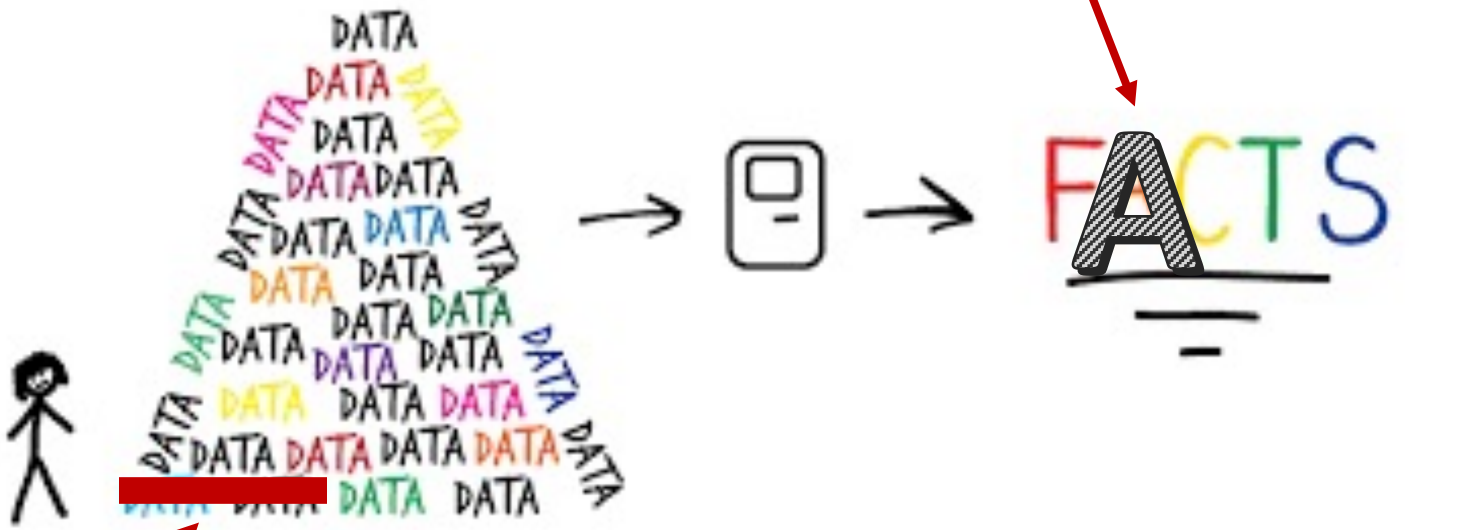
Incremental Computation

COS 326
David Walker
Princeton University

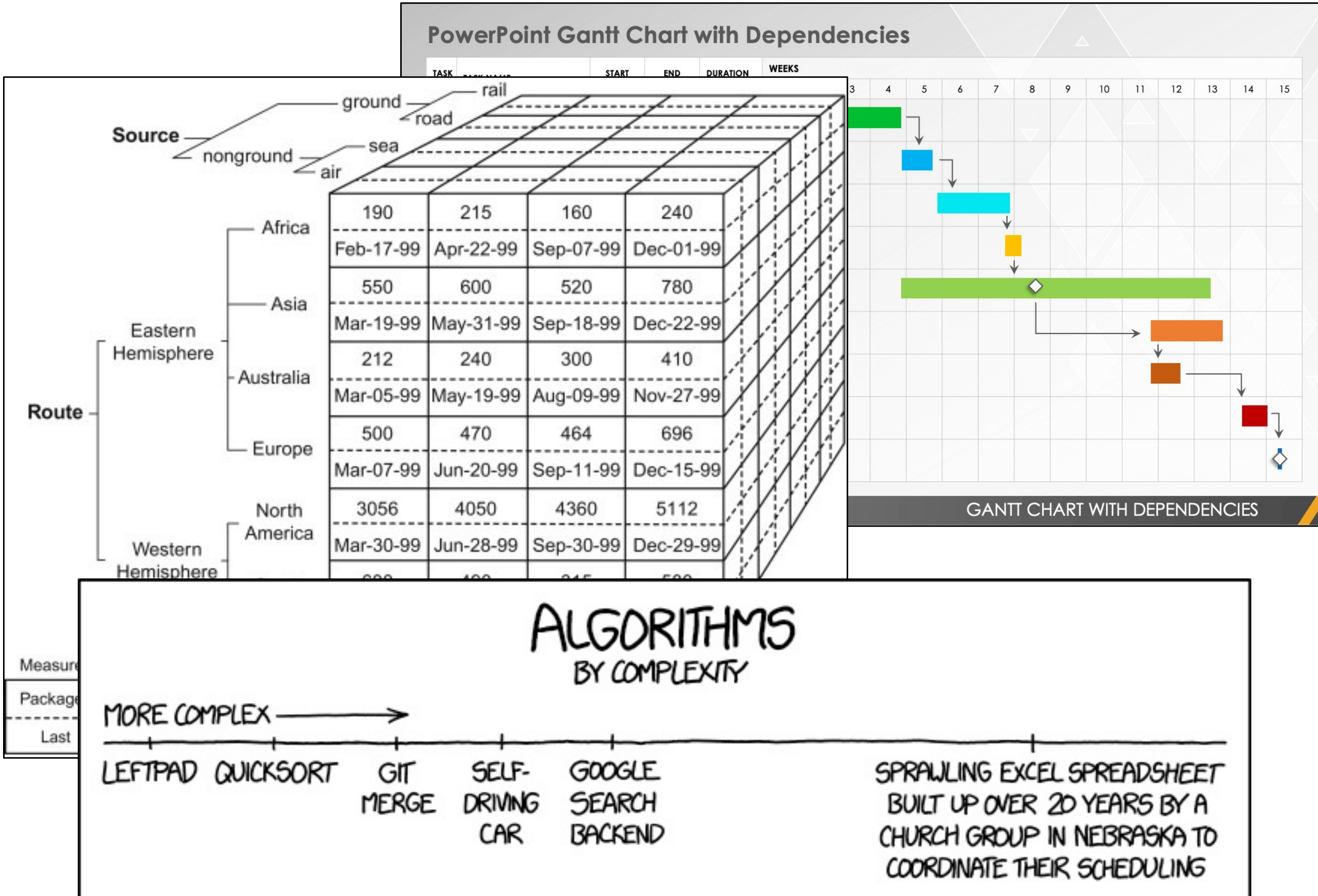

DATA
DATA DATA DATA
DATA
DATADATA
DATA DATA DATA DATA
DATA DATA
DATA DATA DATA
DATA DATA DATA DATA
DATA DATA DATA DATA
DATA DATA DATA DATA DATA DATA
DATA DATA DATA DATA
FACTS

