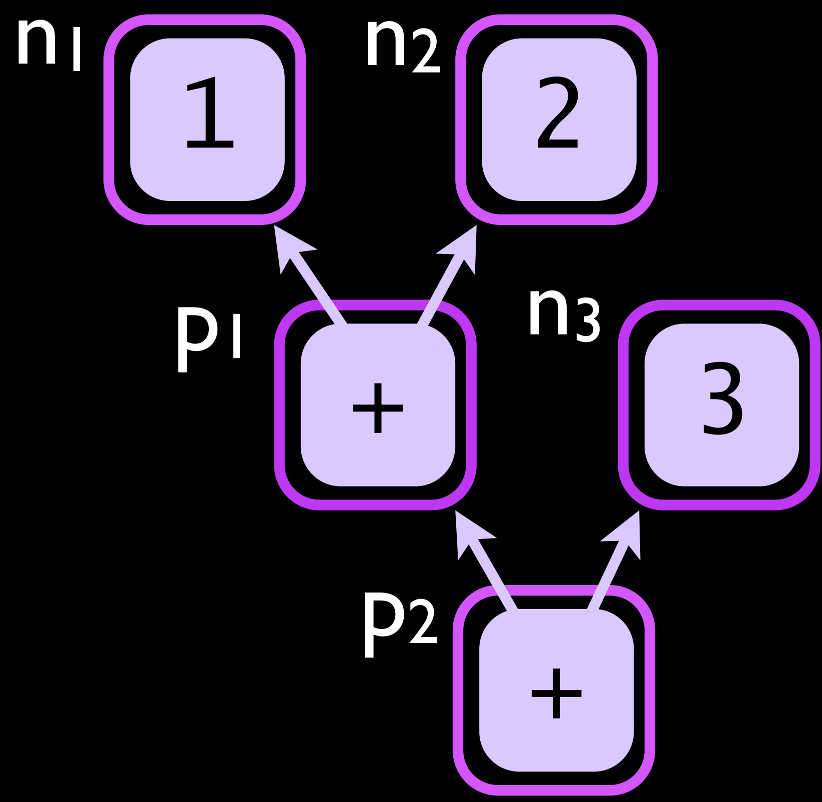



```
type cell = formula ref
and formula =
  | Leaf of int
  | Plus of cell * cell
```

Evaluator logic

```
eval : cell → (int thunk)

eval c = thunk ((
   case (get c) of
    | Leaf n  ⇒  n
    | Plus(c1, c2) ⇒
        force (eval c1) +
        force (eval c2)
  ))
```

Spread Sheet Evaluator

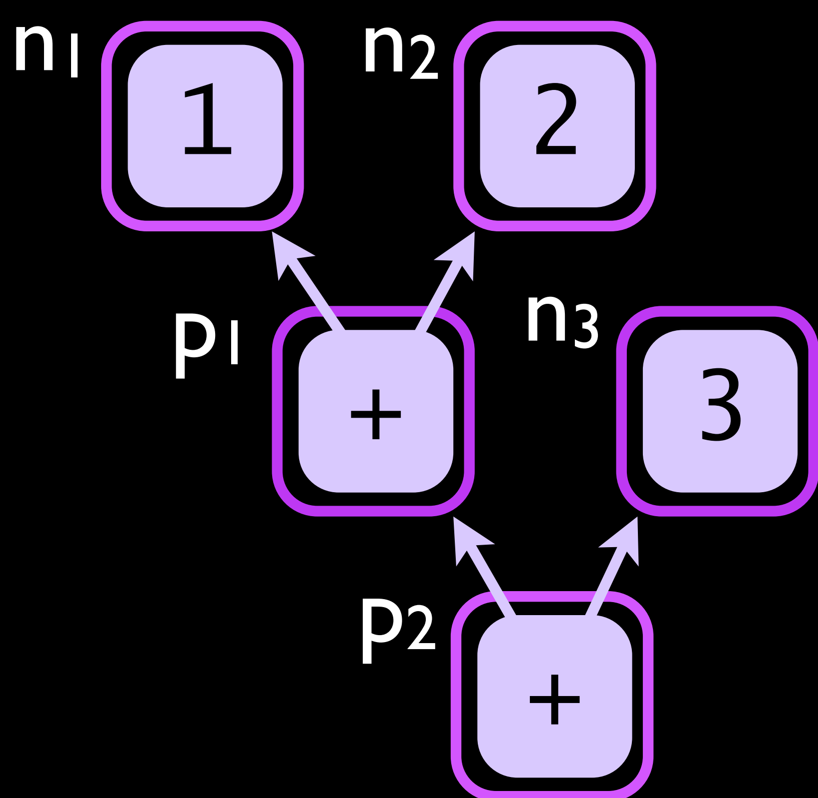



```
set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit
```

"User interface" (REPL)

```
type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
```

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

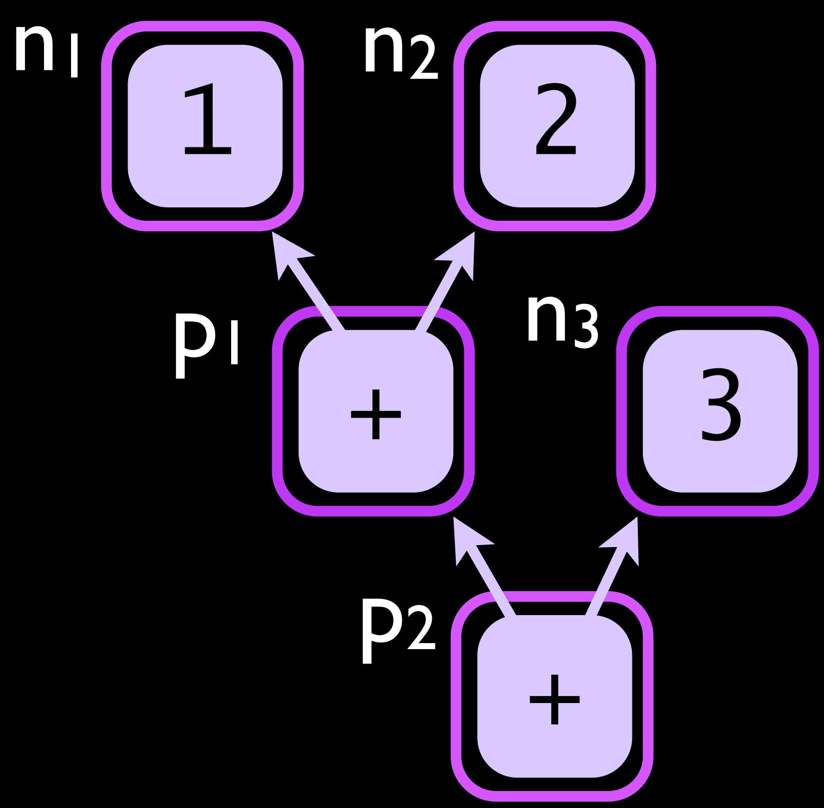
display : (int thunk) → unit

“User ” (REPL)

Demands evaluation

```
type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
```

Spread Sheet Evaluator

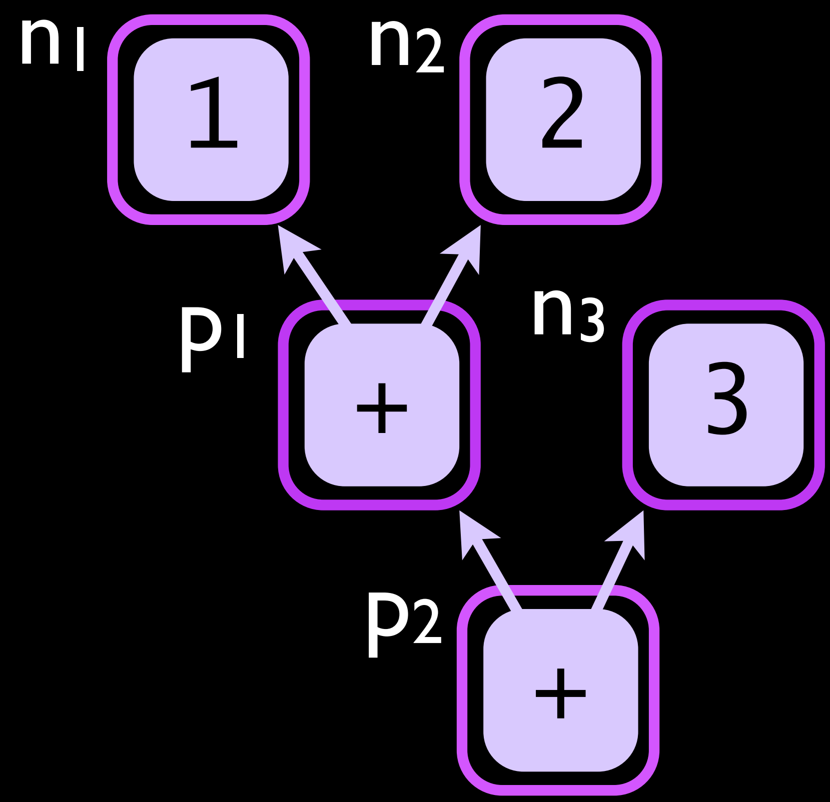



```
set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit
```

"User interface" (REPL)

```
type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
```

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

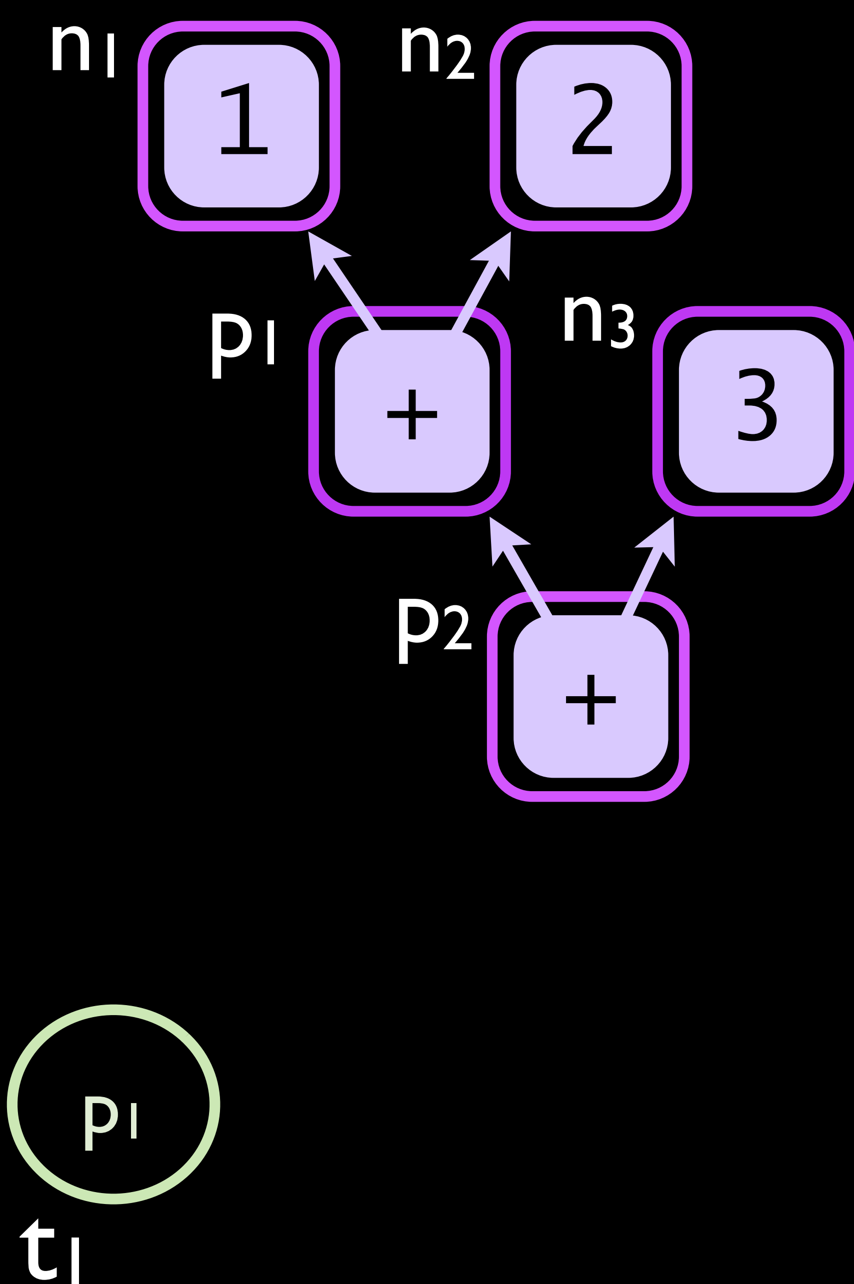
display : (int thunk) → unit

```
type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
```

“User interface” (REPL)

☞ let t_1 = eval p_1

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

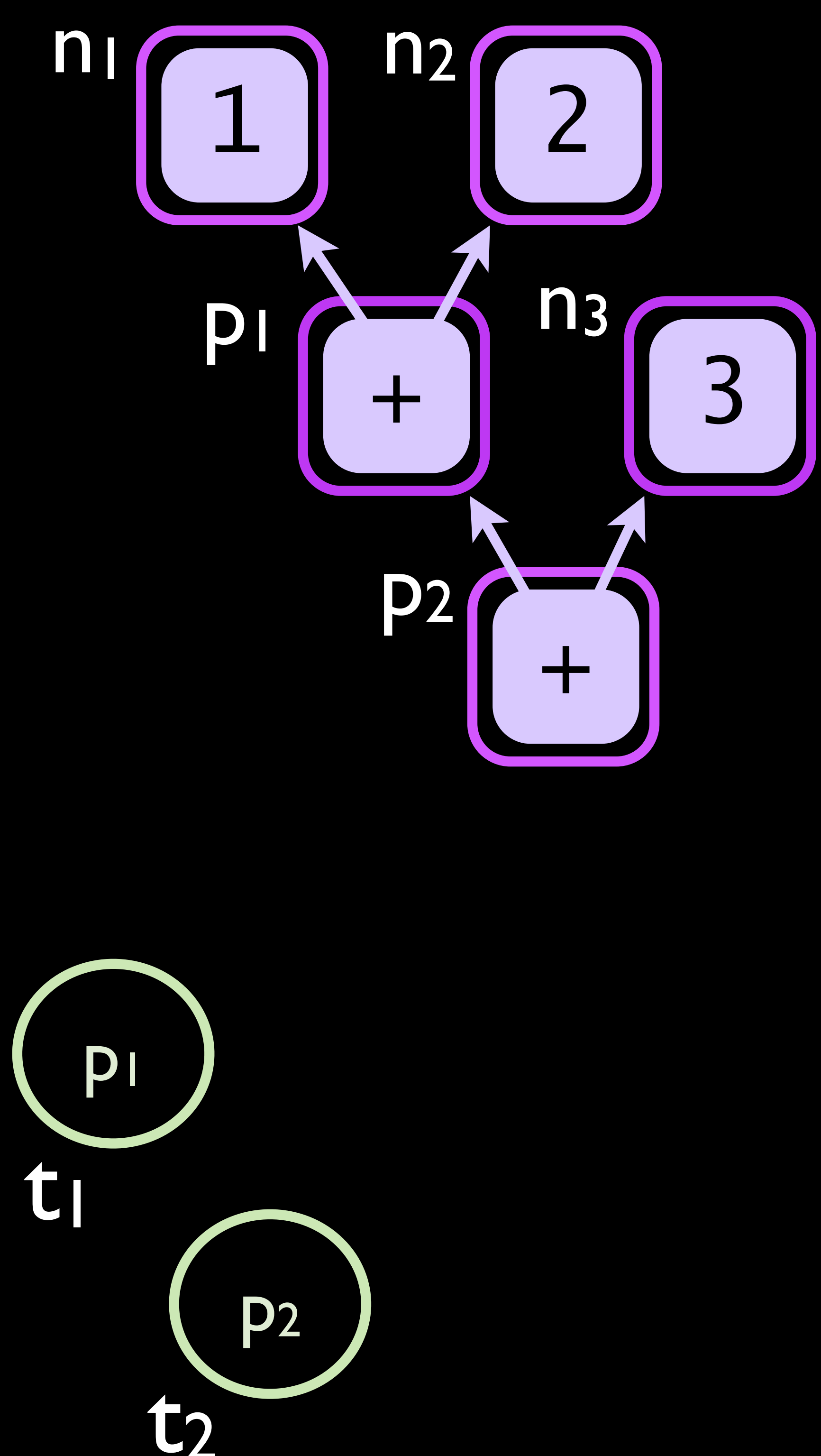
display : (int thunk) → unit

"User interface" (REPL)