recompute facts

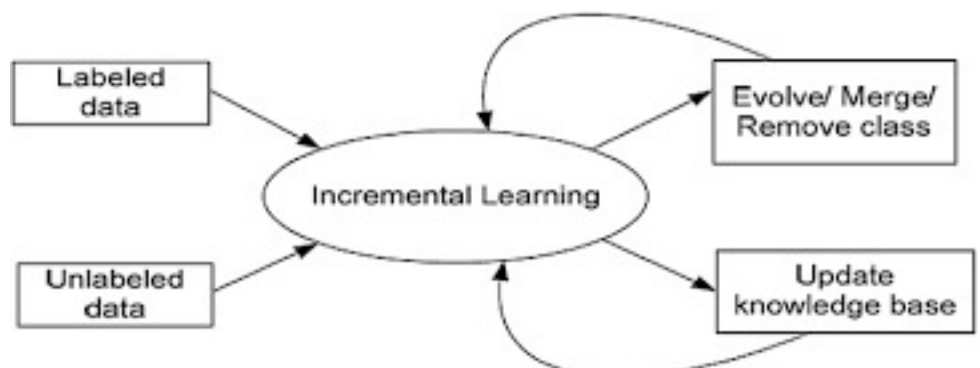


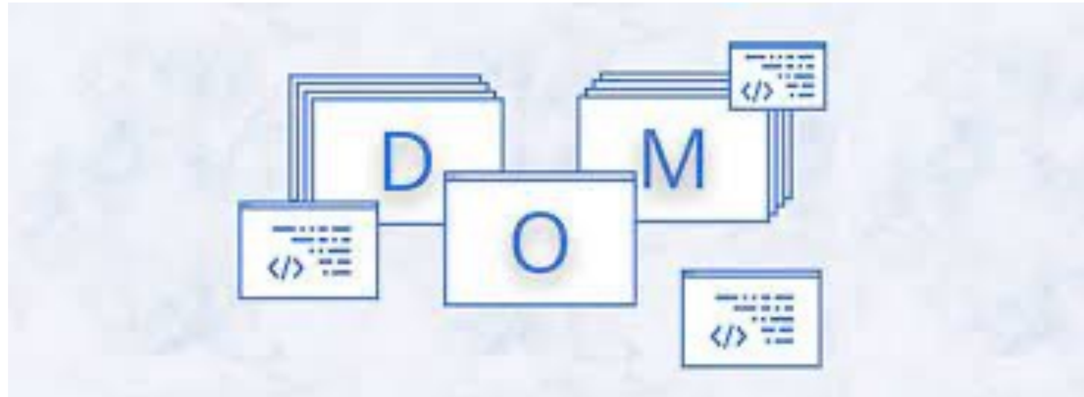
data, data

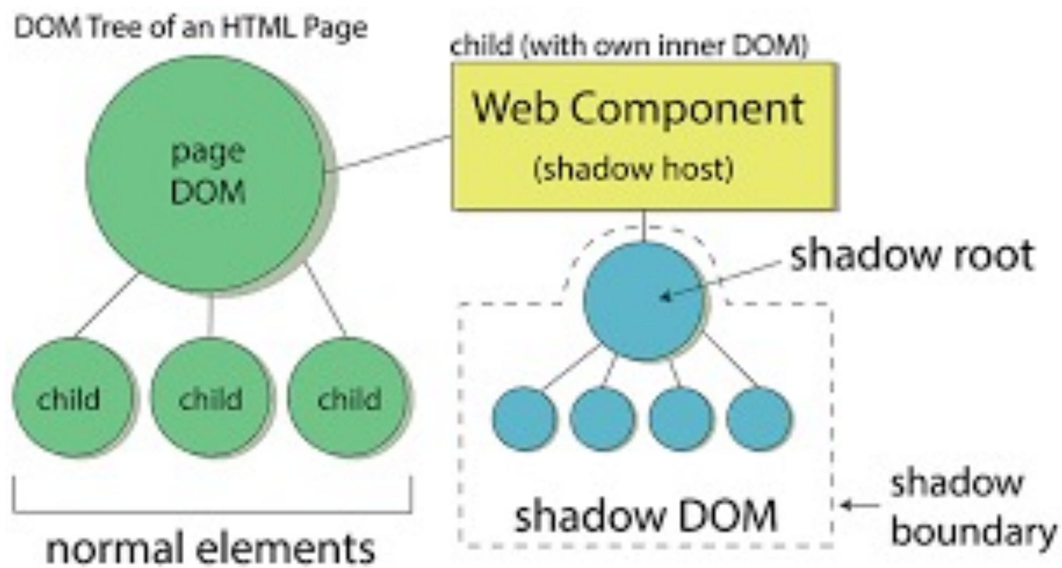
Spreadsheets




THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

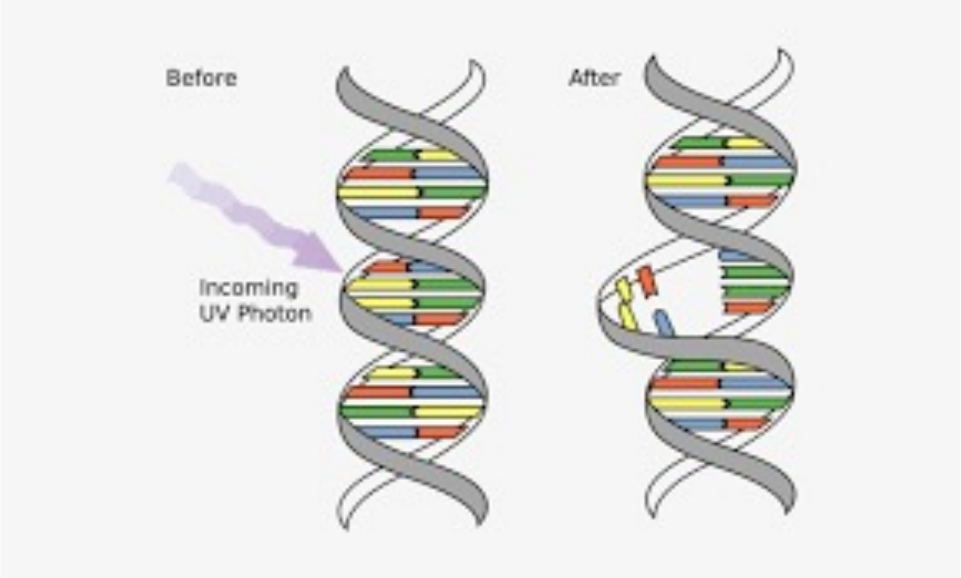

Labeled data
Unlabeled data
Incremental Learning
Evolve/ Merge/ Remove class
Update knowledge base

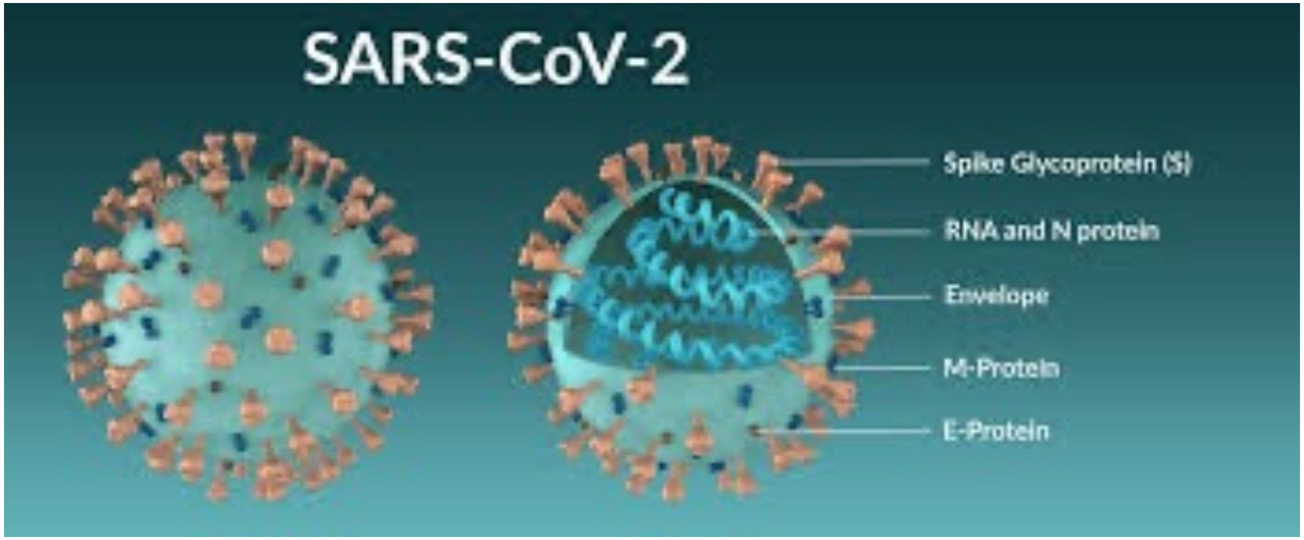




https://blog.janestreet.com/incrementality-and-the-web/

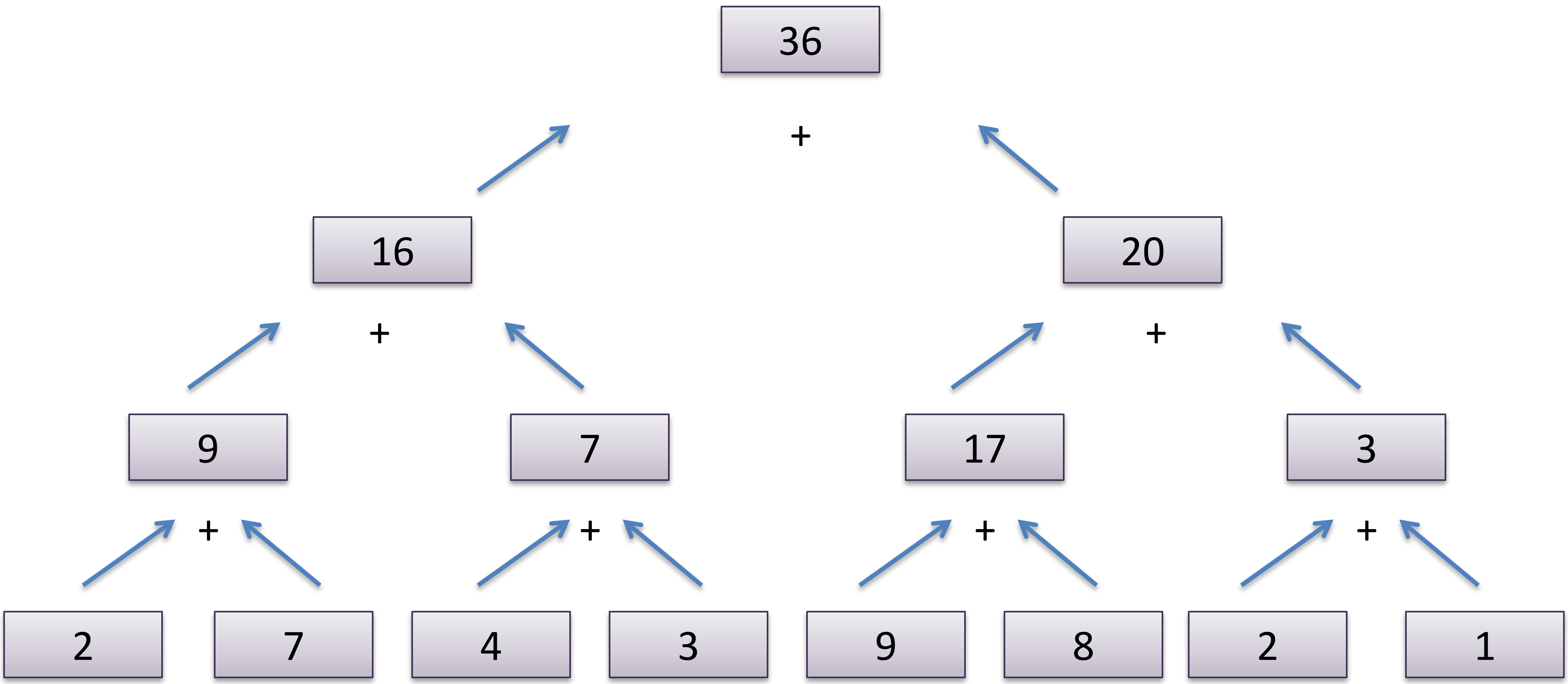
Computational Biology, DNA, and Mutation





INCREMENTAL COMPUTING IN OCAML

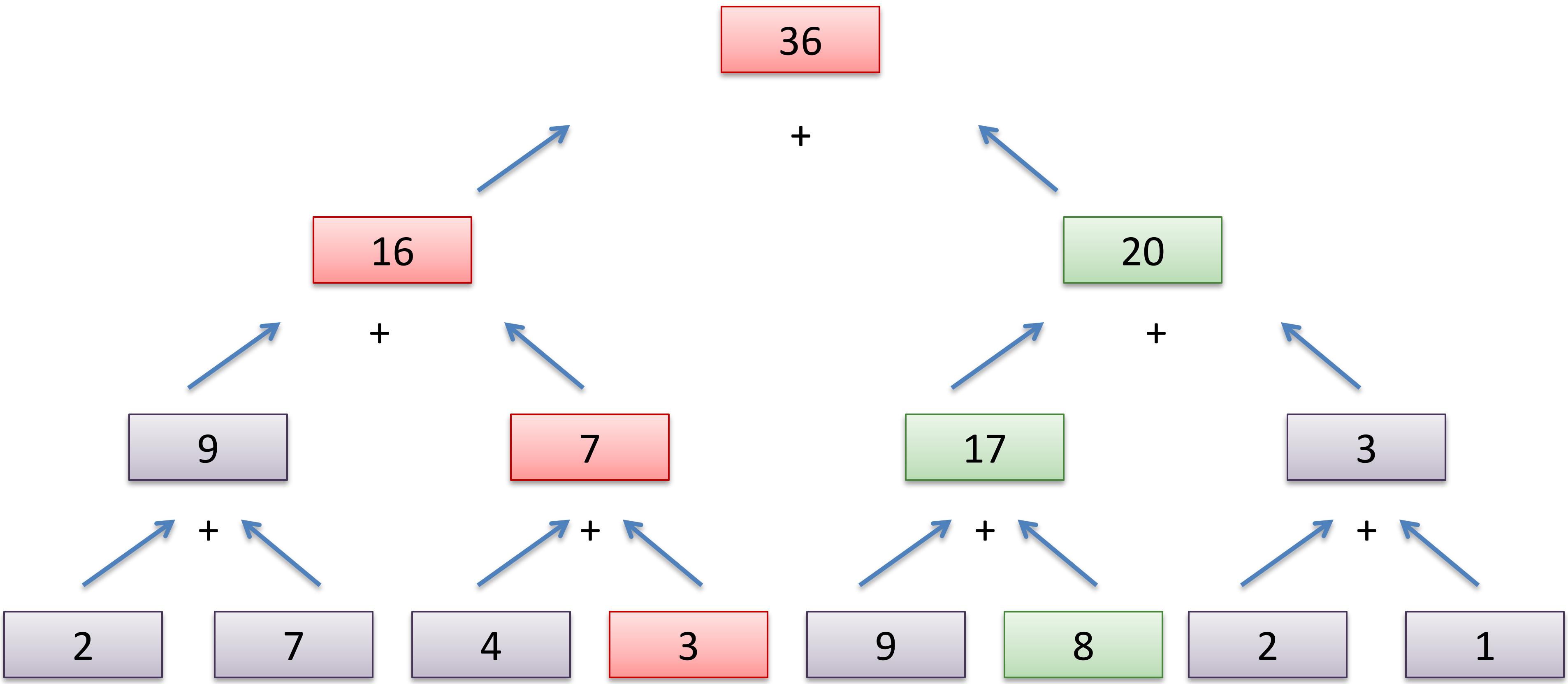
Efficient Parallel Computations



Work(n) = ~n additions to sum a vector of length n

Span(n) = ~log(n) additions – *the length of the longest dependency chain*

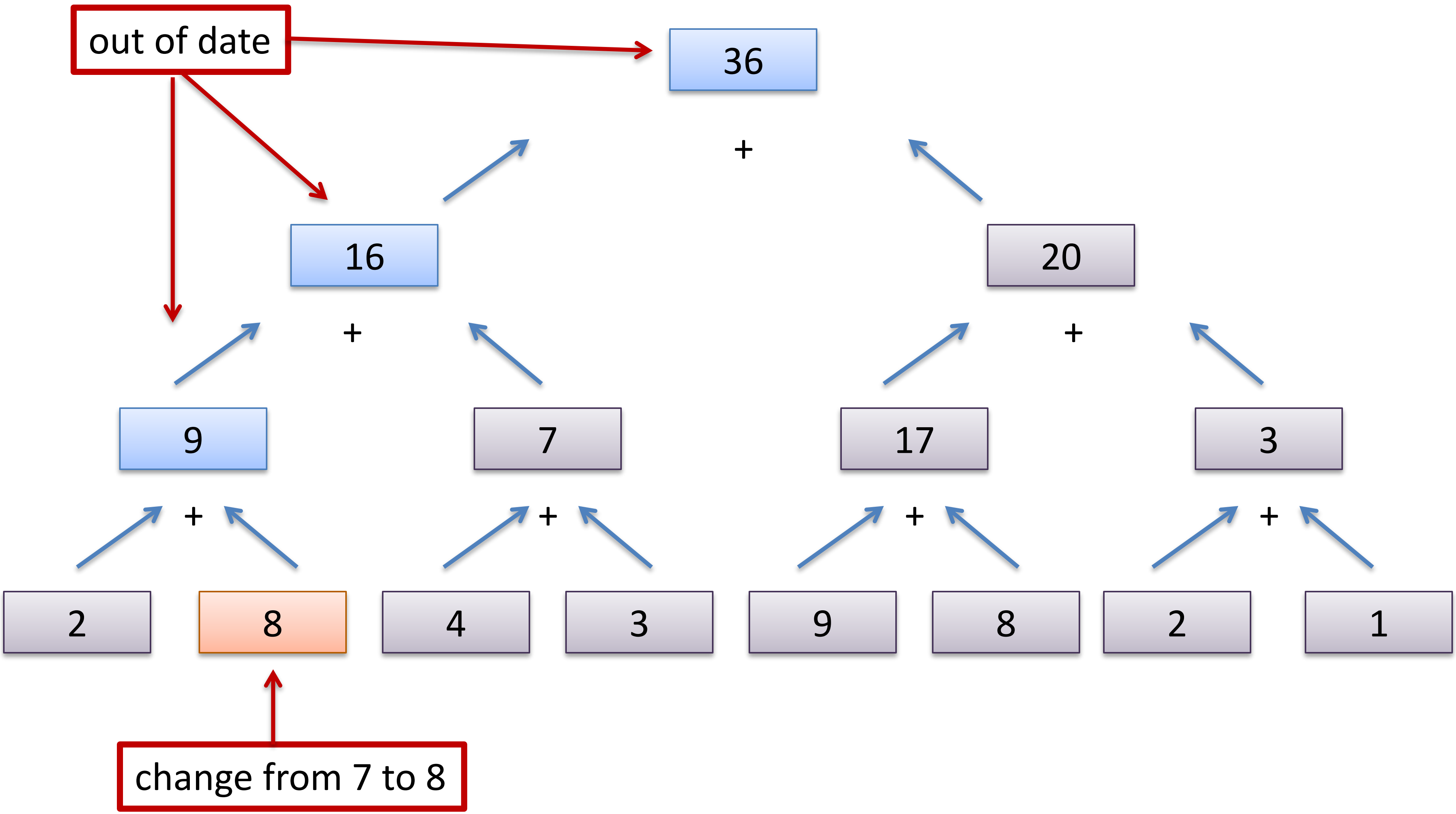
Efficient Parallel Computations



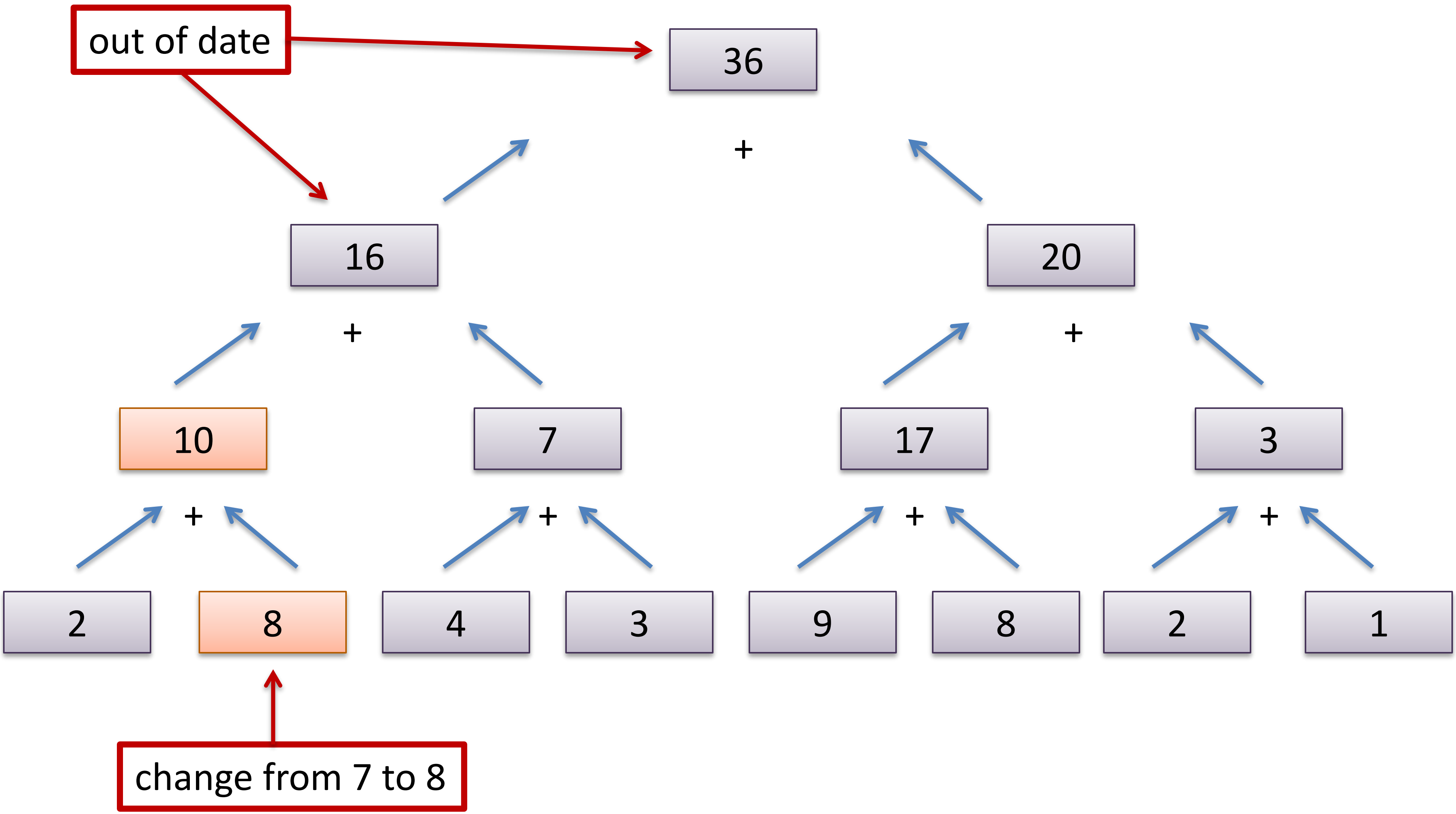
Work(n) = ~n additions to sum a vector of length n