☞ let t_1 = eval p_1

☞

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

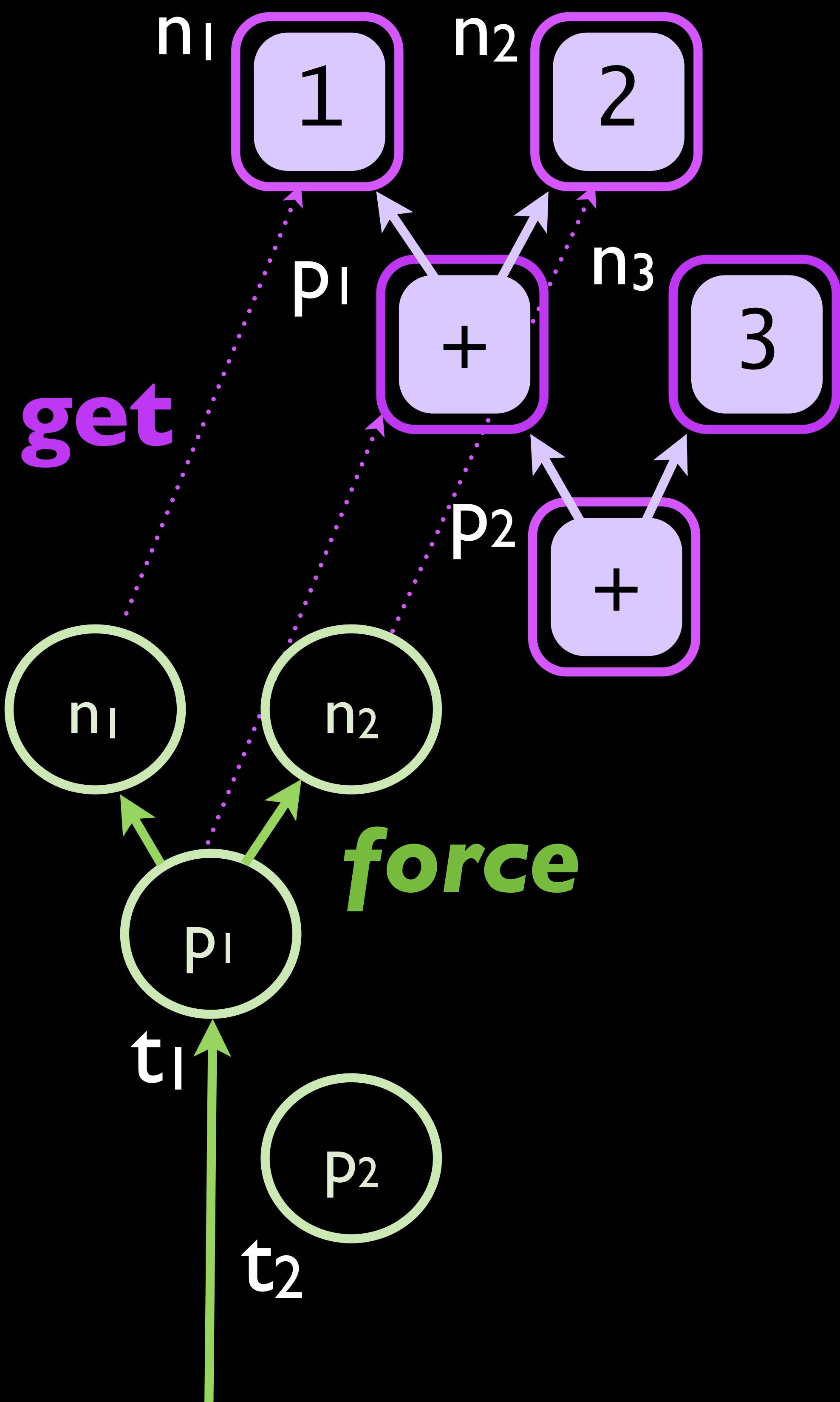
“User interface” (REPL)

☞ let t_1 = eval p_1

☞ let t_2 = eval p_2

☞

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

"User interface" (REPL)

☞ let t_1 = eval p_1

☞ let t_2 = eval p_2

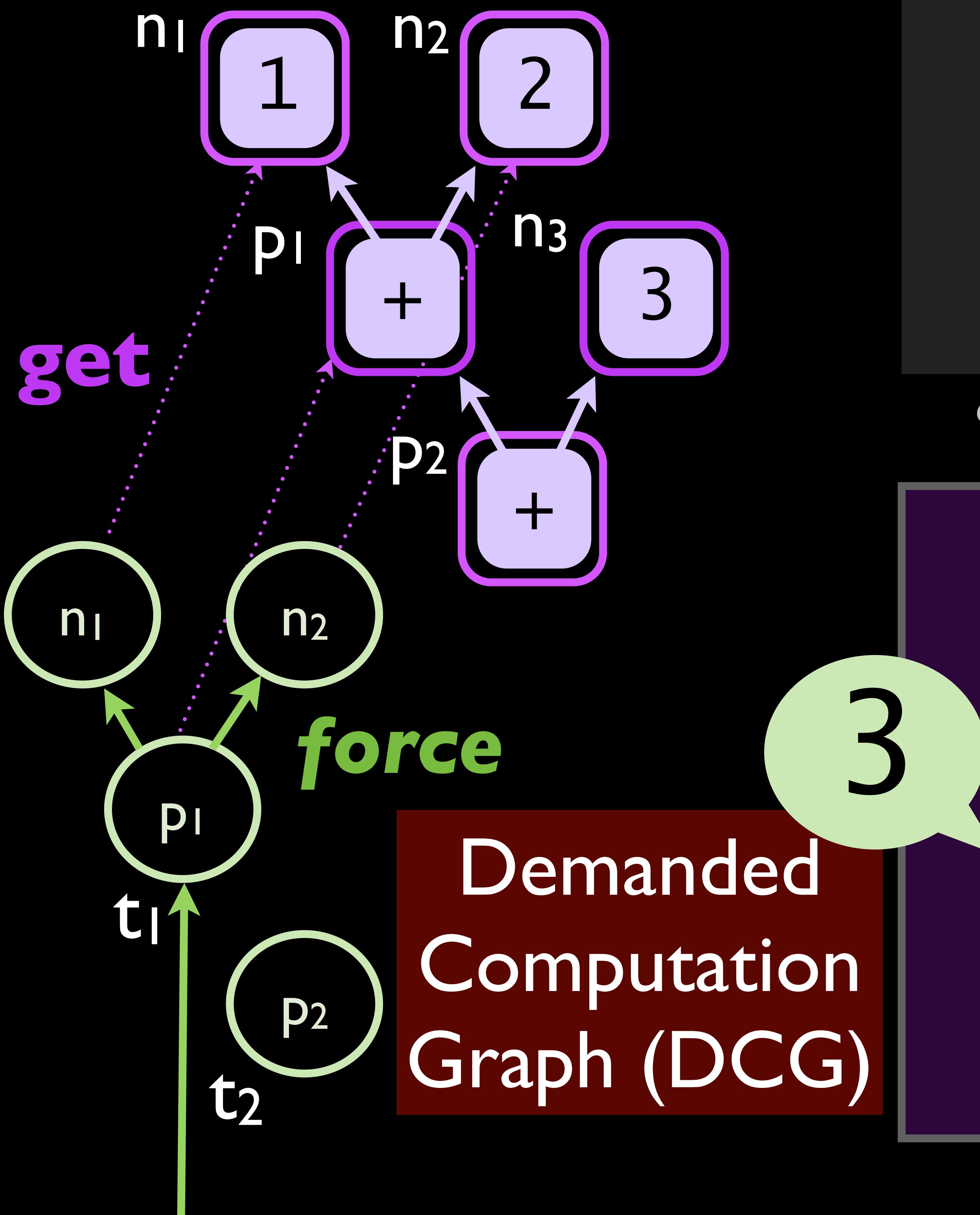
☞ display t_1

☞

3

demand!

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

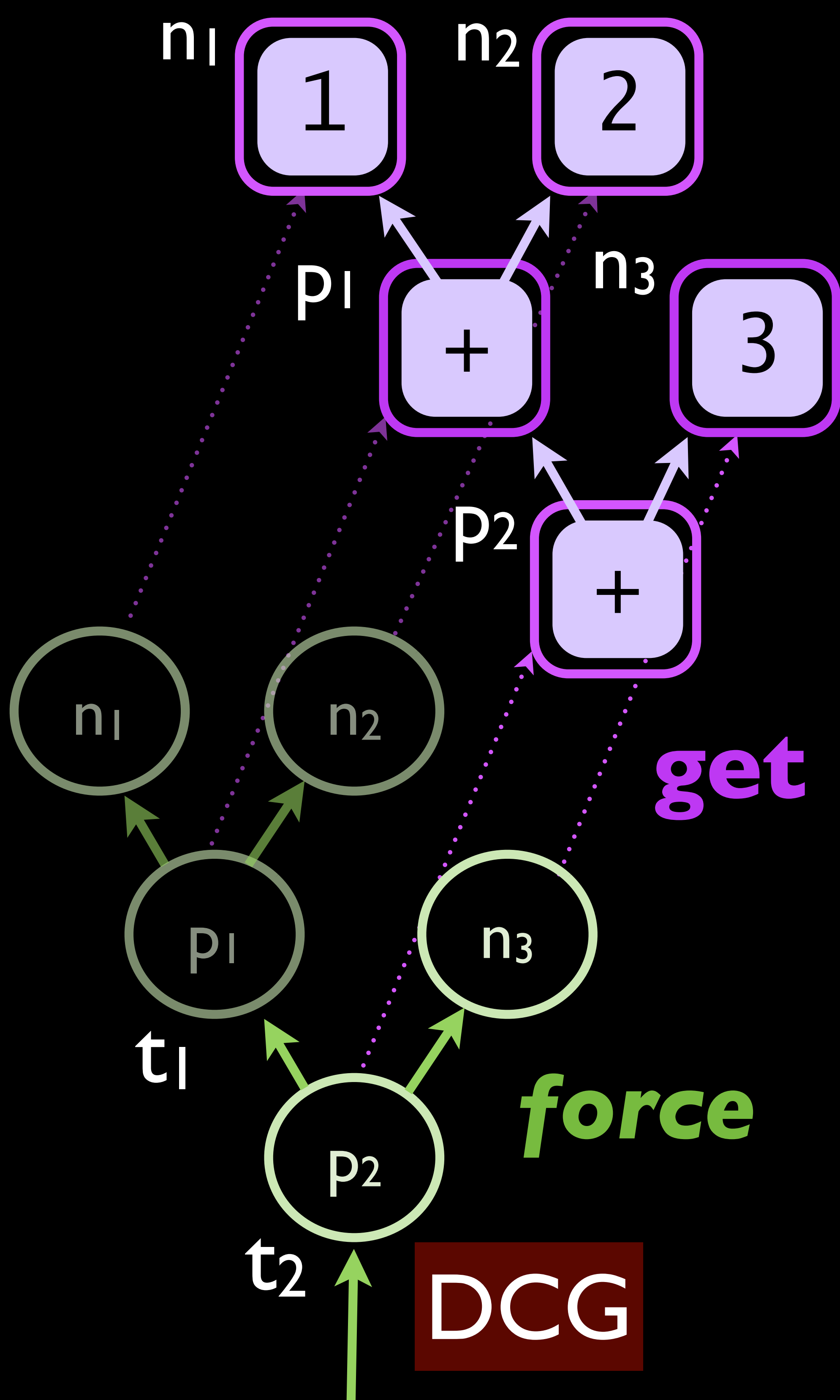
☞ let t_1 = eval p_1

☞ let t_2 = eval p_2

☞ display t_1

☞

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

☞ let t_1 = eval p_1

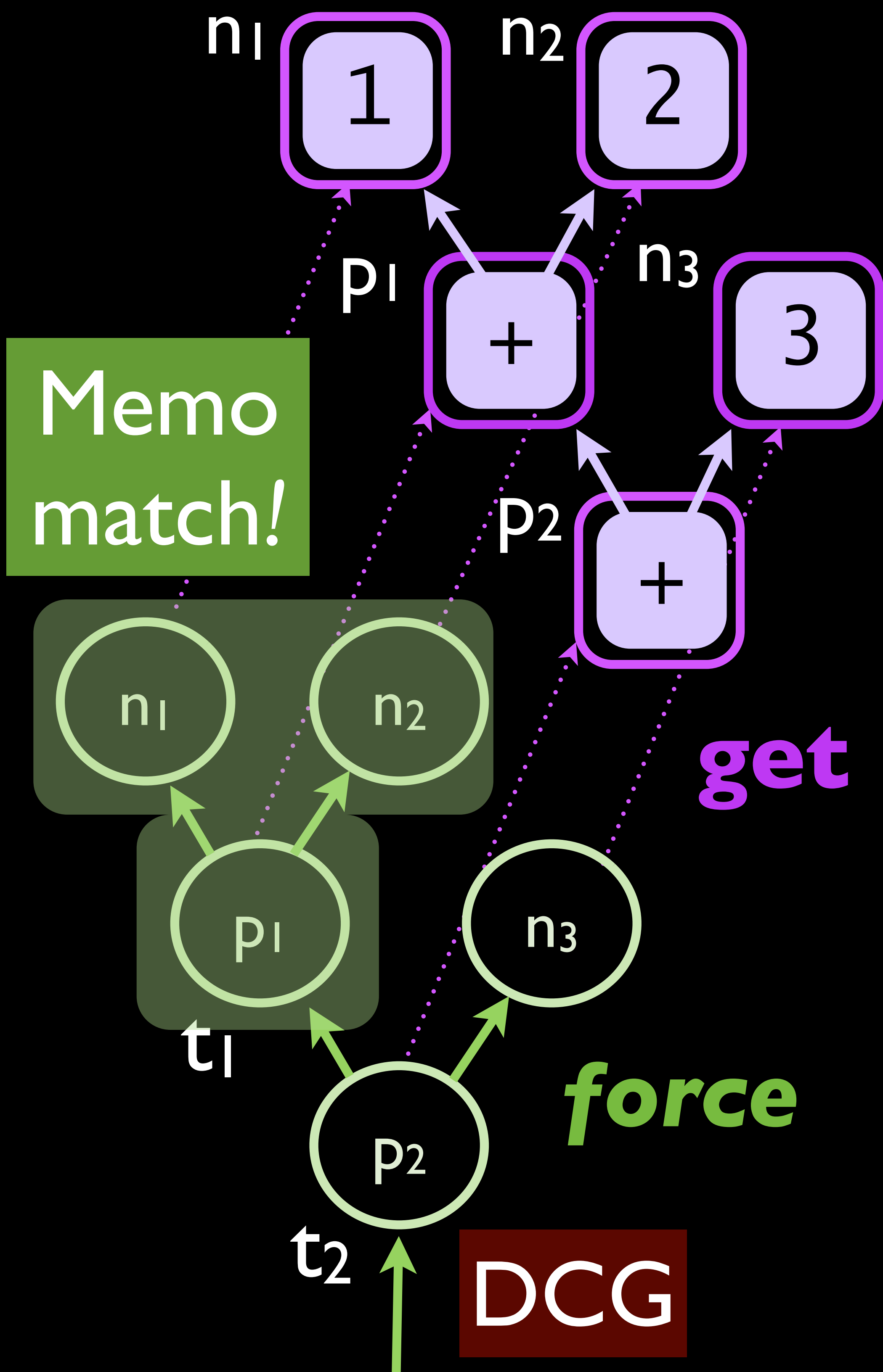
☞ let t_2 = eval p_2

☞ display t_1

☞ display t_2

☞

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

"User interface" (REPL)