Span(n) = ~log(n) additions – *the length of the longest dependency chain*

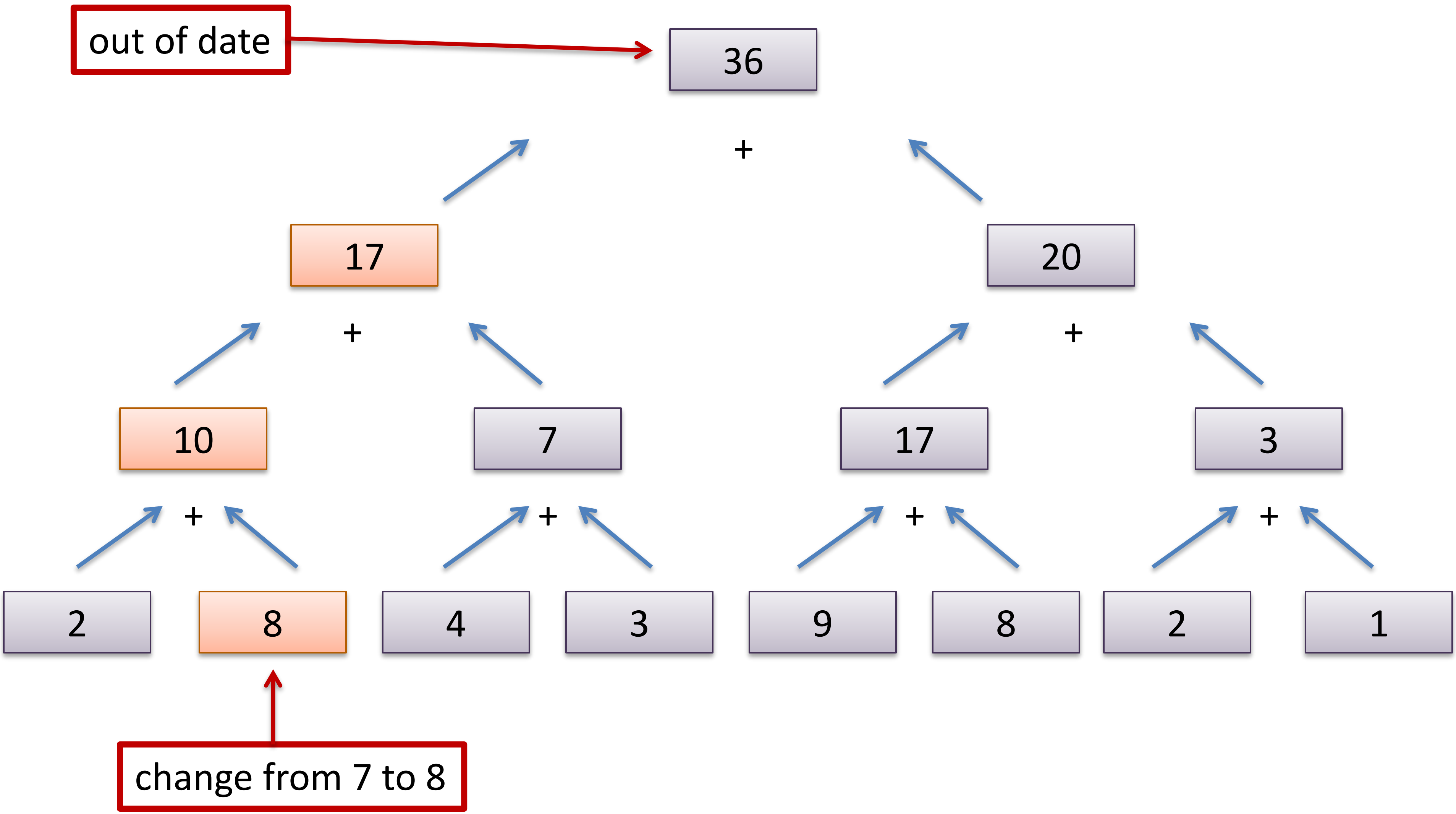
Efficient Incremental Computations



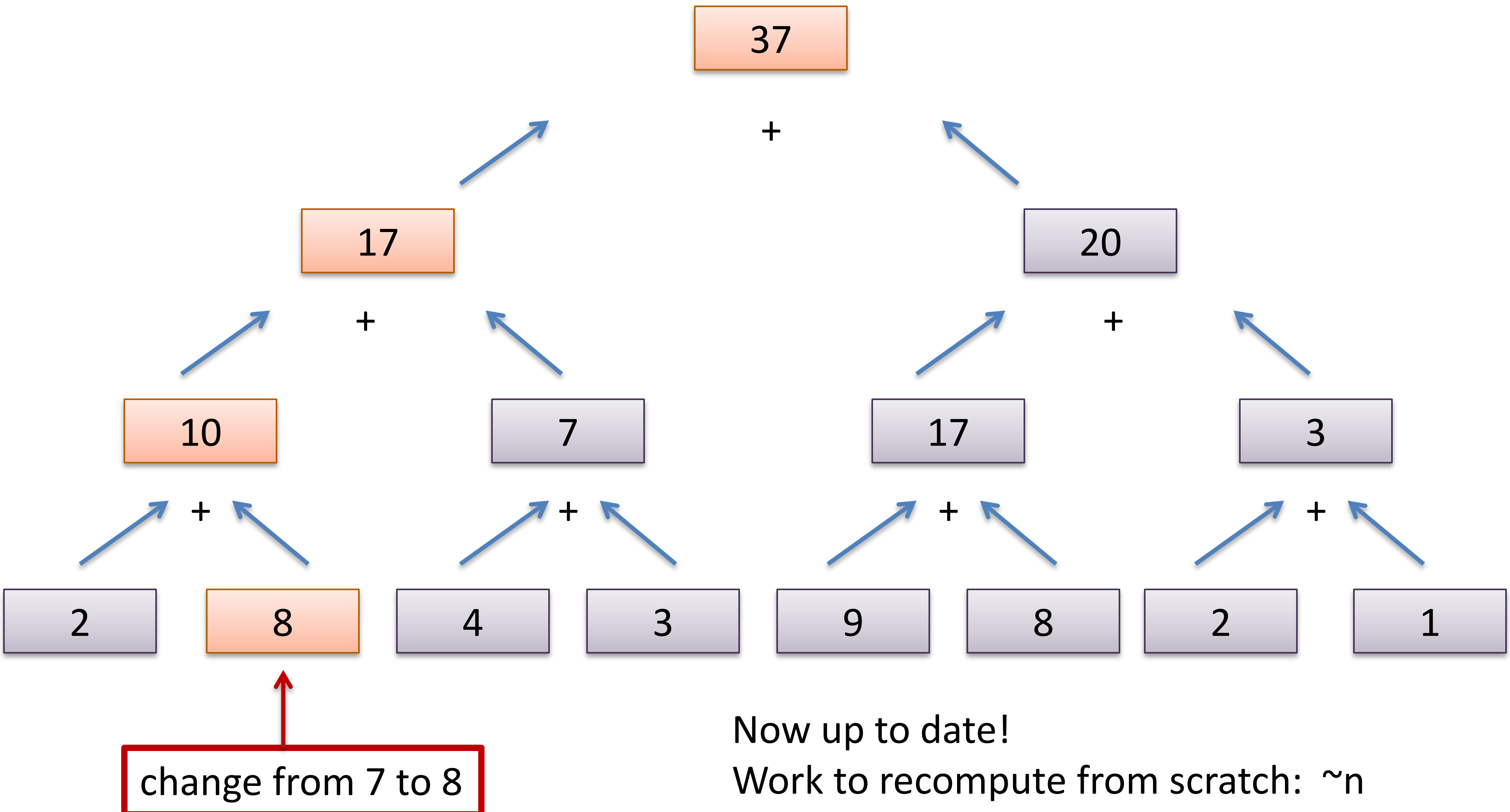
Efficient Incremental Computations



Efficient Incremental Computations



Efficient Incremental Computations



Now up to date!
Work to recompute from scratch: ~n
Work to recompute incrementally: ~log n

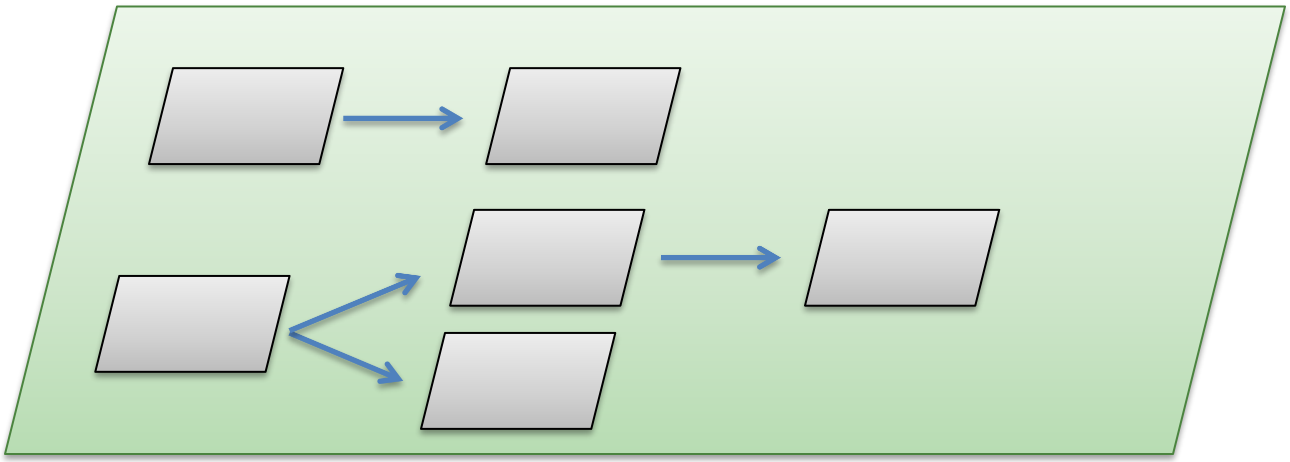
Parallel vs Incremental Computation

Similarity: span (ie: length of the longest dependency chain) of a computation governs latency

Difference: we will do a parallel computation *once*. We will do an incremental computation *many times*.

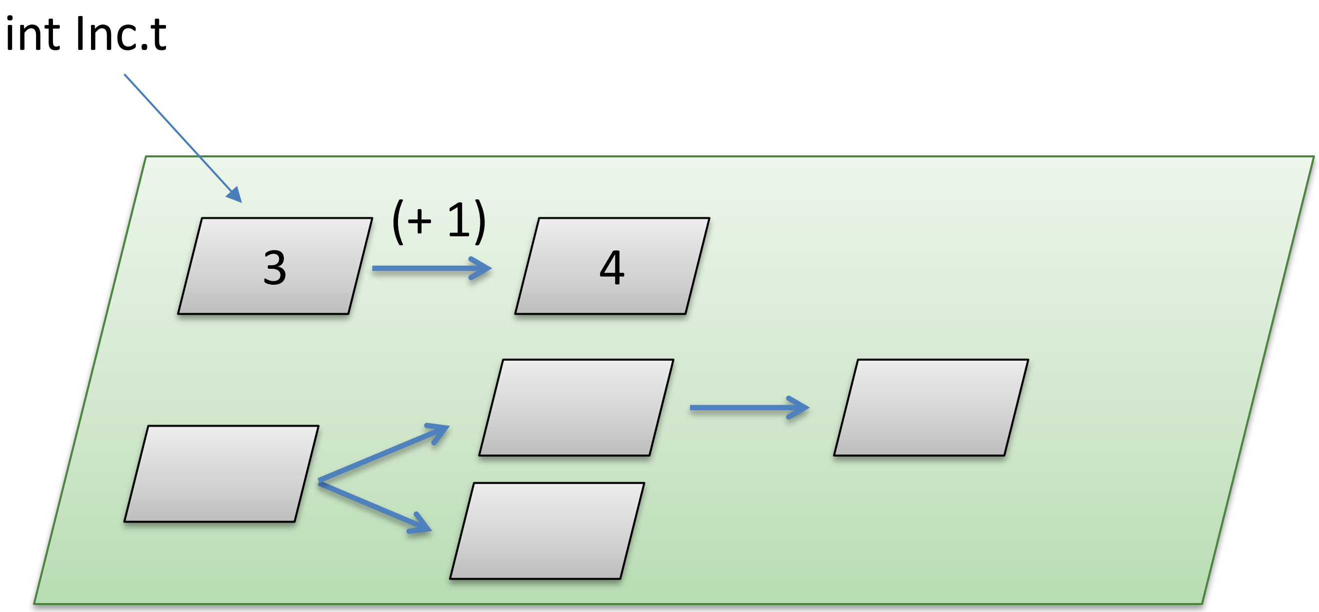
- the parallel dependency graph was *implicit*
  – represented the series of function calls made in order
- the incremental dependency graphs will be *explicit*
  – we will need to create a data structure that stores the computation graph so it can be reused

Incremental Dependency Graphs



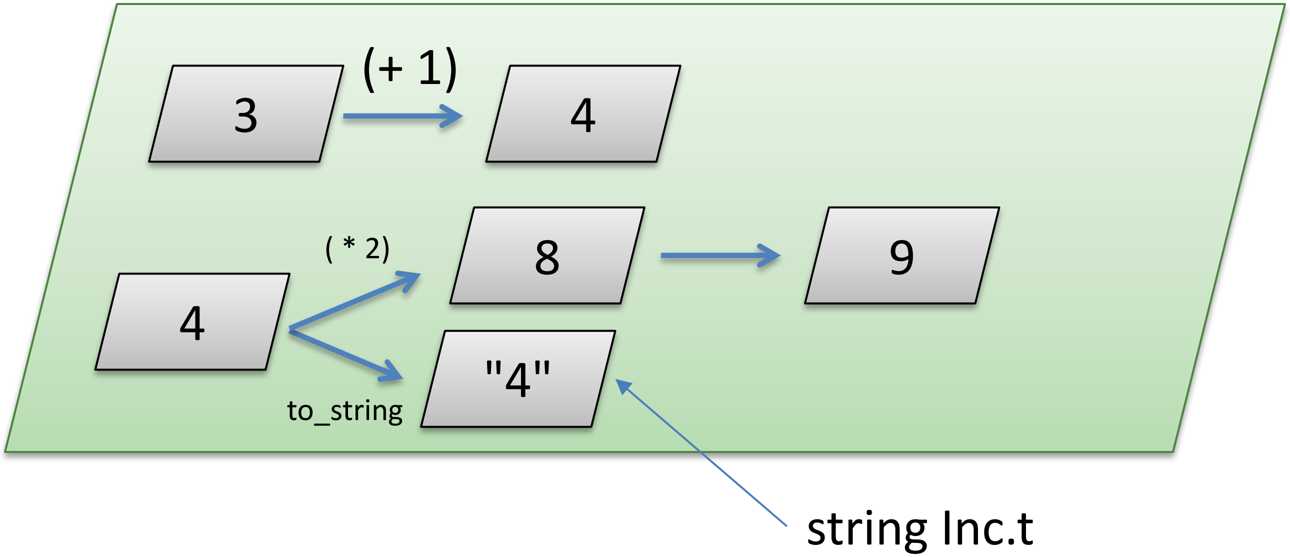
- **Nodes** have type 'a Inc.t
  - nodes store a current value with type 'a
- **Edges** are functions with type 'a -> 'b
  - if the argument 'a changes, the function recomputes 'b

Incremental Dependency Graphs



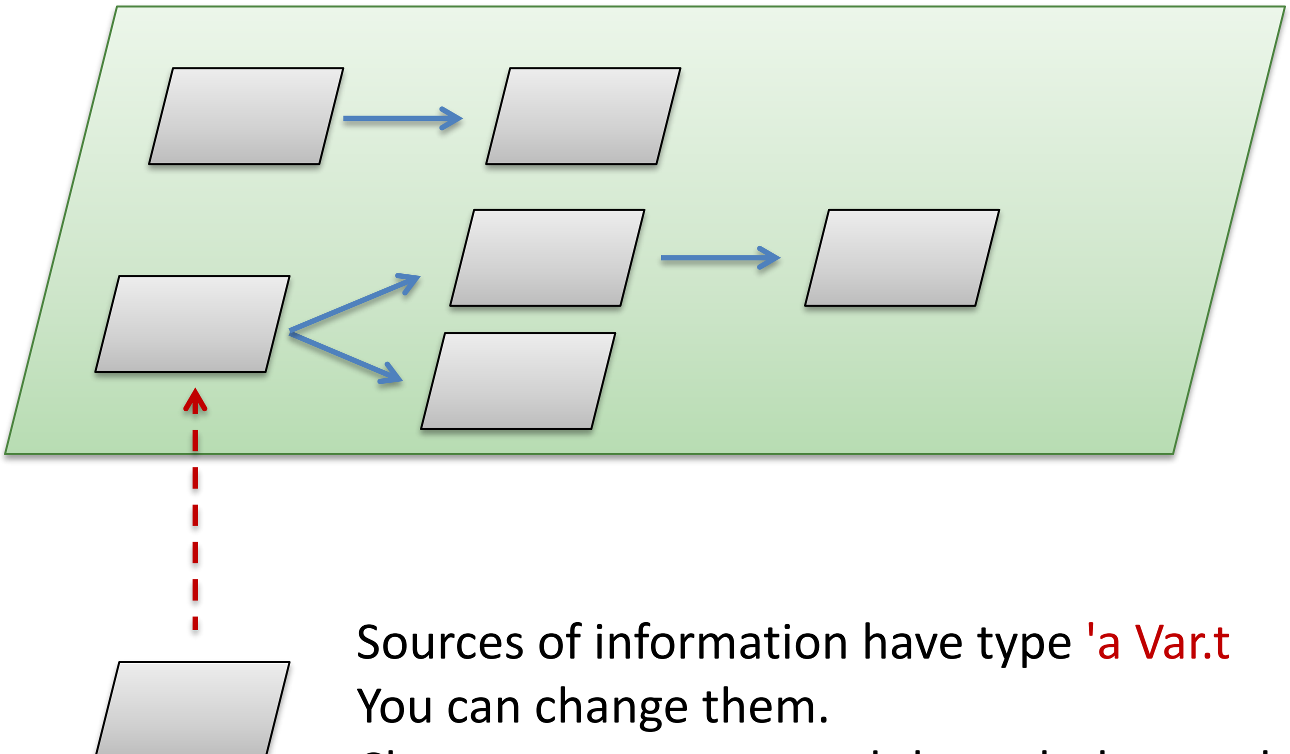
- **Nodes** have type 'a Inc.t
  - nodes store a current value with type 'a
- **Edges** are functions with type 'a -> 'b
  - if the argument 'a changes, the function recomputes 'b

Incremental Dependency Graphs



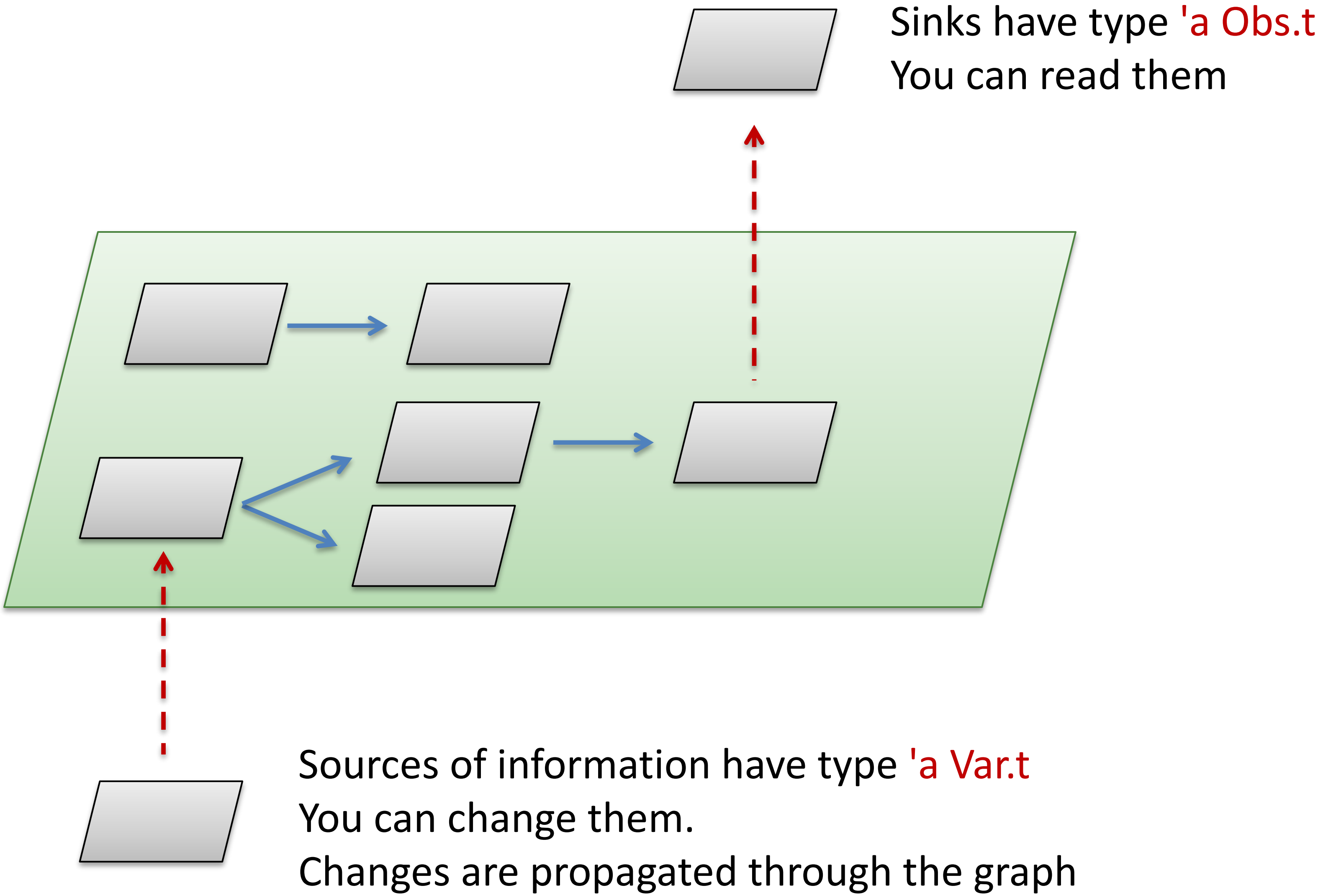
- **Nodes** have type 'a Inc.t
  - nodes store a current value with type 'a
- **Edges** are functions with type 'a -> 'b
  - if the argument 'a changes, the function recomputes 'b