☞ let t_1 = eval p_1

☞ let t_2 = eval p_2

☞ display t_1

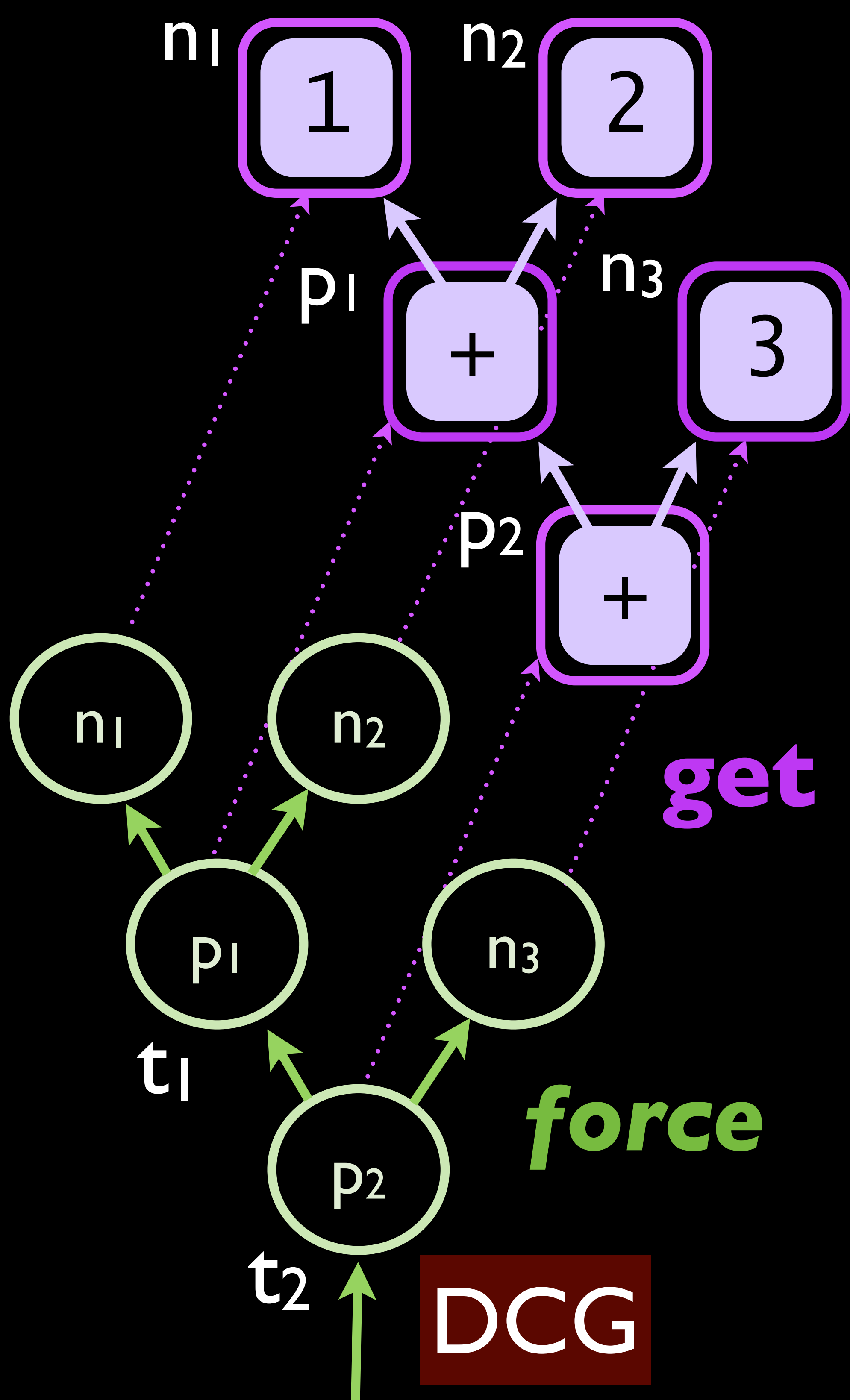
☞ display t_2

☞

6

Memo match! ***Sharing***

Spread Sheet Evaluator



set : cell x formula → unit

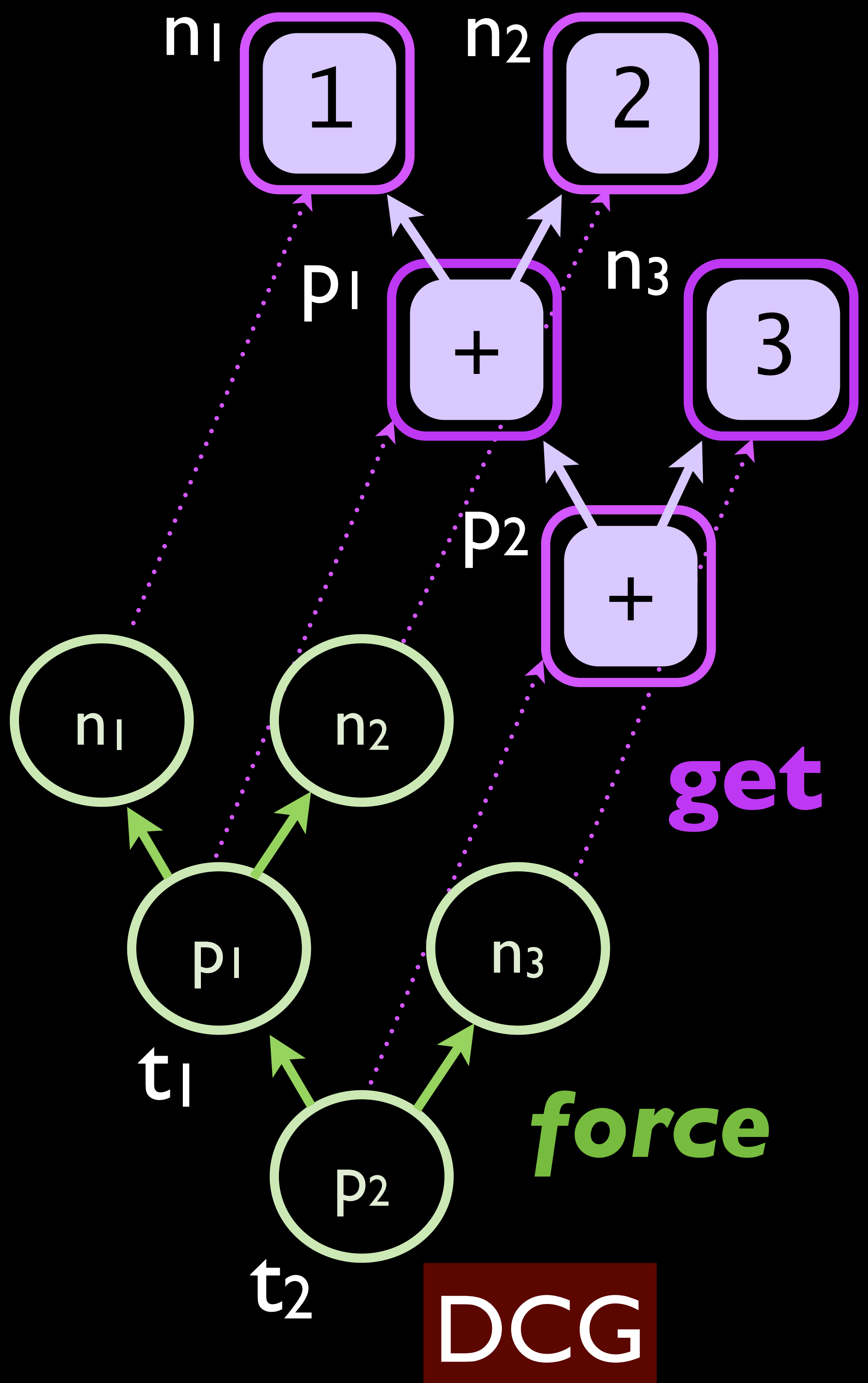
eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

- ☞ let t_1 = eval p_1
- ☞ let t_2 = eval p_2
- ☞ display t_1
- ☞ display t_2
- ☞ clear

Spread Sheet Evaluator



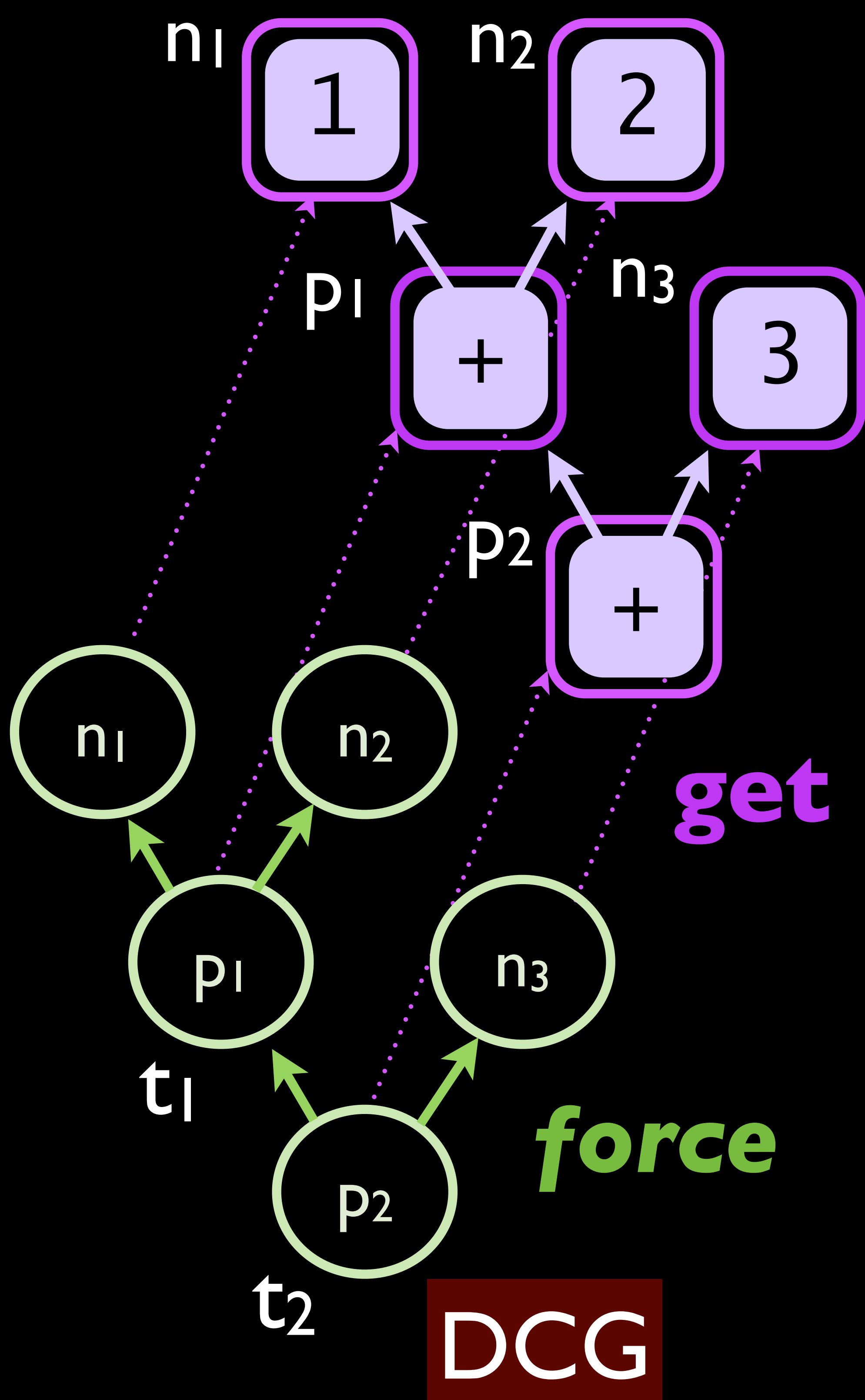
set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

Spread Sheet Evaluator



set : cell x formula → unit

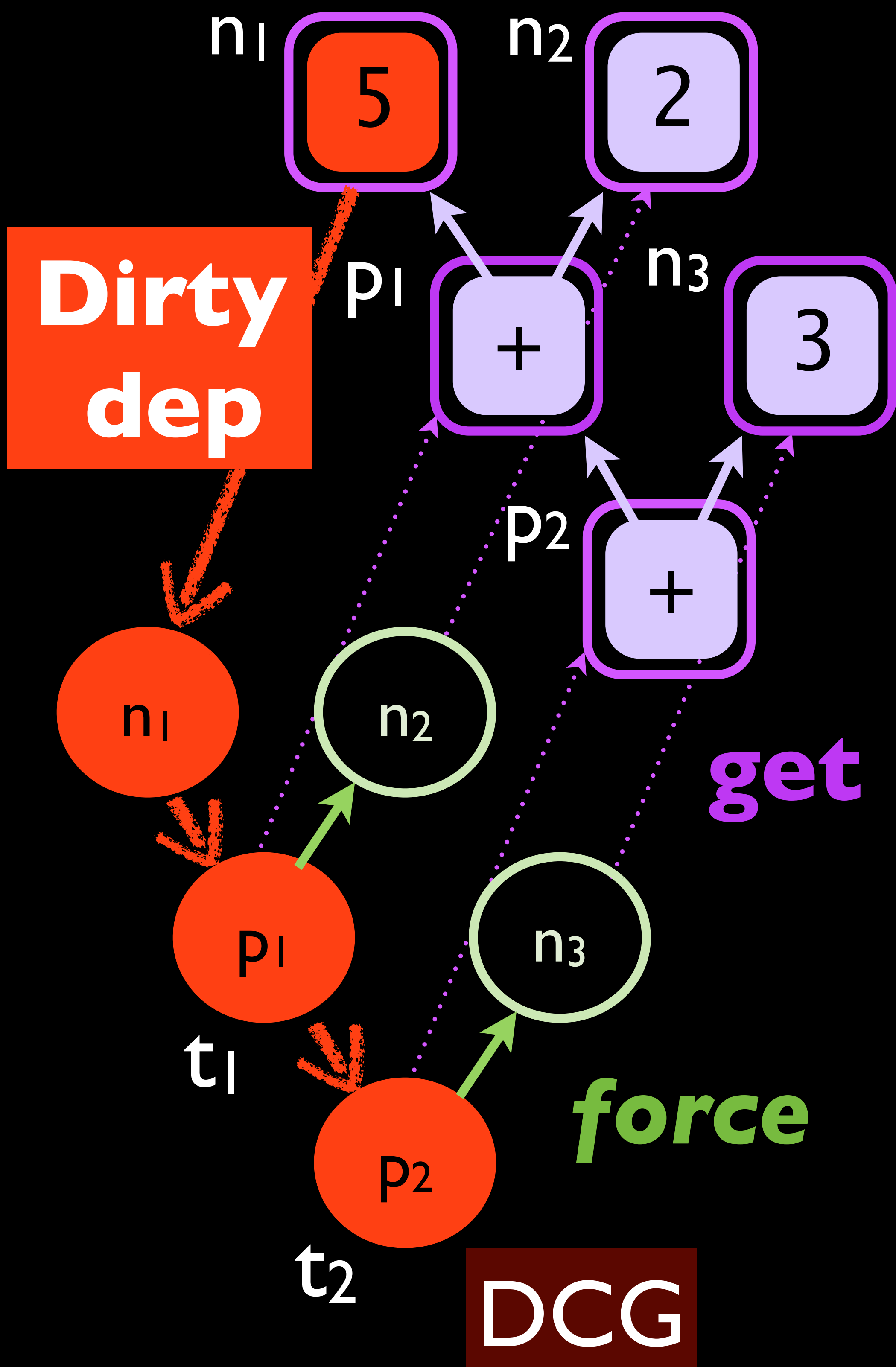
eval : cell → (int thunk)

display : (int thunk) → unit

"User interface" (REPL)

☞ set n_1 ← Leaf 5

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

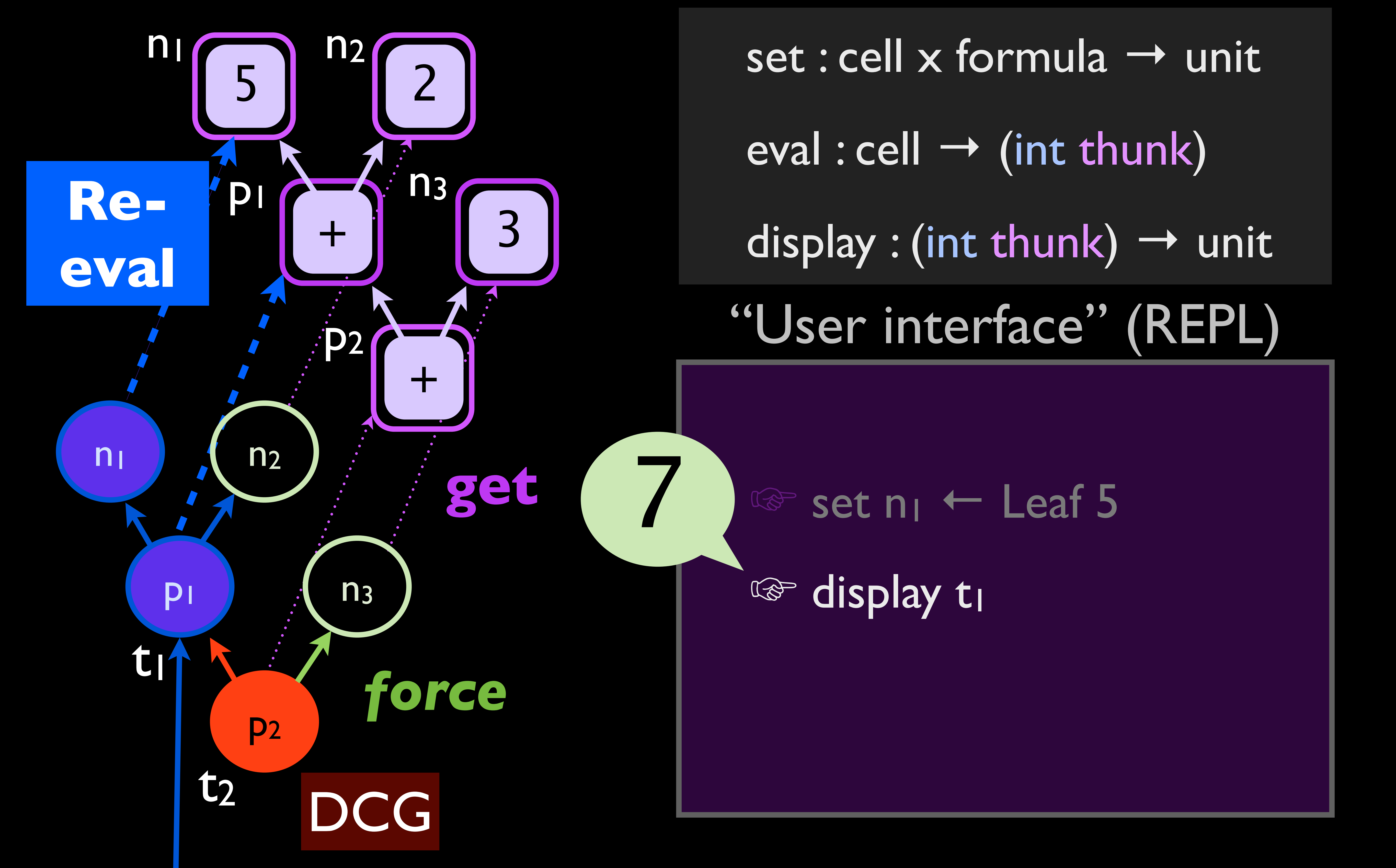
"User interface" (REPL)