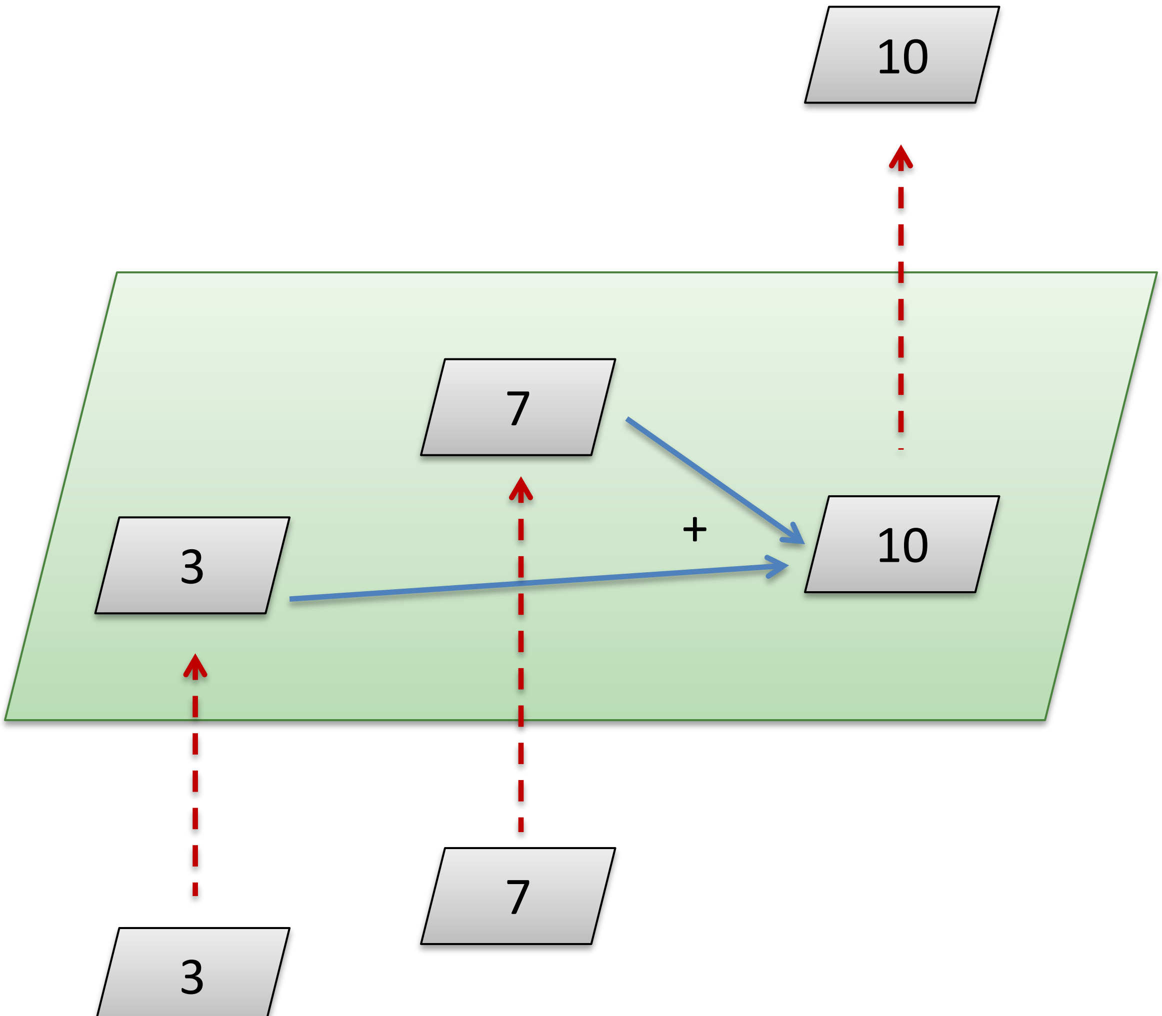
Accessing Incremental Dependency Graphs



Sources of information have type 'a Var.t
You can change them.
Changes are propagated through the graph

Accessing Incremental Dependency Graphs

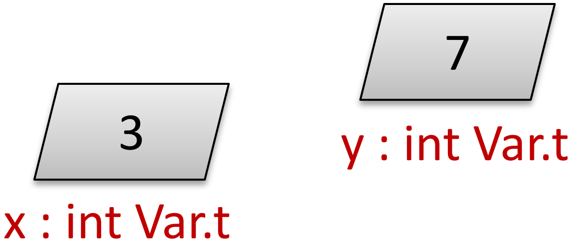





```
let x = Var.create 3 in
let y = Var.create 7 in
let z =
  Inc.map2
    (Var.watch x)
    (Var.watch y)
    ~f:(fun x y -> x + y) in
let z_o = Inc.observe z in
```

Building an Incremental Computation

1. Create *initial sources* with Var.create

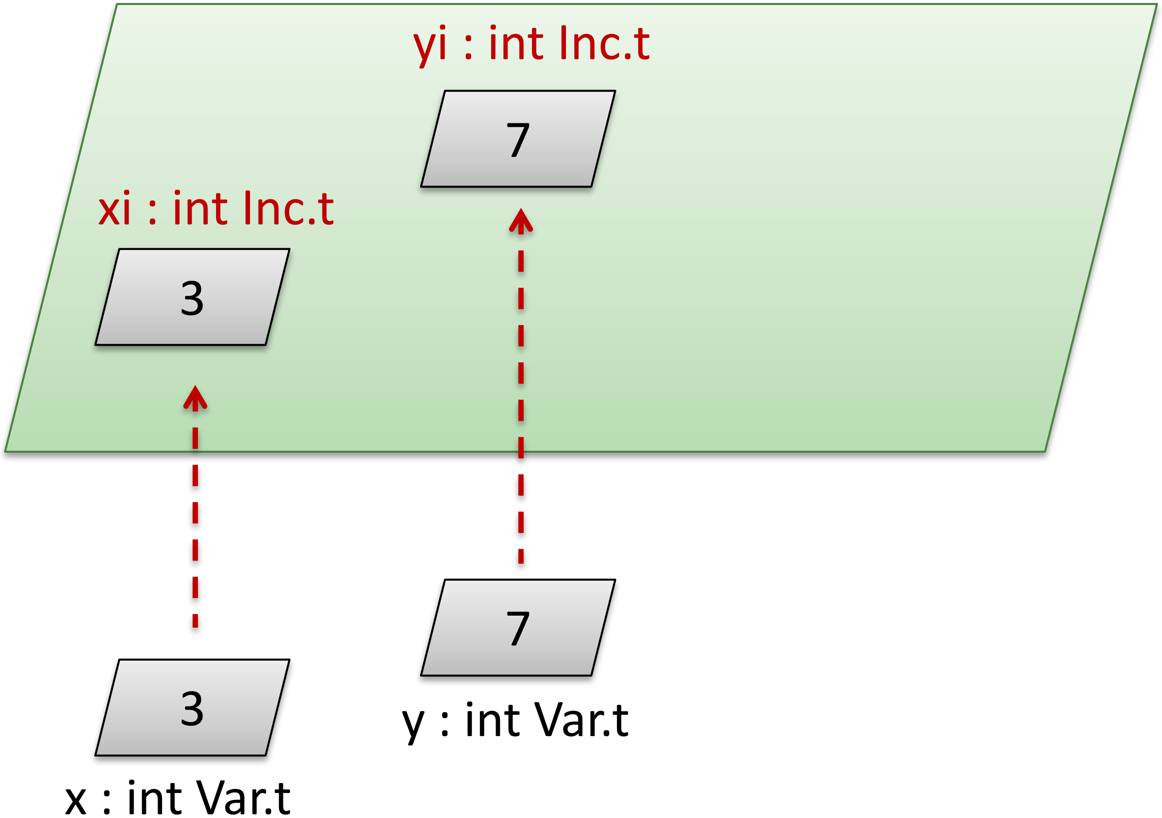



```
let x = Var.create 3 in
let y = Var.create 7 in
```

Building an Incremental Computation

2. Create incremental nodes by *watching* sources for change.

Var.watch : 'a Var.t -> 'a Inc.t

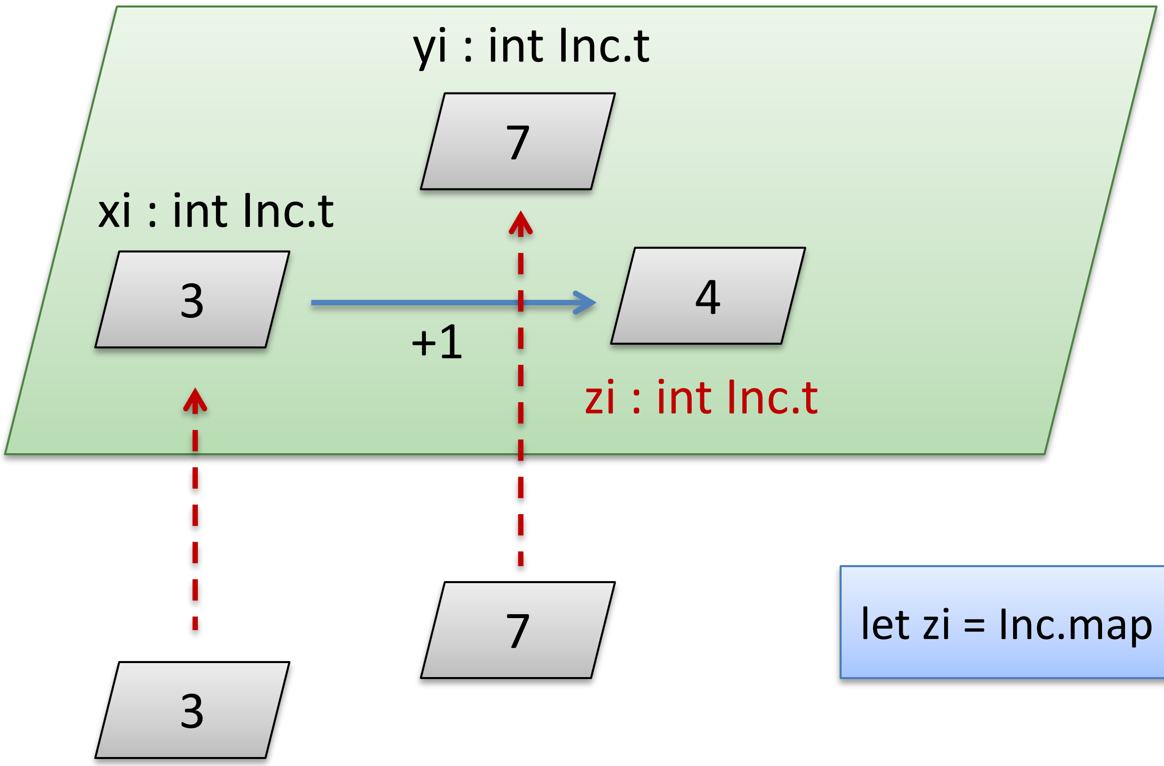



```
let xi = Var.watch x in
let yi = Var.watch y in
```