☞ set n_1 ← Leaf 5

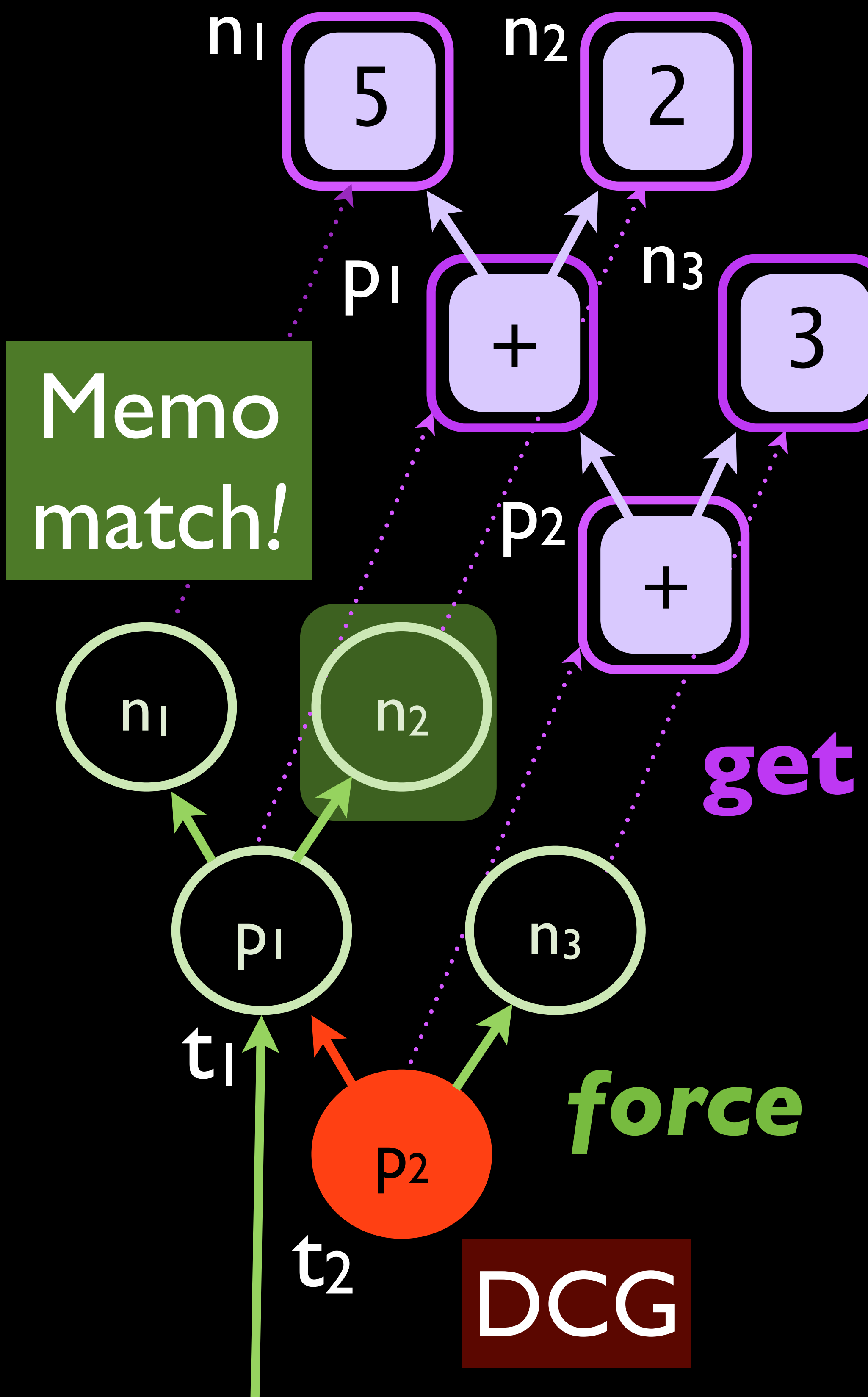
☞

Dirty phase

Spread Sheet Evaluator



Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

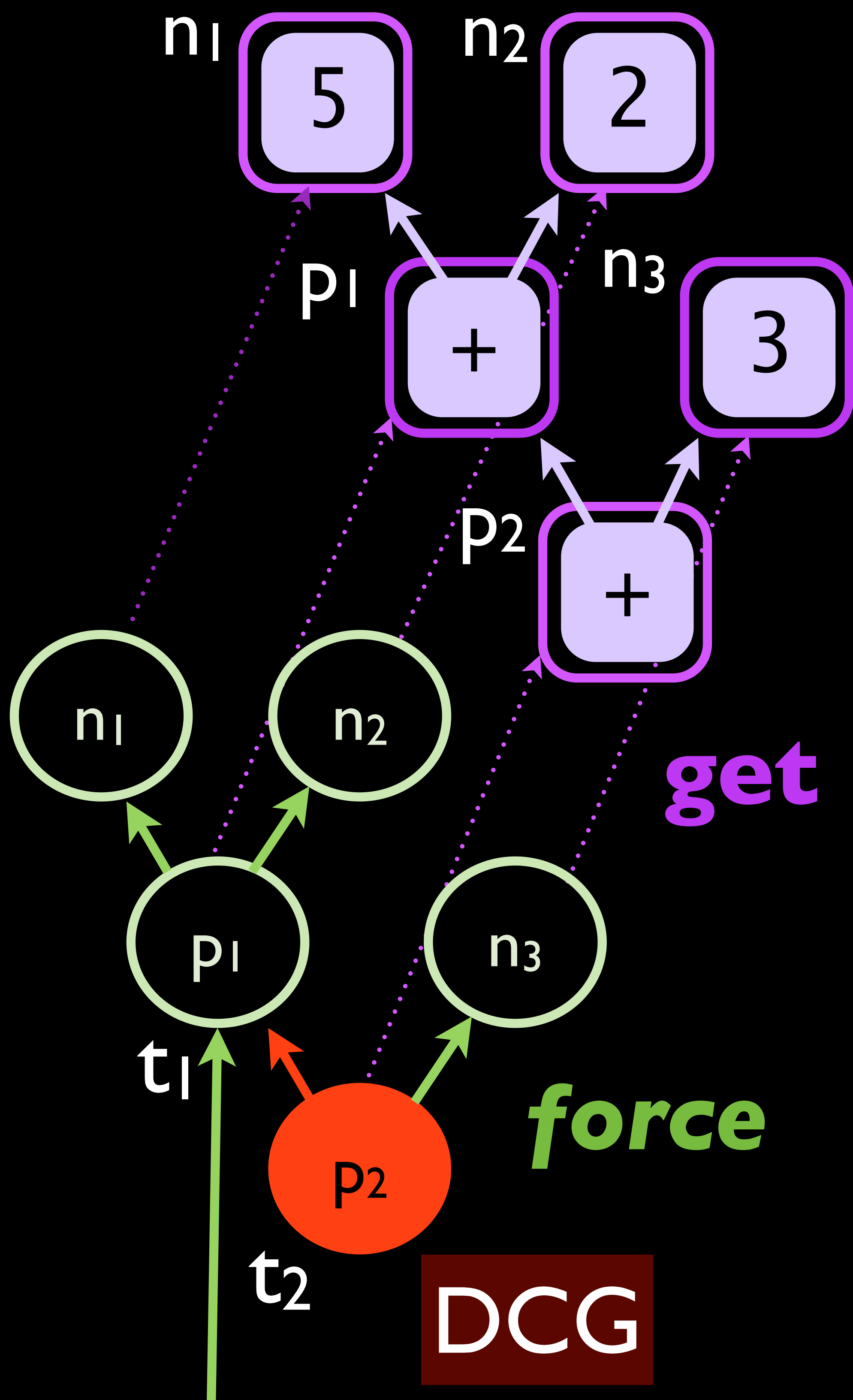
"User interface" (REPL)

7

☞ set n_1 ← Leaf 5

☞ display t_1

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

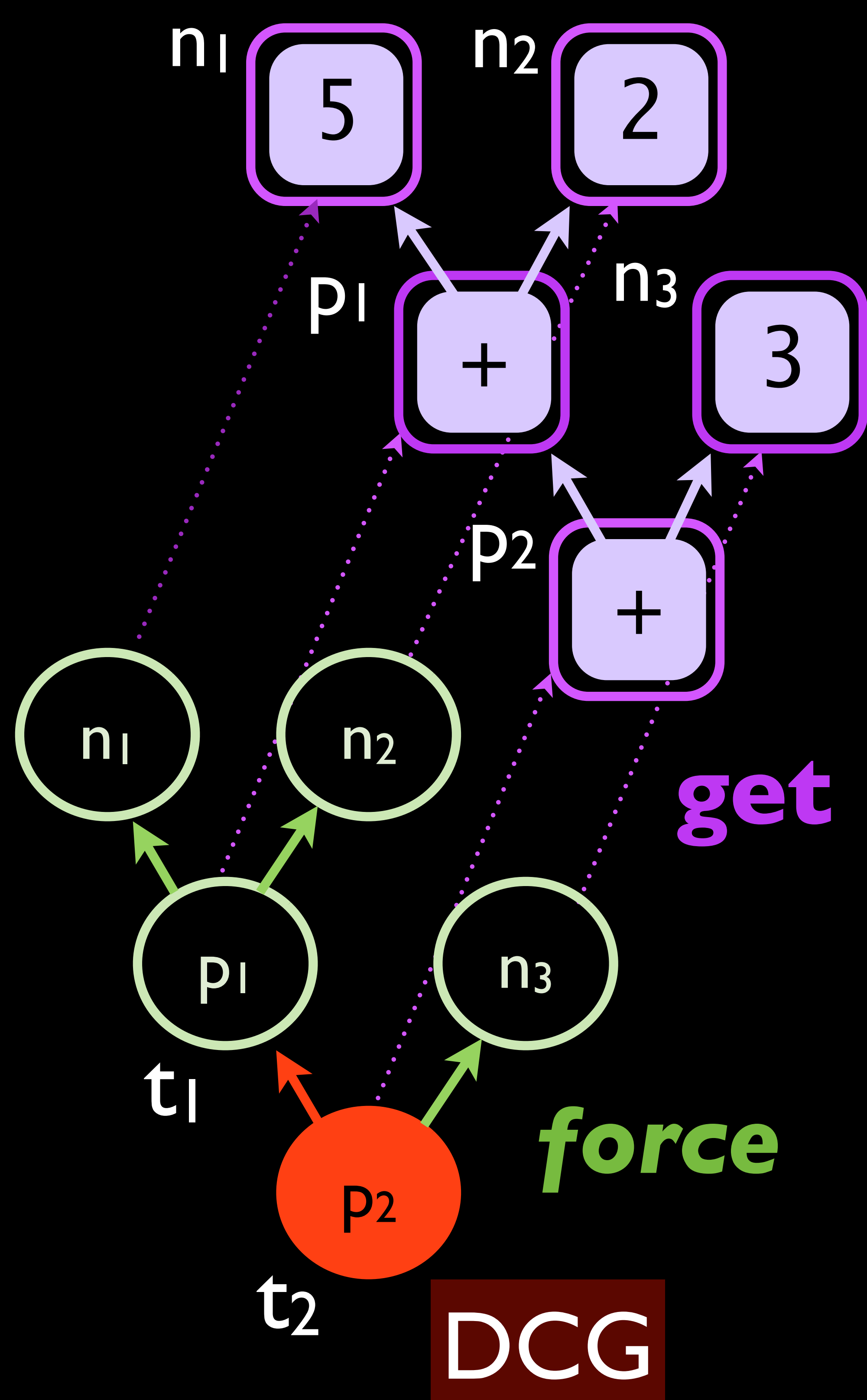
display : (int thunk) → unit

“User interface” (REPL)

☞ set n_1 ← Leaf 5

☞ display t_1

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

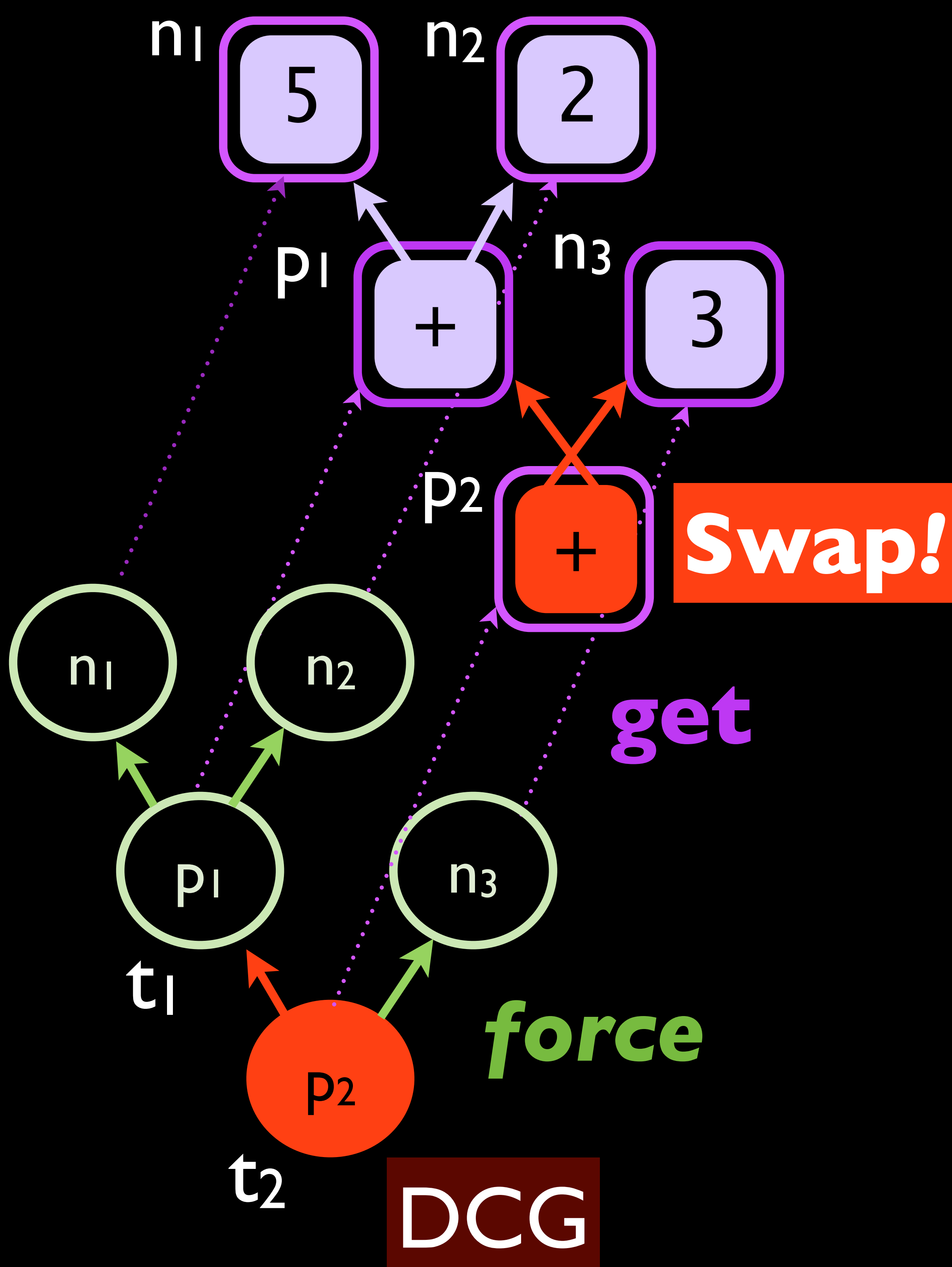
"User interface" (REPL)

☞ set n_1 ← Leaf 5

☞ display t_1

☞ set p_2 ← Plus(n_3,p_1)

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

"User interface" (REPL)

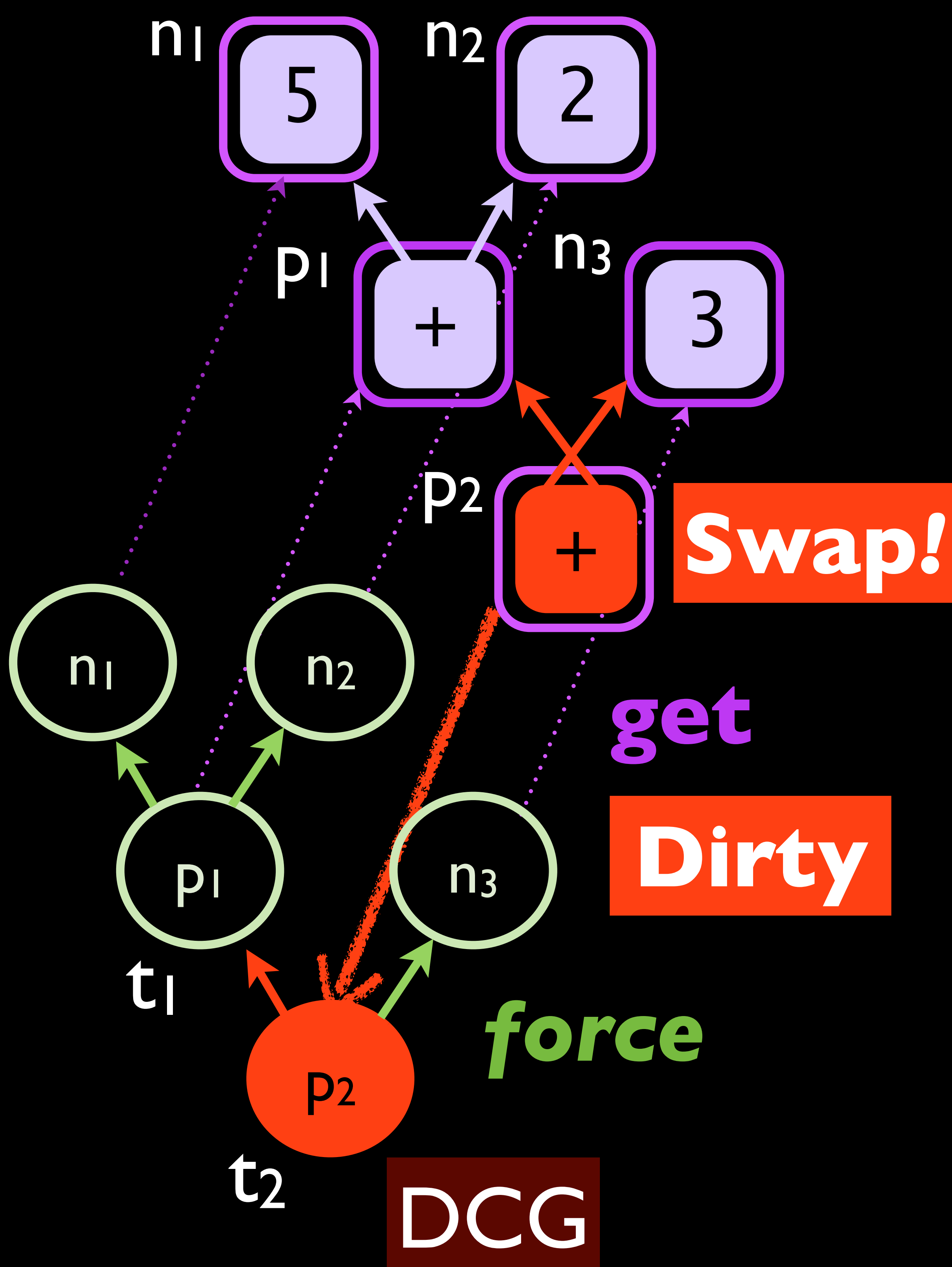
☞ set n_1 ← Leaf 5

☞ display t_1

☞ set p_2 ← Plus(n_3, p_1)

☞

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

"User interface" (REPL)

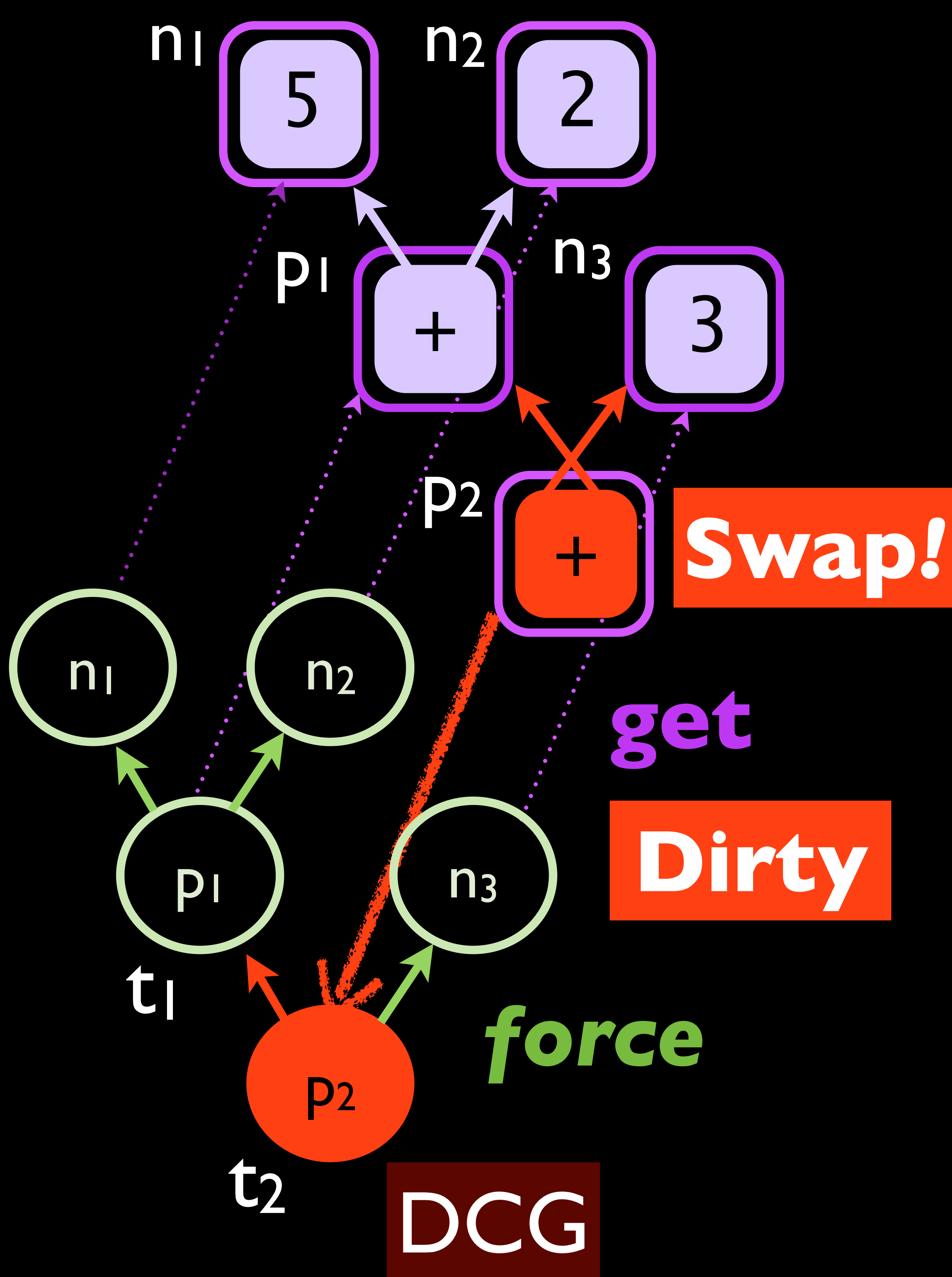
☞ set n_1 ← Leaf 5

☞ display t_1

☞ set p_2 ← Plus(n_3, p_1)

☞

Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

☞ set n_1 ← Leaf 5

☞ display t_1

☞ set p_2 ← Plus(n_3, p_1)

☞ display t_2

Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

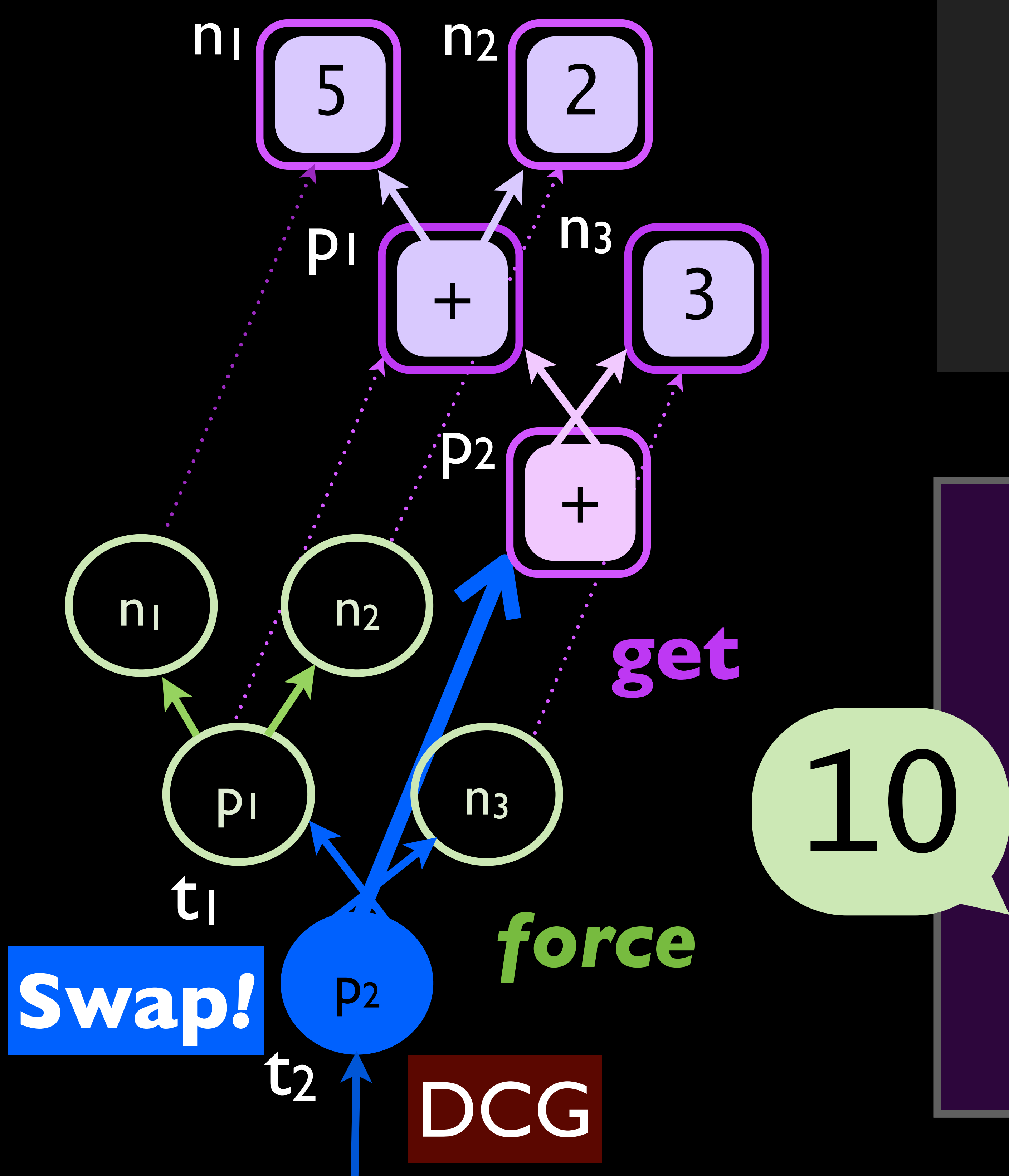
☞ set n_1 ← Leaf 5

☞ display t_1

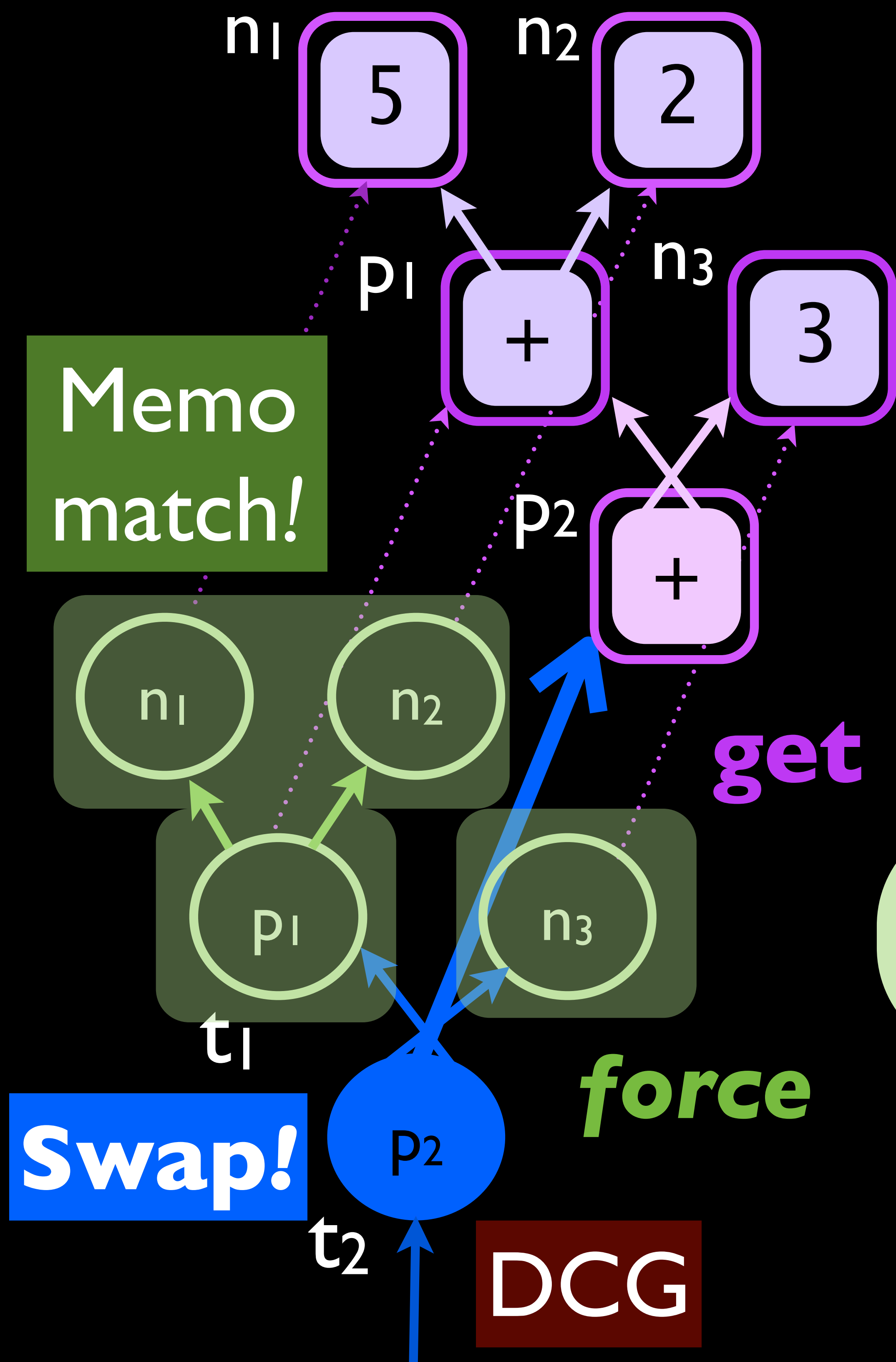
☞ set p_2 ← Plus(n_3, p_1)

☞ display t_2

☞



Spread Sheet Evaluator



set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

"User interface" (REPL)

☞ set n_1 ← Leaf 5

☞ display t_1

☞ set p_2 ← Plus(n_3, p_1)

☞ display t_2

10

☞

Lazy Structures

Laziness generalizes ***beyond scalars***

Recursive structures: **lists, trees** and **graphs**

```
type 'a lzlist =
  | Nil
  | Cons of 'a * ('a lzlist) thunk
```

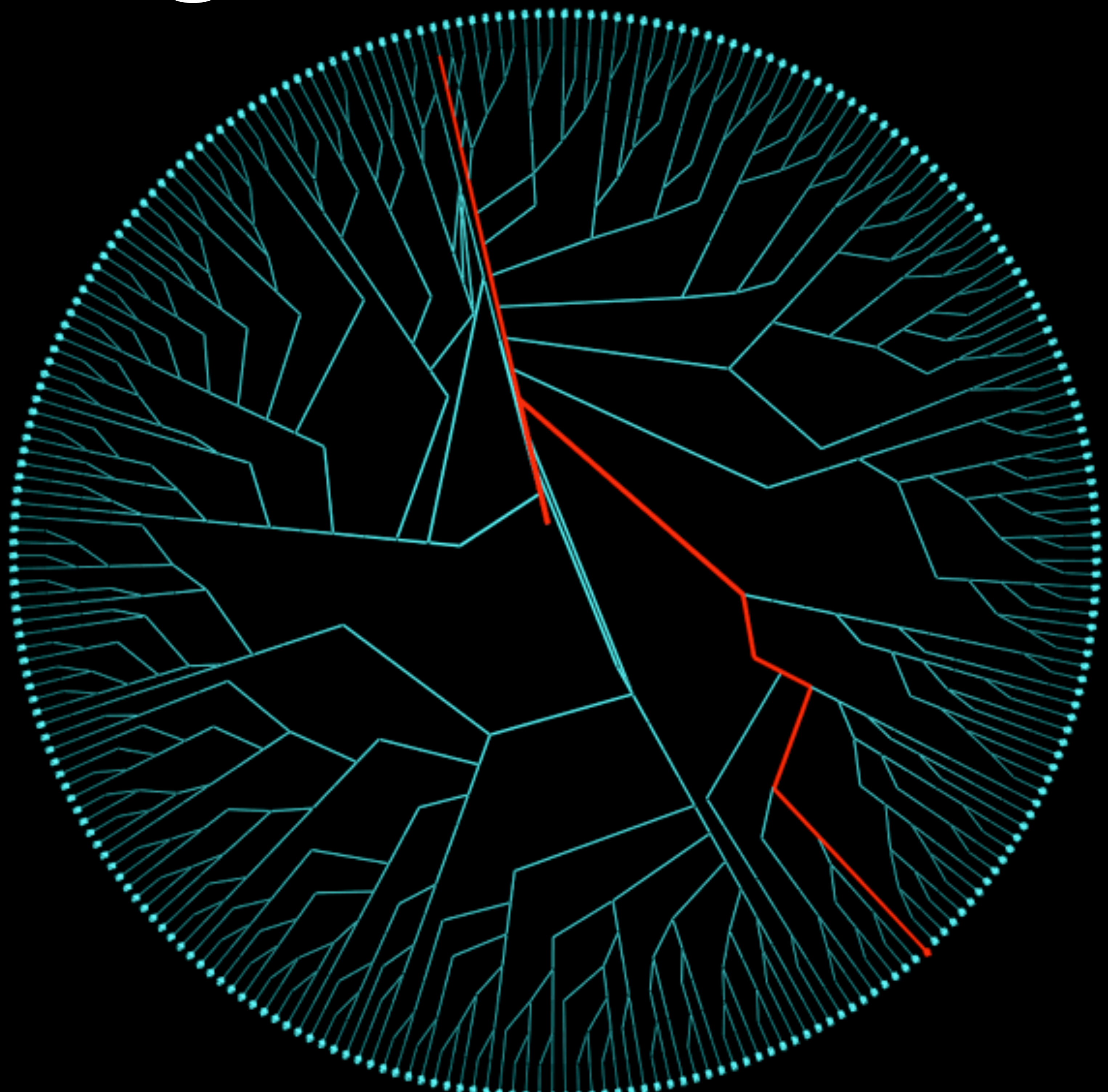
Recursive lazy structure

Merging Lazy Lists

As in conventional lazy programming

```
let rec merge l₁ l₂ = function
 | l₁, Nil ⇒ l₁
 | Nil, l₂ ⇒ l₂
 | Cons(h₁,t₁), Cons(h₂,t₂) ⇒
    if h₁ <= h₂ then
      Cons(h₁, thunk(merge (force t₁) l₂)
    else
      Cons(h₂, thunk(merge l₁ (force t₂))
```

Mergesort **DCG** Viz.