Building an Incremental Computation

3. *Create new incremental nodes* from existing incremental nodes by creating edges using map, map2, map3 ...

Inc.map : 'a Inc.t -> f:('a -> 'b) -> 'b Inc.t

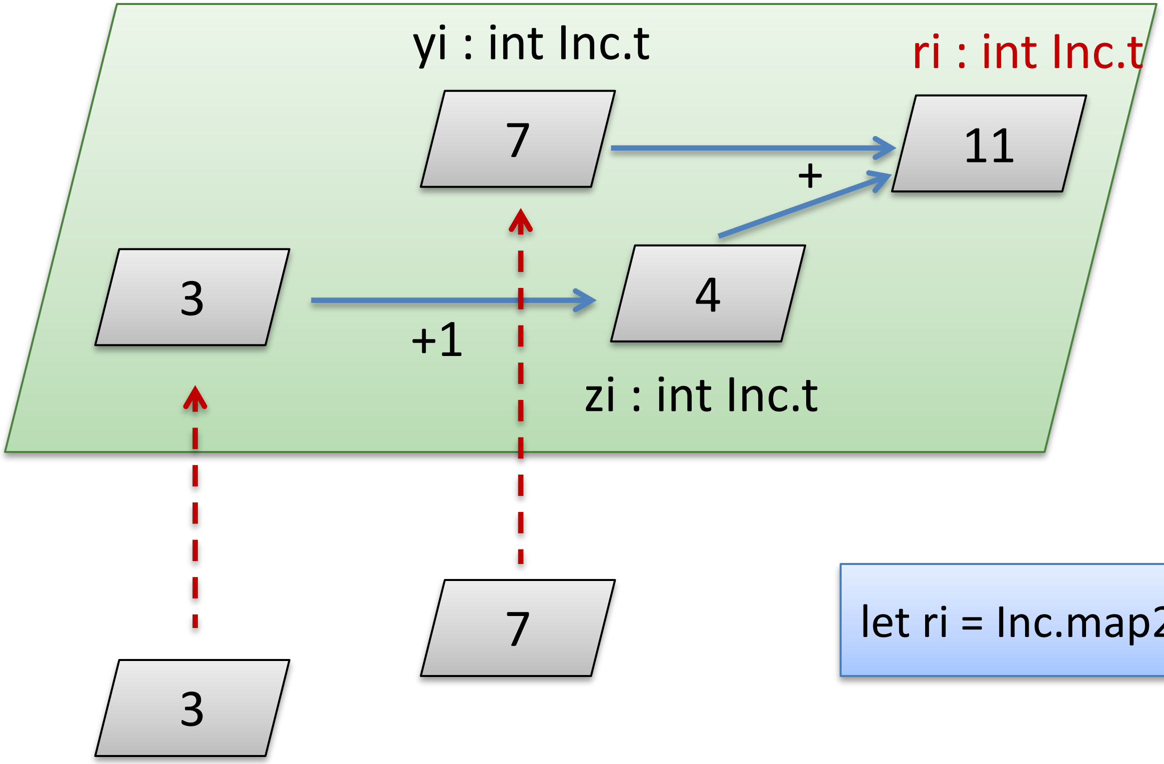



```
let zi = Inc.map  xi ~f:(fun x -> x + 1) in
```

Building an Incremental Computation

3. *Create new incremental nodes* from existing incremental nodes by creating edges using map, map2, map3 ...

Inc.map2 : 'a Inc.t -> 'b Inc.t -> f:('a -> 'b -> 'c) -> 'c Inc.t

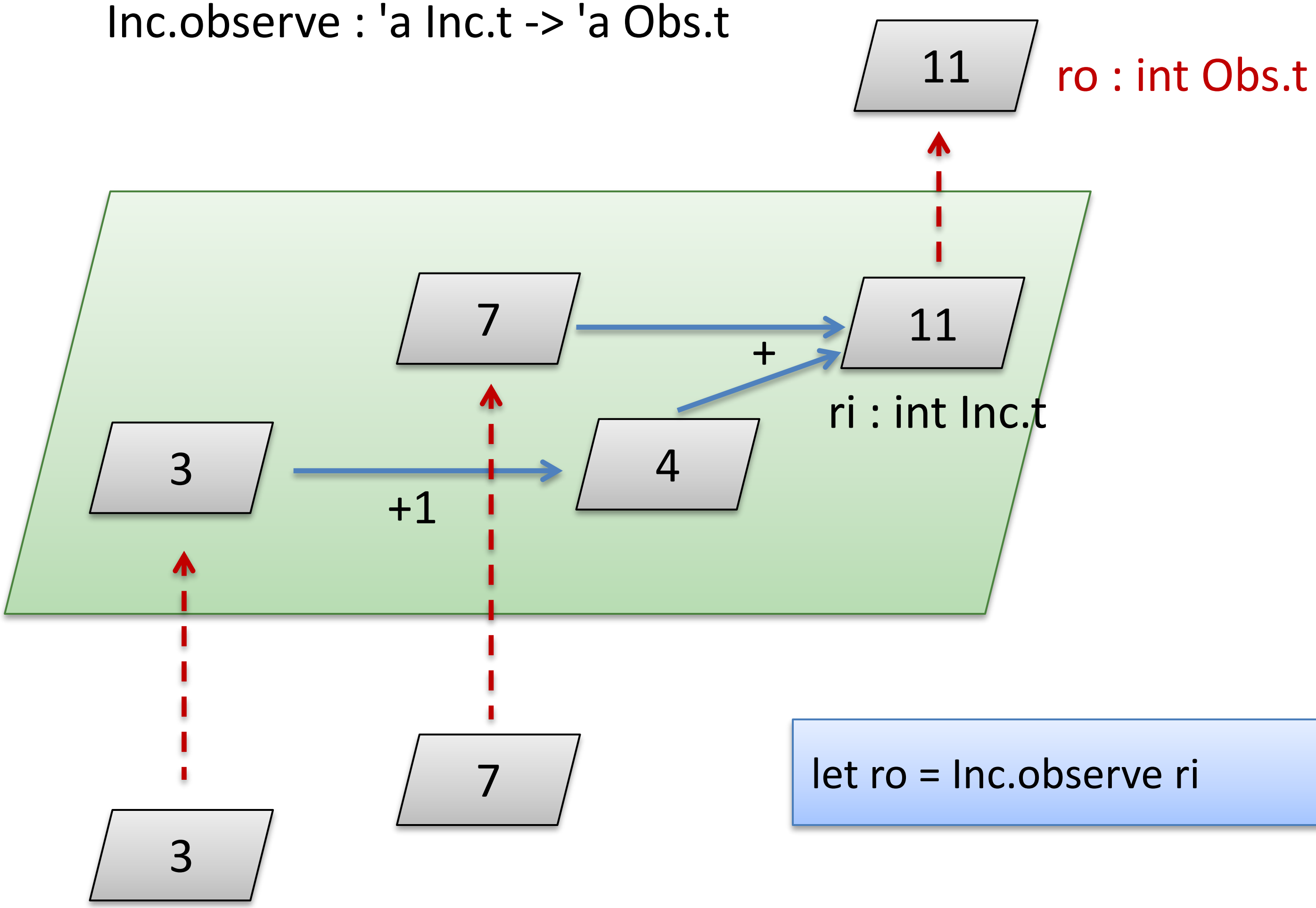



```
let ri = Inc.map2  zi yi ~f:(fun x y -> x + y) in
```