Graphics by ***Piotr Mardziel***

Micro Benchmarks

List and tree applications:

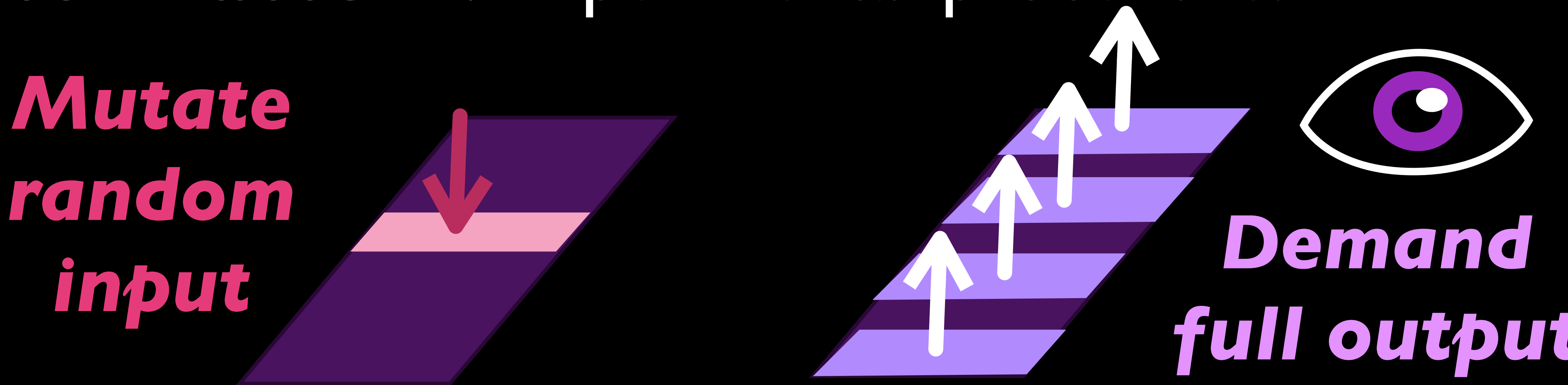
filter, map

fold{min,sum}

quicksort, mergesort

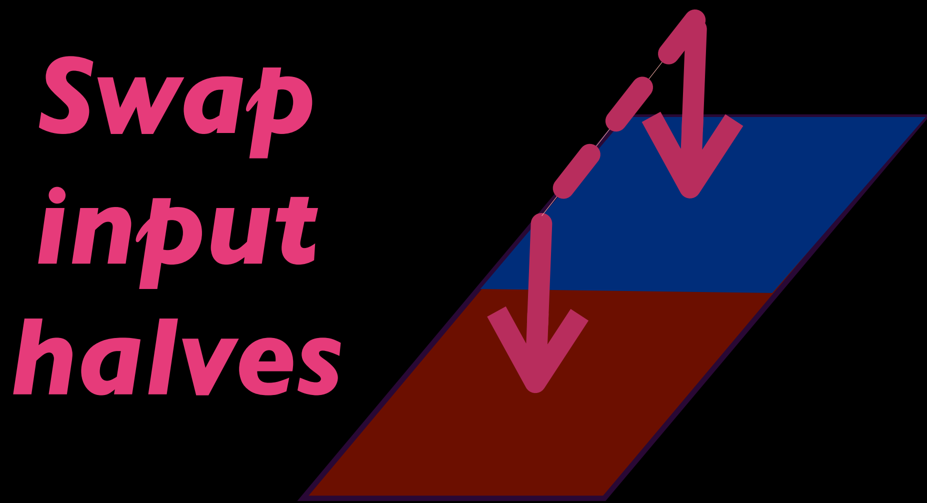
expression tree evaluation

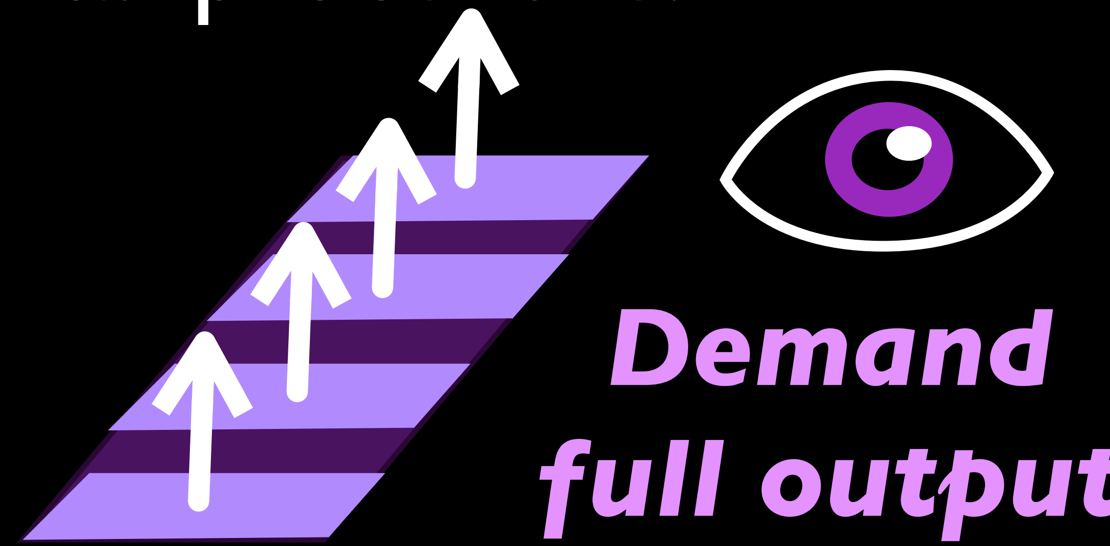
Batch Pattern: Experimental procedure:



Batch	**Baseline** time (s)	***Adapton*** speedup	***SAC*** speedup
filter	0.6	2.0	4.11
map	1.2	2.2	3.32
fold min	1.4	4350	3090
fold sum	1.5	1640	4220
exptree	0.3	497	1490

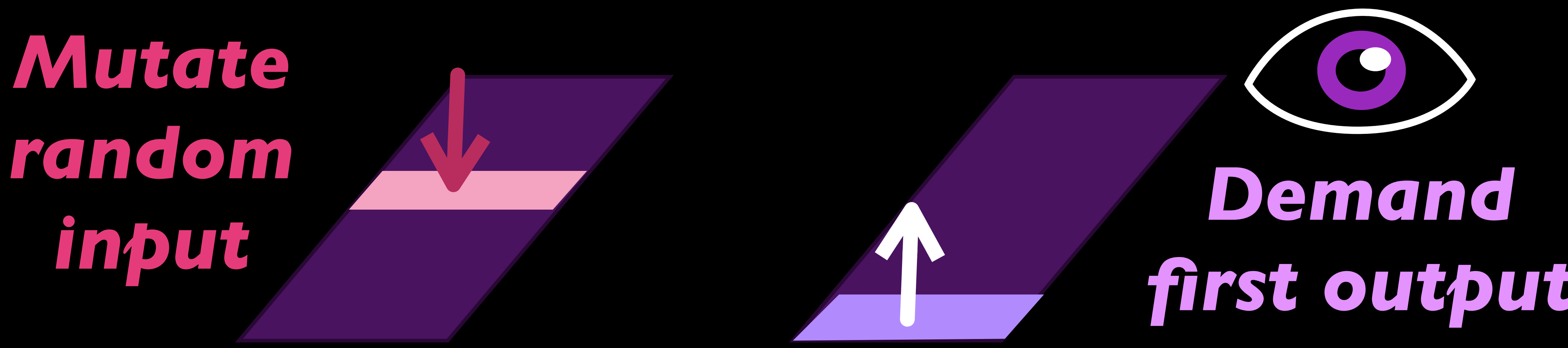
Swap Pattern: Experimental procedure:





Swap	**Baseline** time (s)	***Adapton*** speedup	***SAC*** speedup
filter	0.5	2.0	0.14
map	0.9	2.4	0.25
fold min	1.0	472	0.12
fold sum	1.1	501	0.13
exptree	0.3	667	10

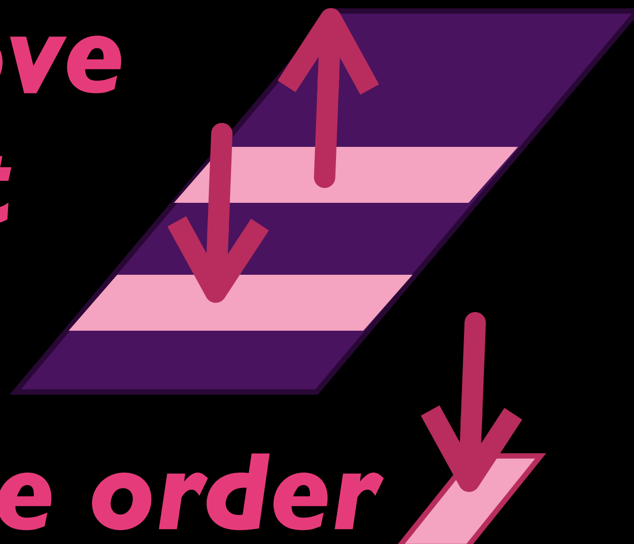
Lazy Pattern: Experimental procedure:



Lazy	**Baseline** time (s)	***Adapton*** speedup	***SAC*** speedup
filter	1.16E-05	12.8	2.2
map	6.86E-06	7.8	1.5
quicksort	7.41E-02	2020	22.9
mergesort	3.46E-01	336	0.148

Switch Pattern: Experimental procedure:

1. Remove
2. Insert

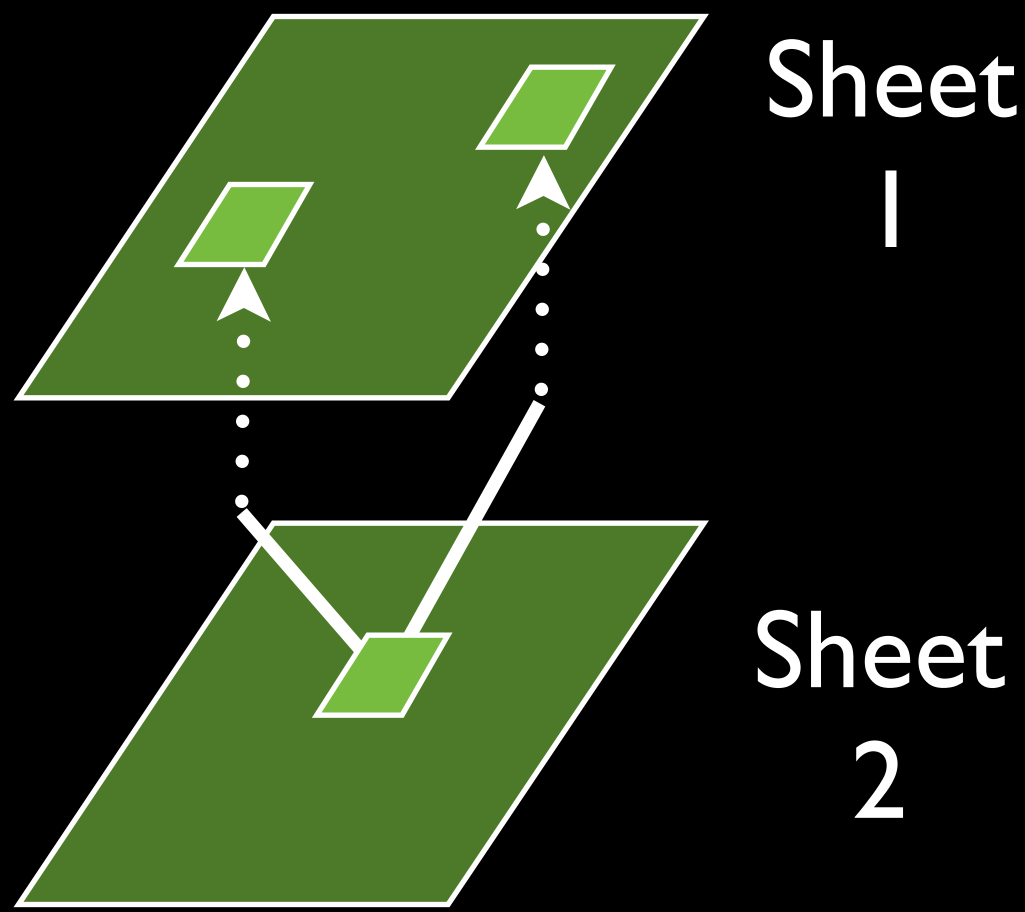


3. Toggle order

Demand first output

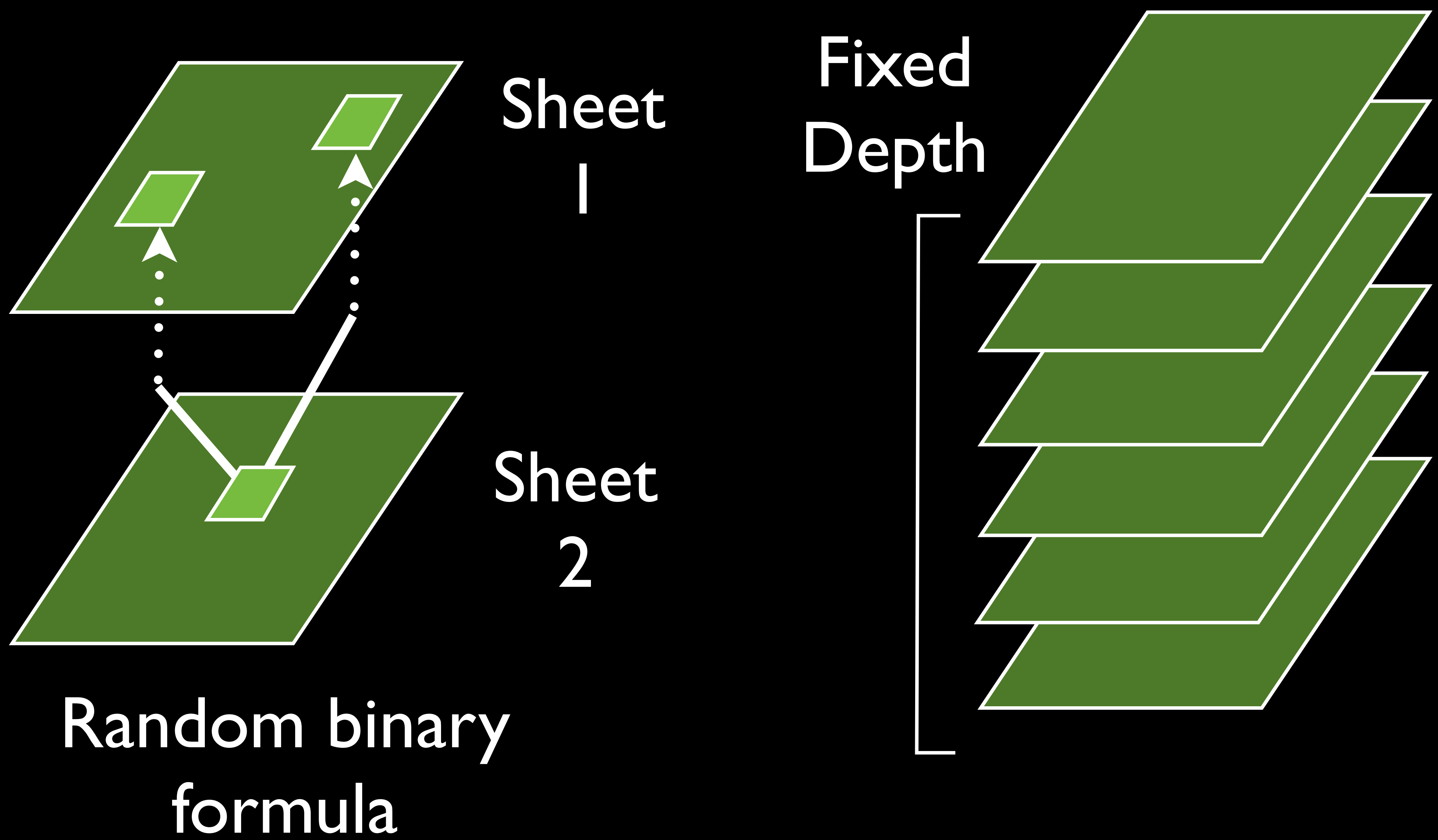
Switch	**Baseline** time (s)	***Adapton*** speedup	***SAC*** speedup
updown1	3.28E-02	22.4	2.47E-03
updown2	3.26E-02	24.7	4.28

Spreadsheet Experiments

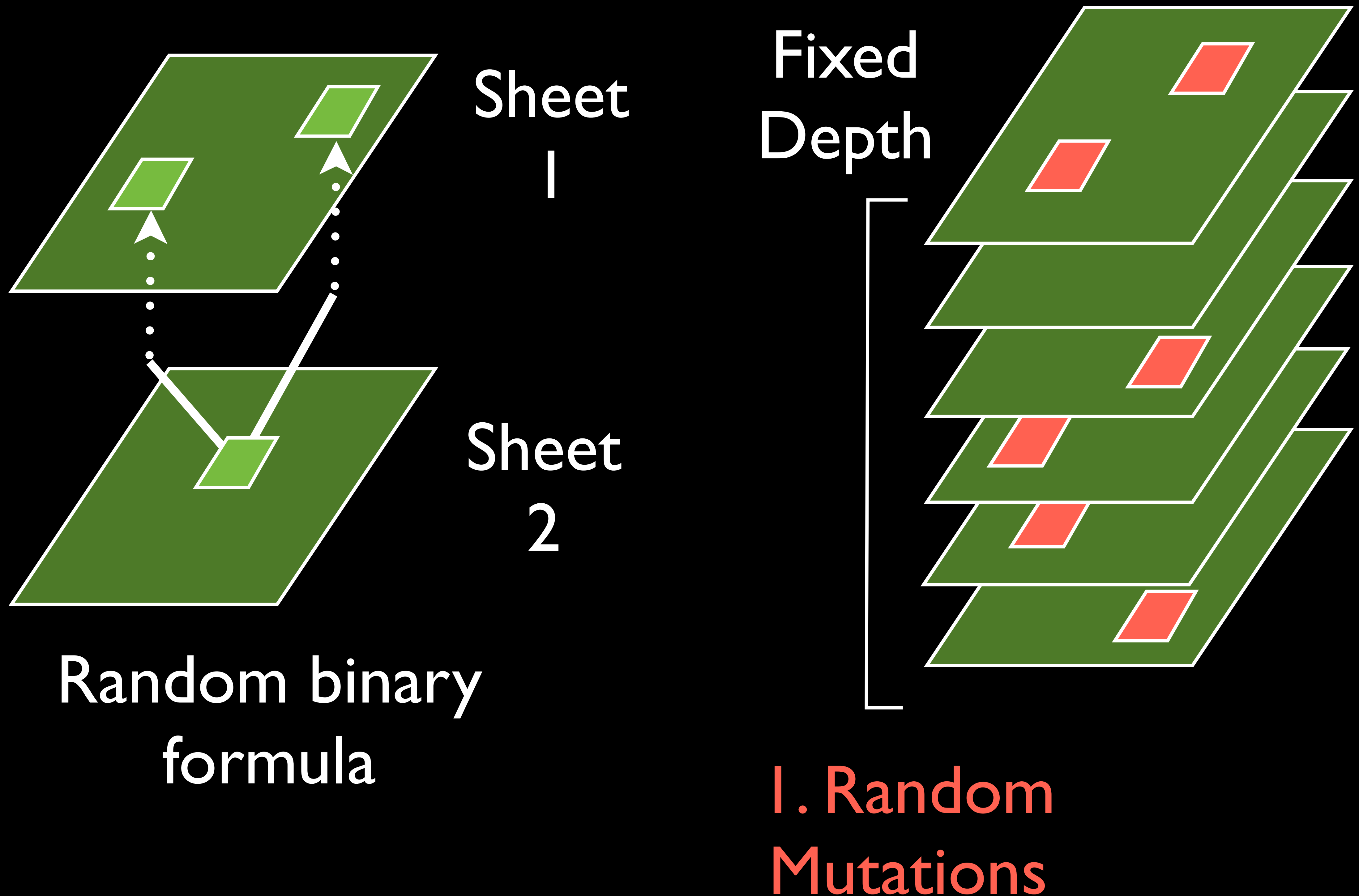


Random binary formula

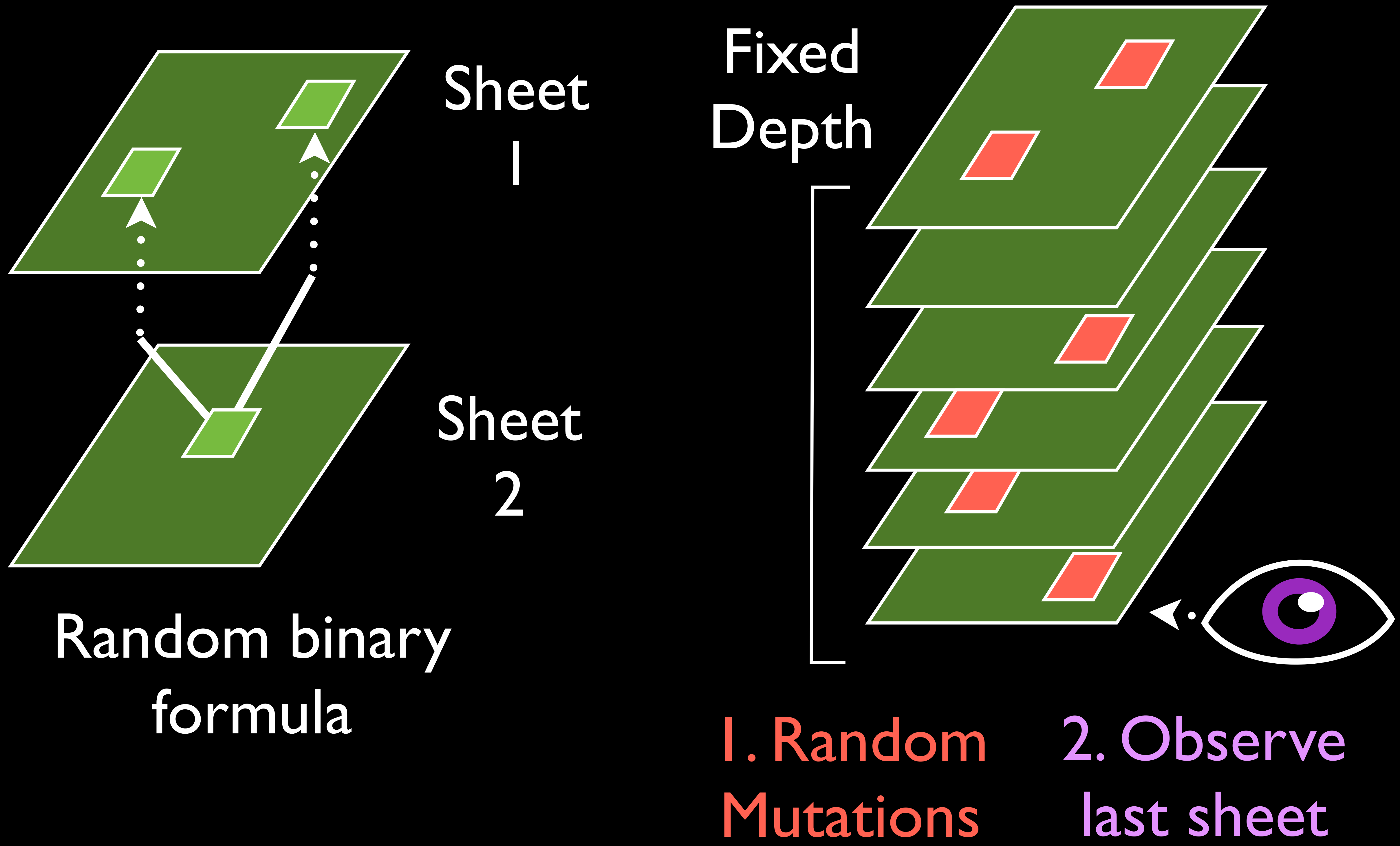
Spreadsheet Experiments



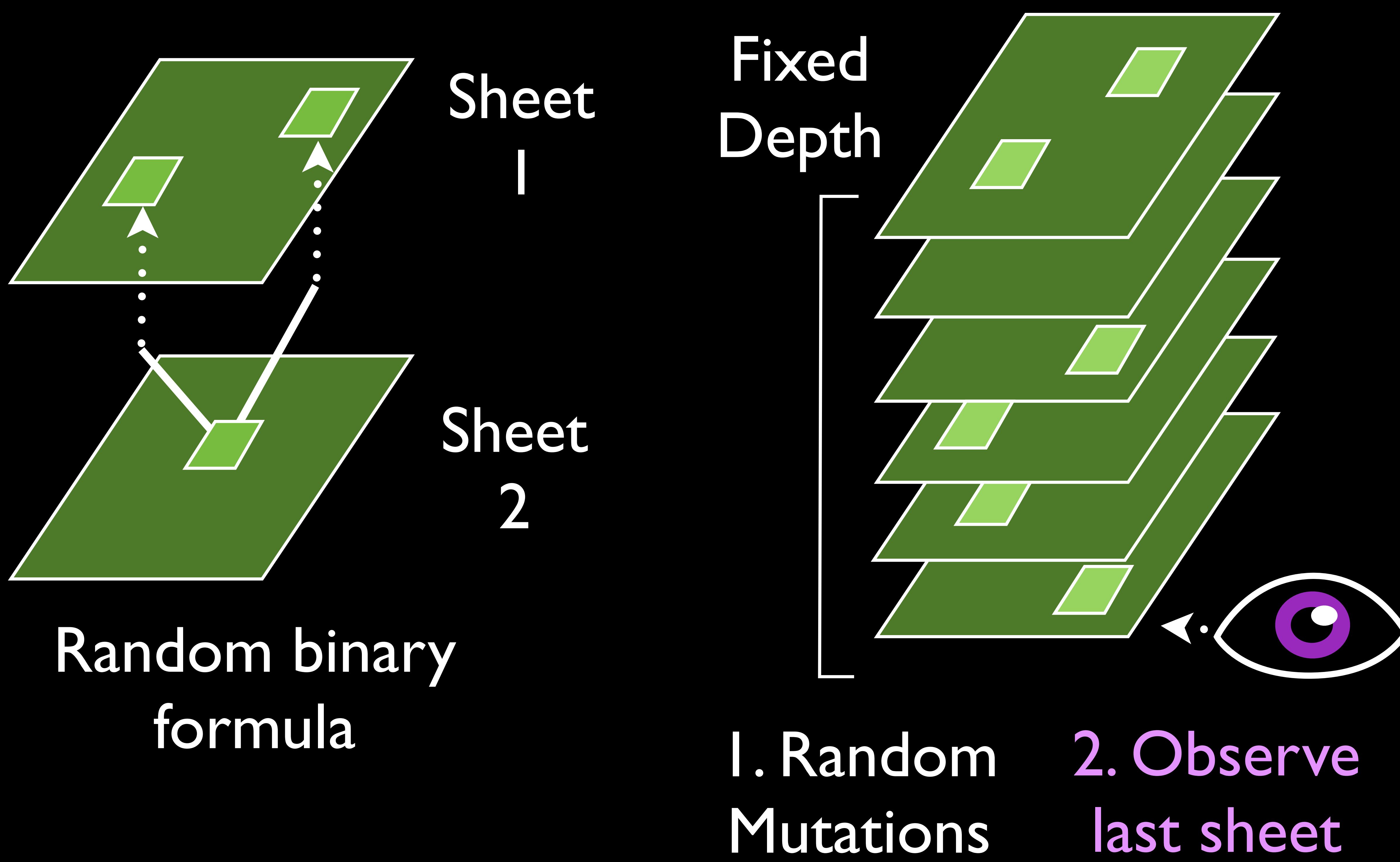
Spreadsheet Experiments



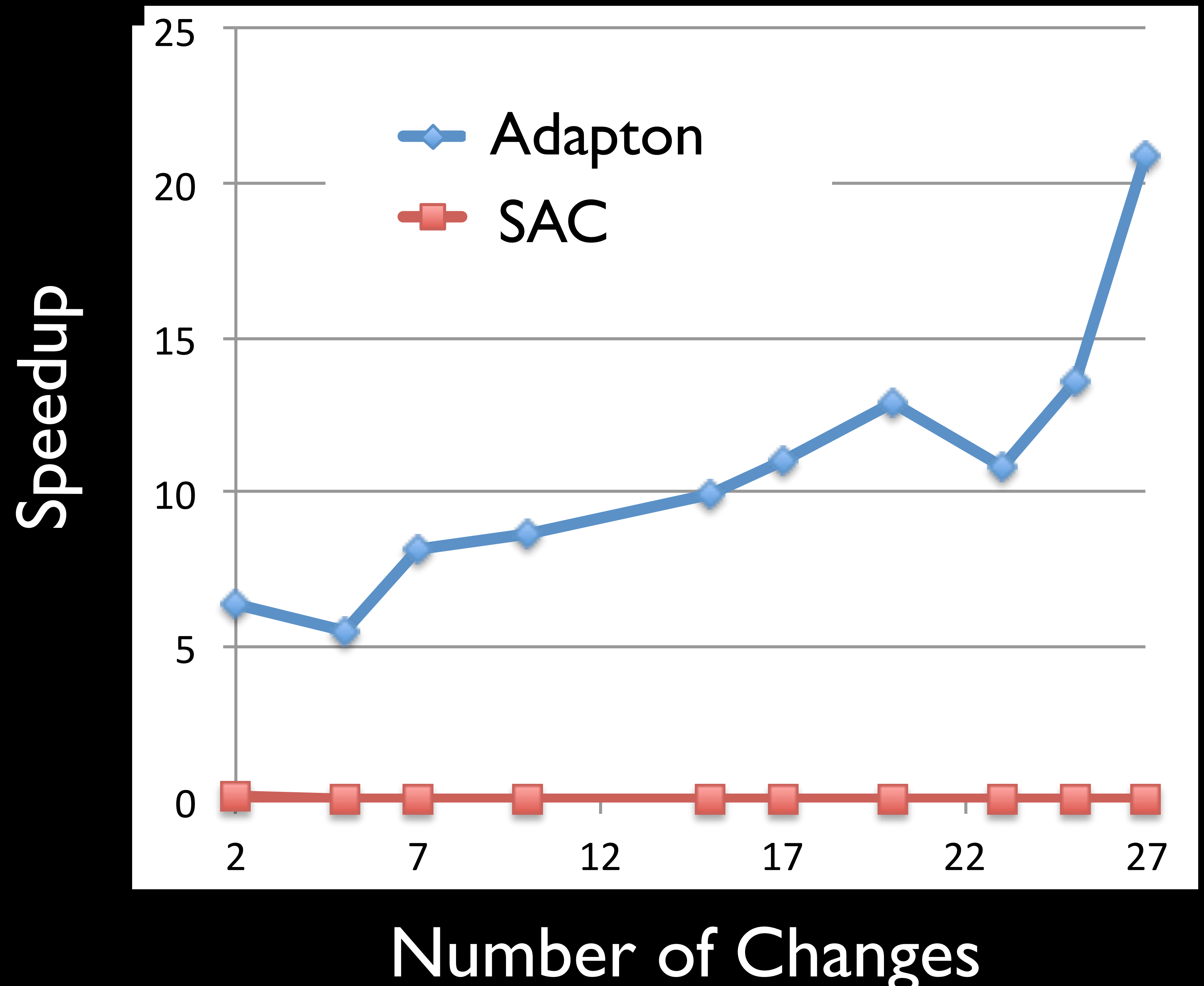
Spreadsheet Experiments



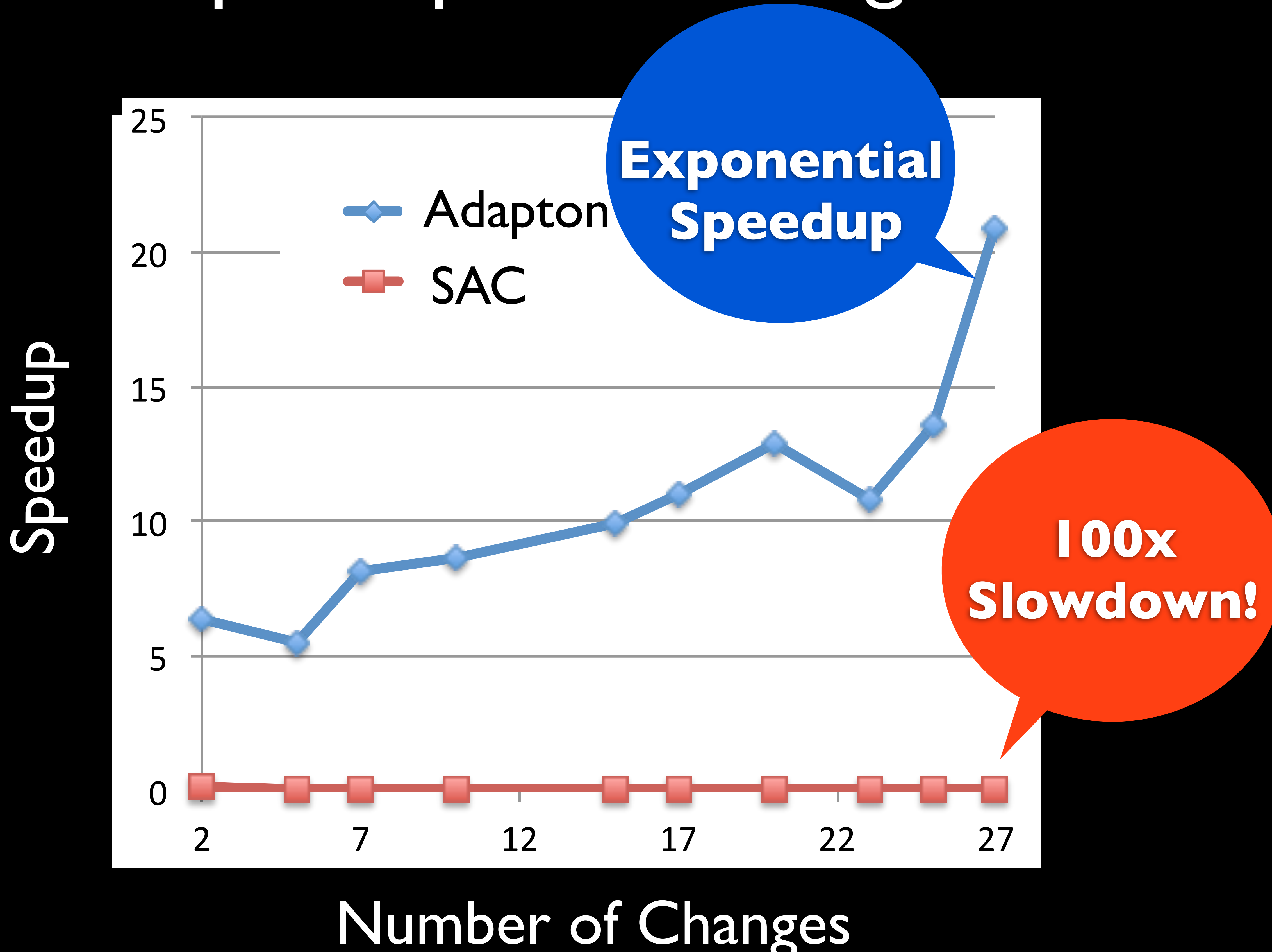
Spreadsheet Experiments



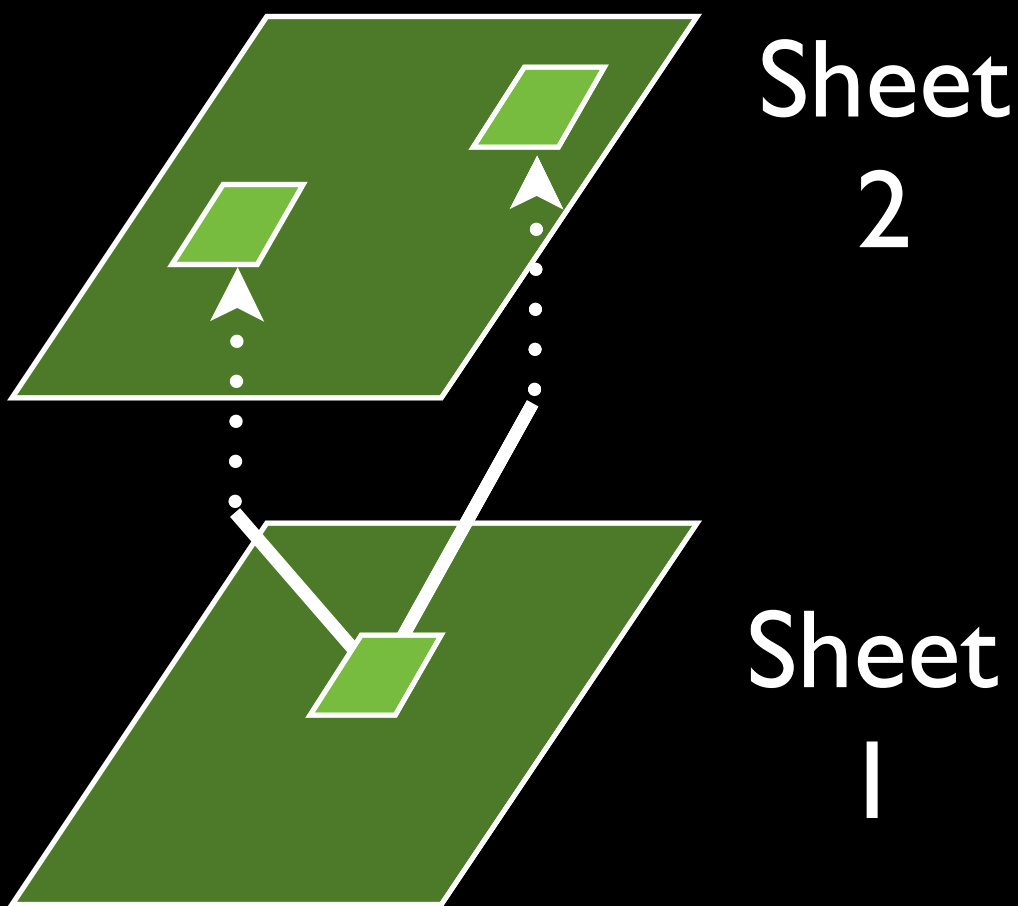
Speedup vs # Changes
(15 sheets deep)



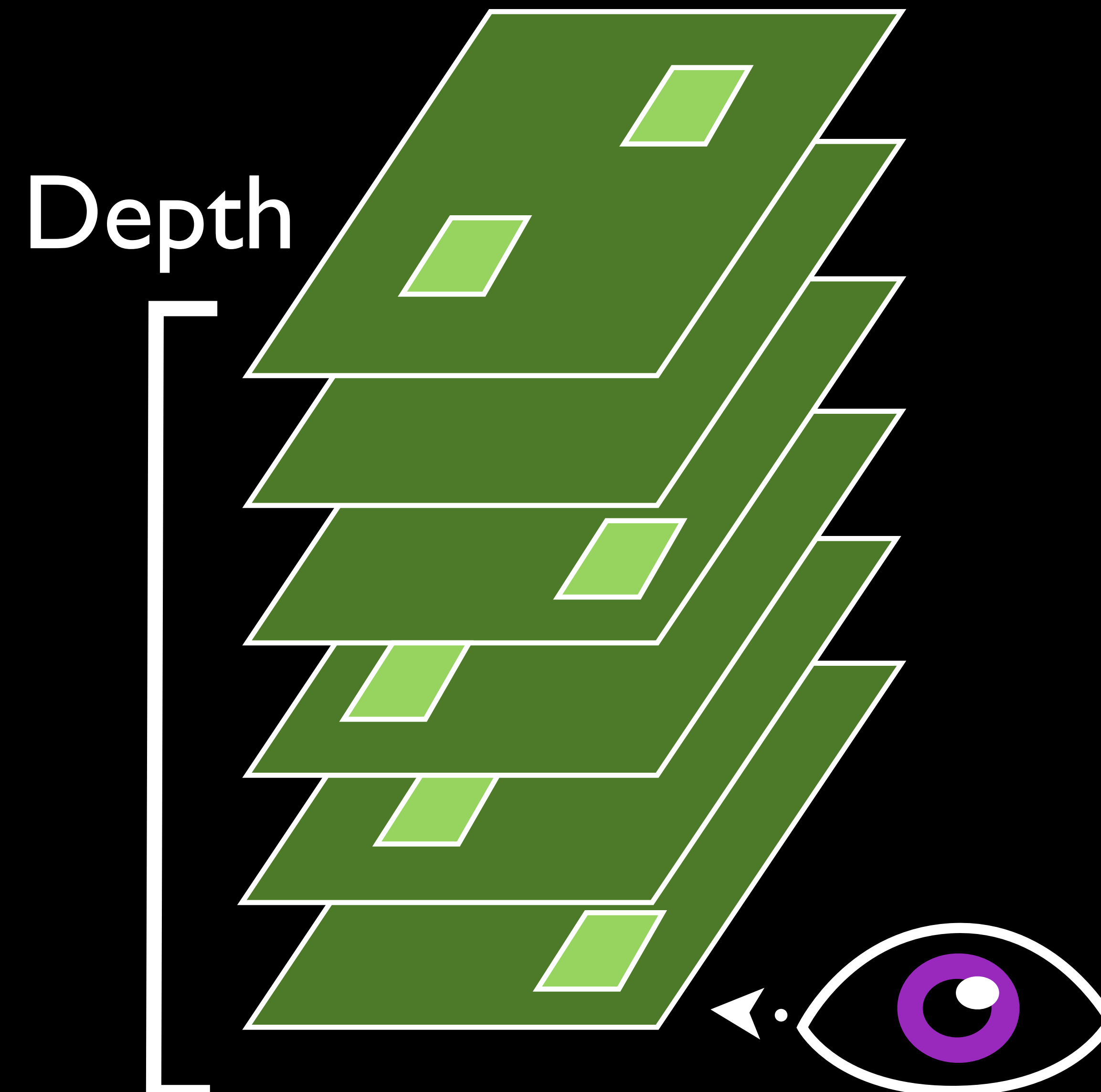
Speedup vs # Changes



Spreadsheet Experiments



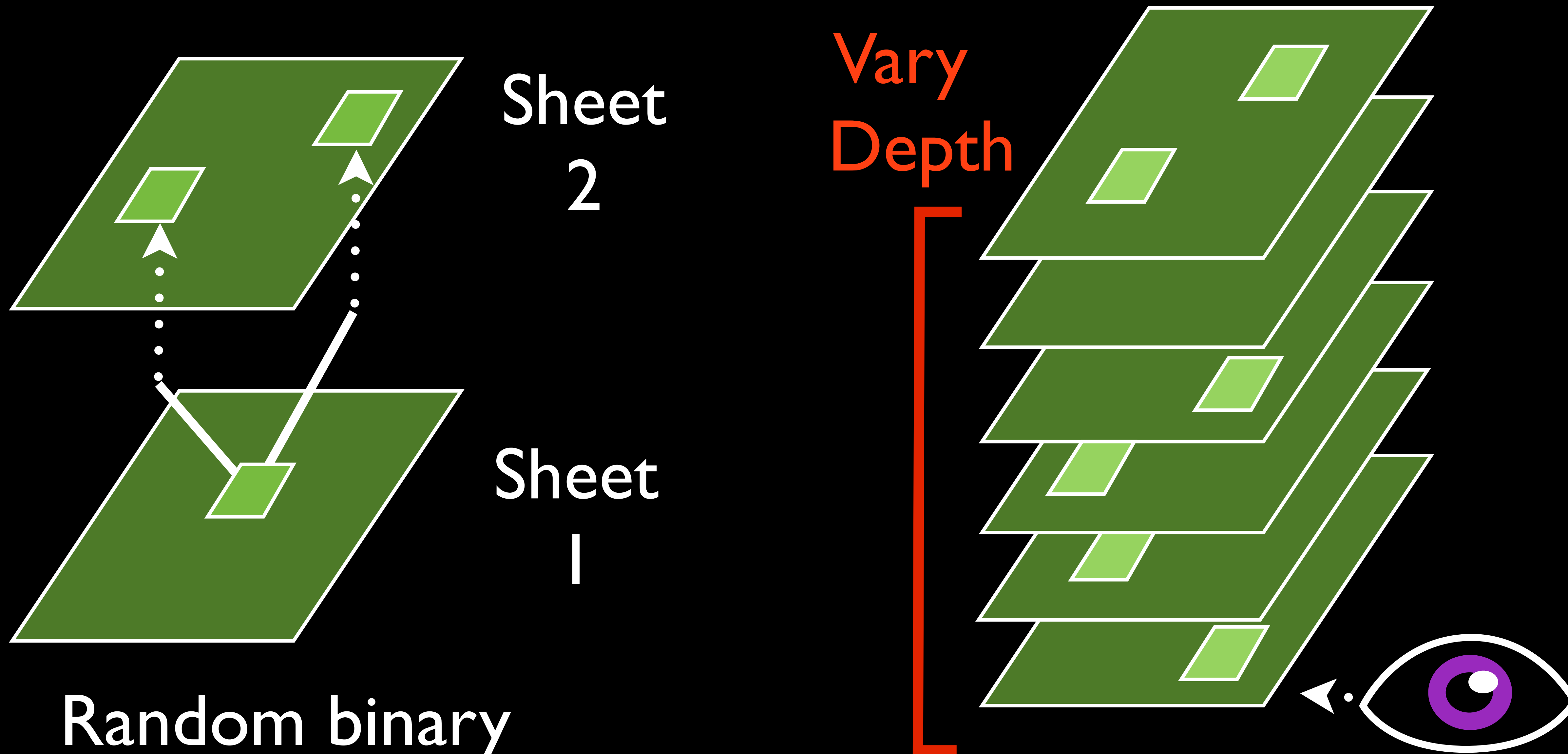
Random binary formula



1. Random Mutations

2. Observe last sheet

Spreadsheet Experiments

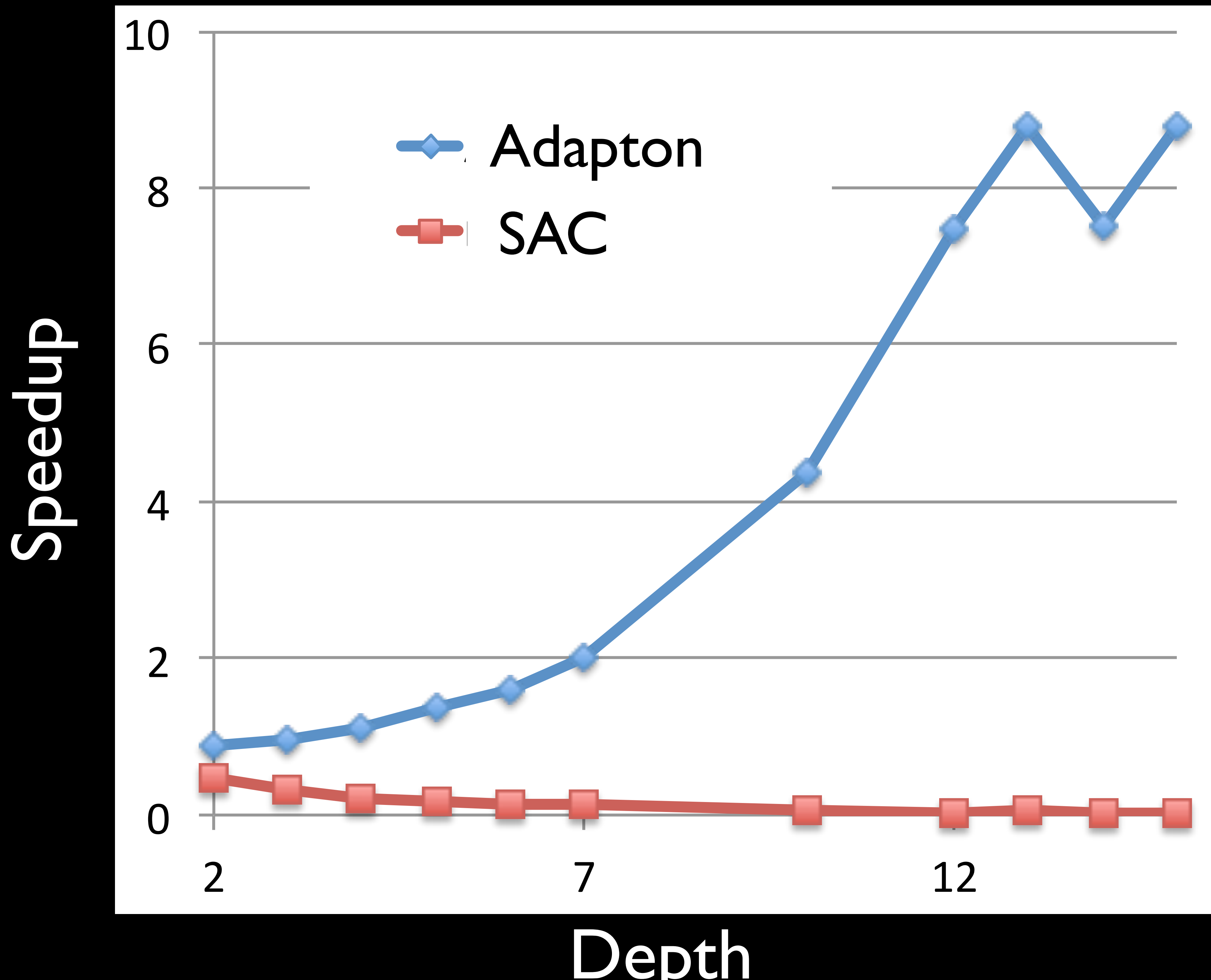


Random binary formula

1. Random Mutations

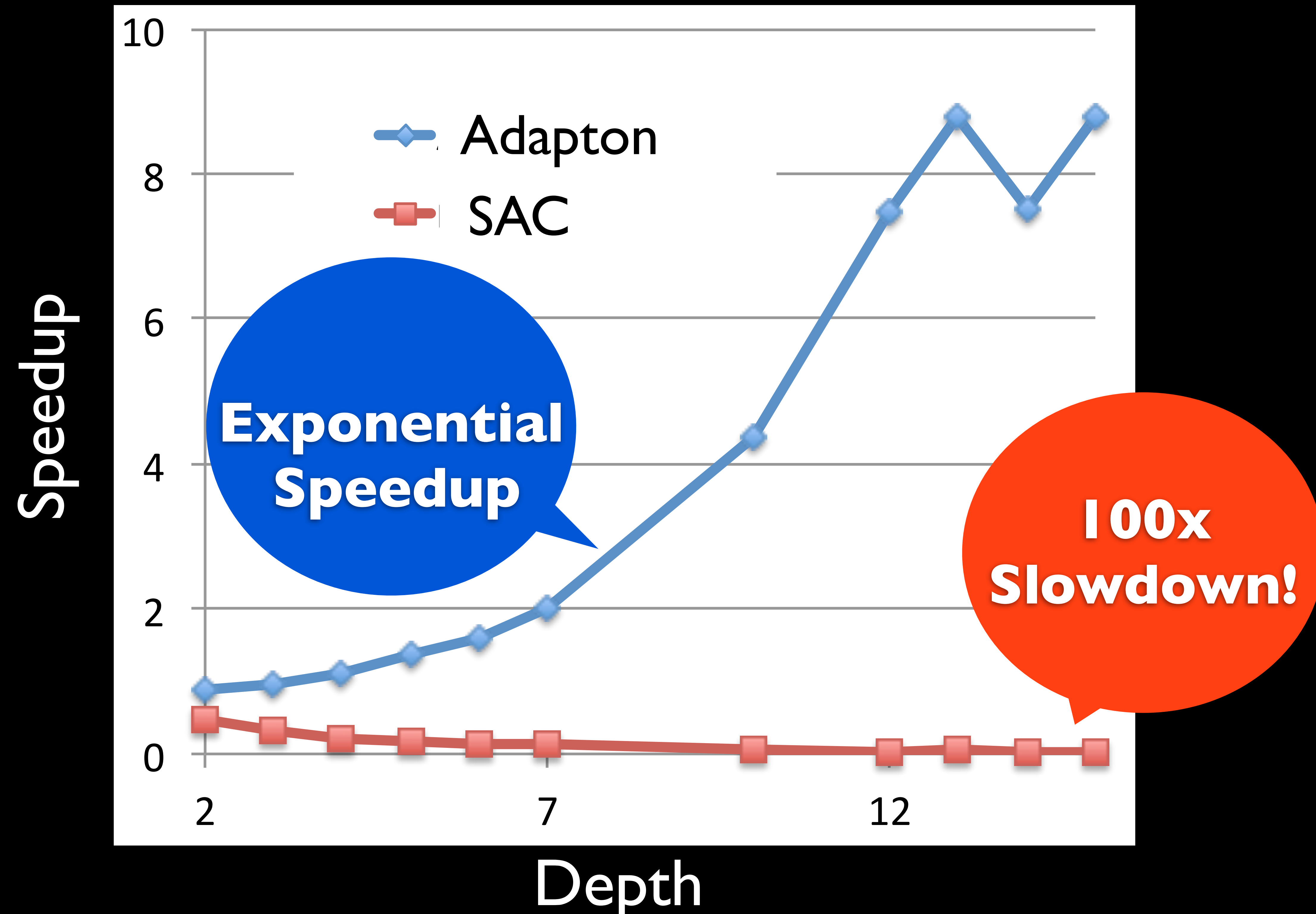
2. Observe last sheet

Speedup vs Sheet Depth
(10 changes between observations)



Speedup vs Sheet Depth

(10 changes between observations)



Paper and Technical Report

- ***Formal semantics*** of Adapton
- ***Algorithms*** to implement Adapton
- More empirical data and analysis

Aside: Formal Semantics

- ***CBPV*** + ***Refs*** + ***Layers*** *(outer versus inner)*
- Syntax for **traces** and **knowledge** formally represents **DCG** structure
- Formal specification of change propagation
- ***Theorems:***
  - ***Type soundness***
  - ***Incremental soundness***
    (" from-scratch consistency ")

Summary

- **Adapton**: Composable, Demand-Driven IC
  - ***Demand-driven*** change propagation
  - Reuse patterns:
    Sharing, swapping and switching
- Formal specification (see paper)
- Implemented in OCaml (and Python)
- Empirical evaluation shows **speedups**

http://ter.ps/adapton

	pattern	input #	LazyNonInc baseline time (s)	LazyNonInc baseline mem (MB)	ADAPTON vs. LazyNonInc time spdup	ADAPTON vs. LazyNonInc mem ovrhd	EagerTotalOrder vs. LazyNonInc time spdup	EagerTotalOrder vs. LazyNonInc mem ovrhd
filter	lazy	1e6	1.16e-5	96.7	12.8	2.7	2.24	8.0
map	lazy	1e6	6.85e-6	96.7	7.80	2.7	1.53	8.0
quicksort	lazy	1e5	0.0741	18.6	2020	8.7	22.9	144.1
mergesort	lazy	1e5	0.346	50.8	336	7.8	0.148	96.5
filter	swap	1e6	0.502	157	1.99	10.1	0.143	17.3
map	swap	1e6	0.894	232	2.36	6.9	0.248	12.5
fold(min)	swap	1e6	1.04	179	472	9.1	0.123	33.9
fold(sum)	swap	1e6	1.11	180	501	9.1	0.128	33.8
exptree	swap	1e6	0.307	152	667	11.7	10.1	11.9
updown1	switch	4e4	0.0328	8.63	22.4	14.0	0.00247	429.9
updown2	switch	4e4	0.0326	8.63	24.7	13.8	4.28	245.7
filter	batch	1e6	0.629	157	2.04	10.1	4.11	9.0
map	batch	1e6	1.20	232	2.21	6.9	3.32	6.6
fold(min)	batch	1e6	1.43	179	4350	9.0	3090	8.0
fold(sum)	batch	1e6	1.48	180	1640	9.1	4220	8.0
exptree	batch	1e6	0.308	152	497	11.7	1490	9.7