Building an Incremental Computation

4. Extract *observable* results from graph

Inc.observe : 'a Inc.t -> 'a Obs.t

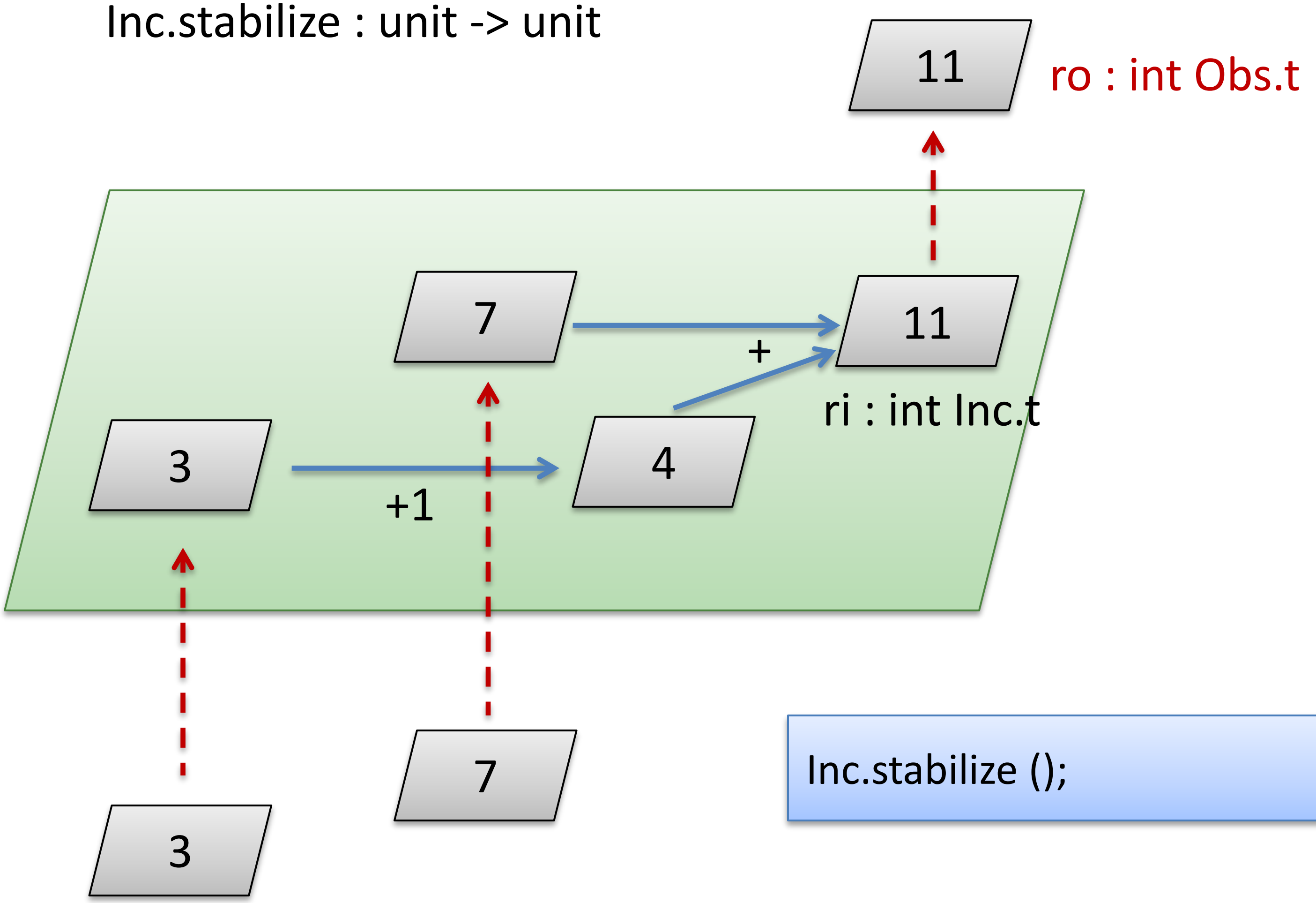


let ro = Inc.observe ri

Building an Incremental Computation

5. *Stabilize* (ie: push any pending changes through the graph)

Inc.stabilize : unit -> unit

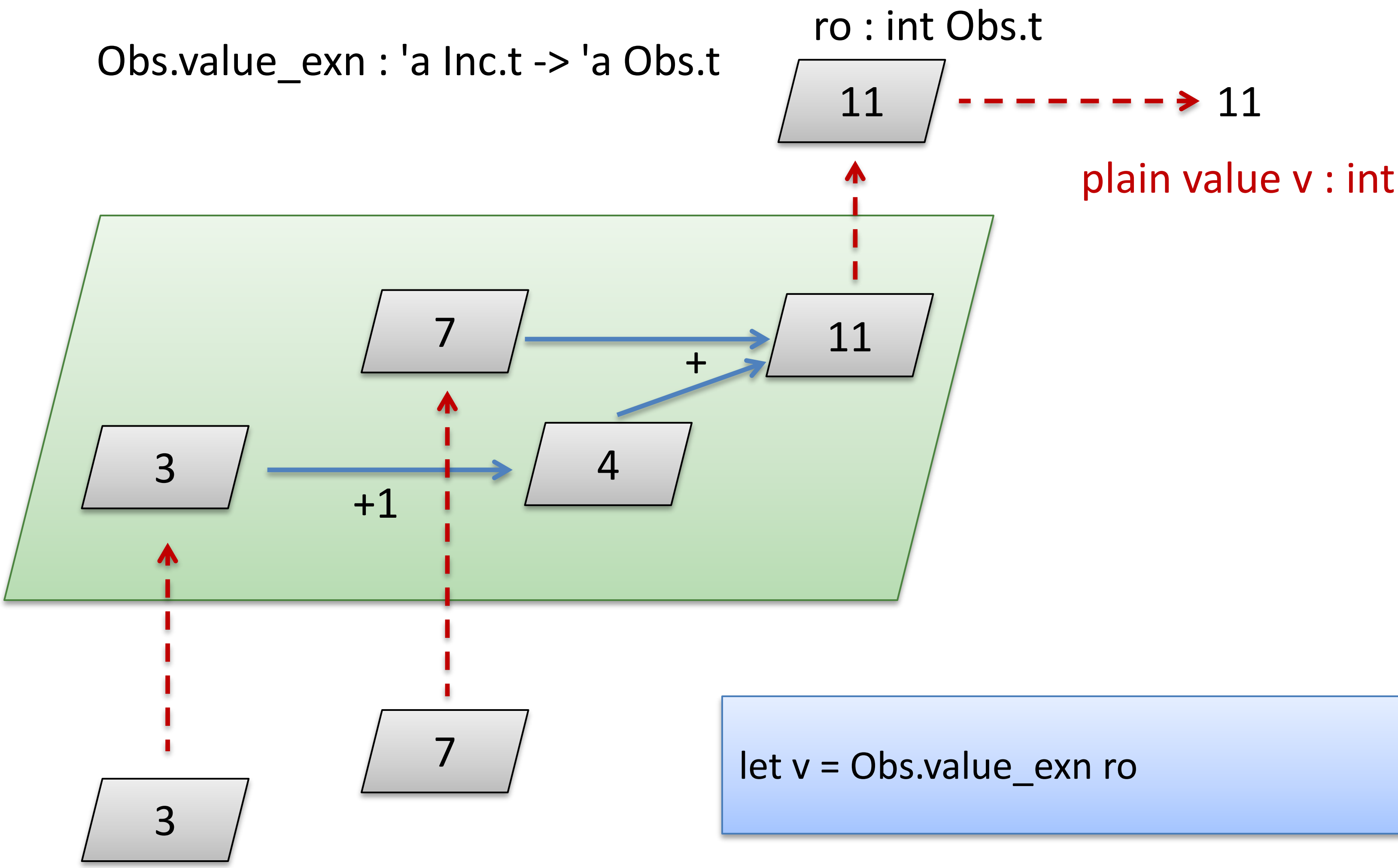



```
Inc.stabilize ();
```

Building an Incremental Computation

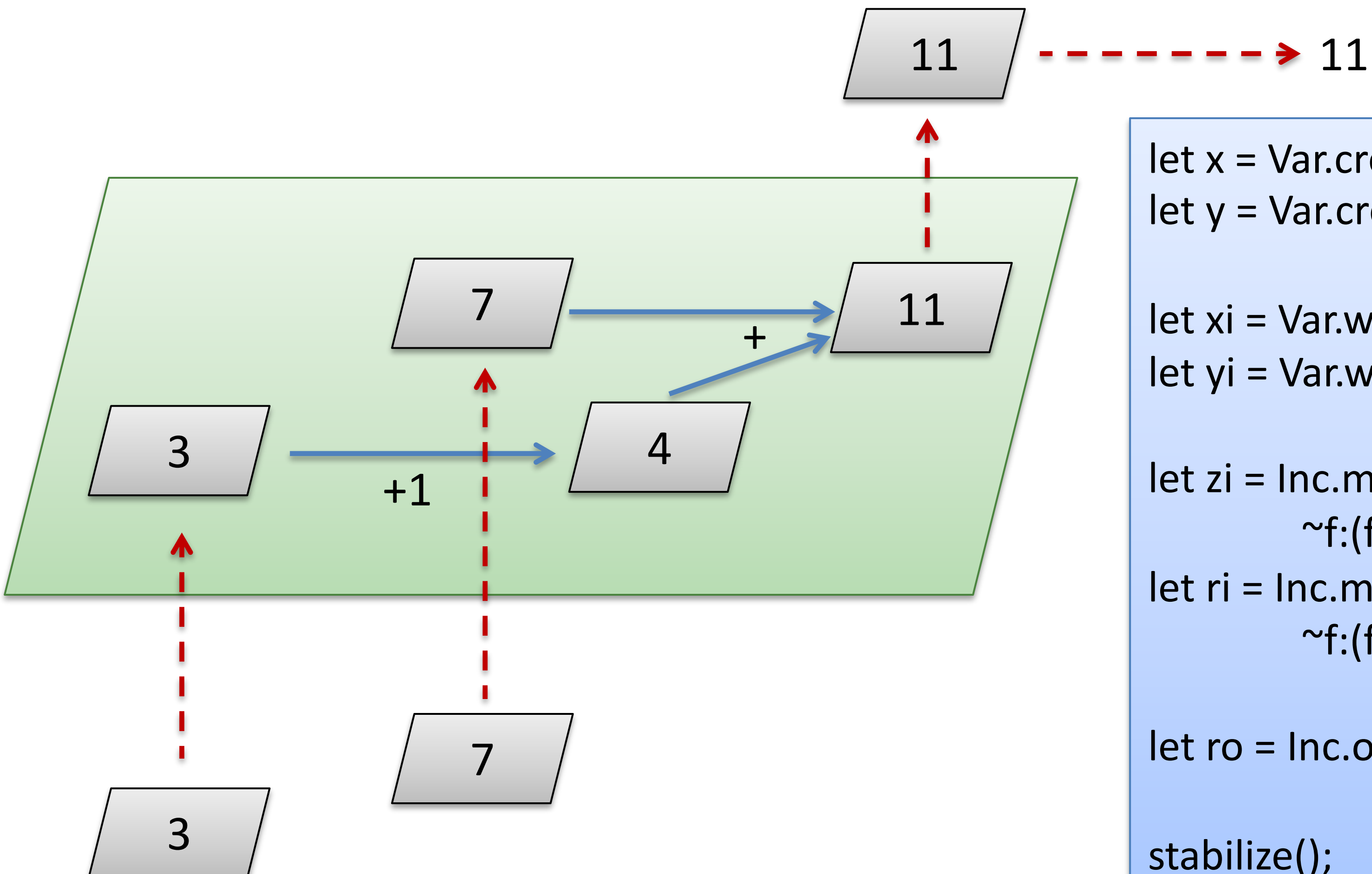
6. Get *plain value* from observable after stabilizing.

Obs.value_exn : 'a Inc.t -> 'a Obs.t




```
let v = Obs.value_exn ro
```

Building an Incremental Computation



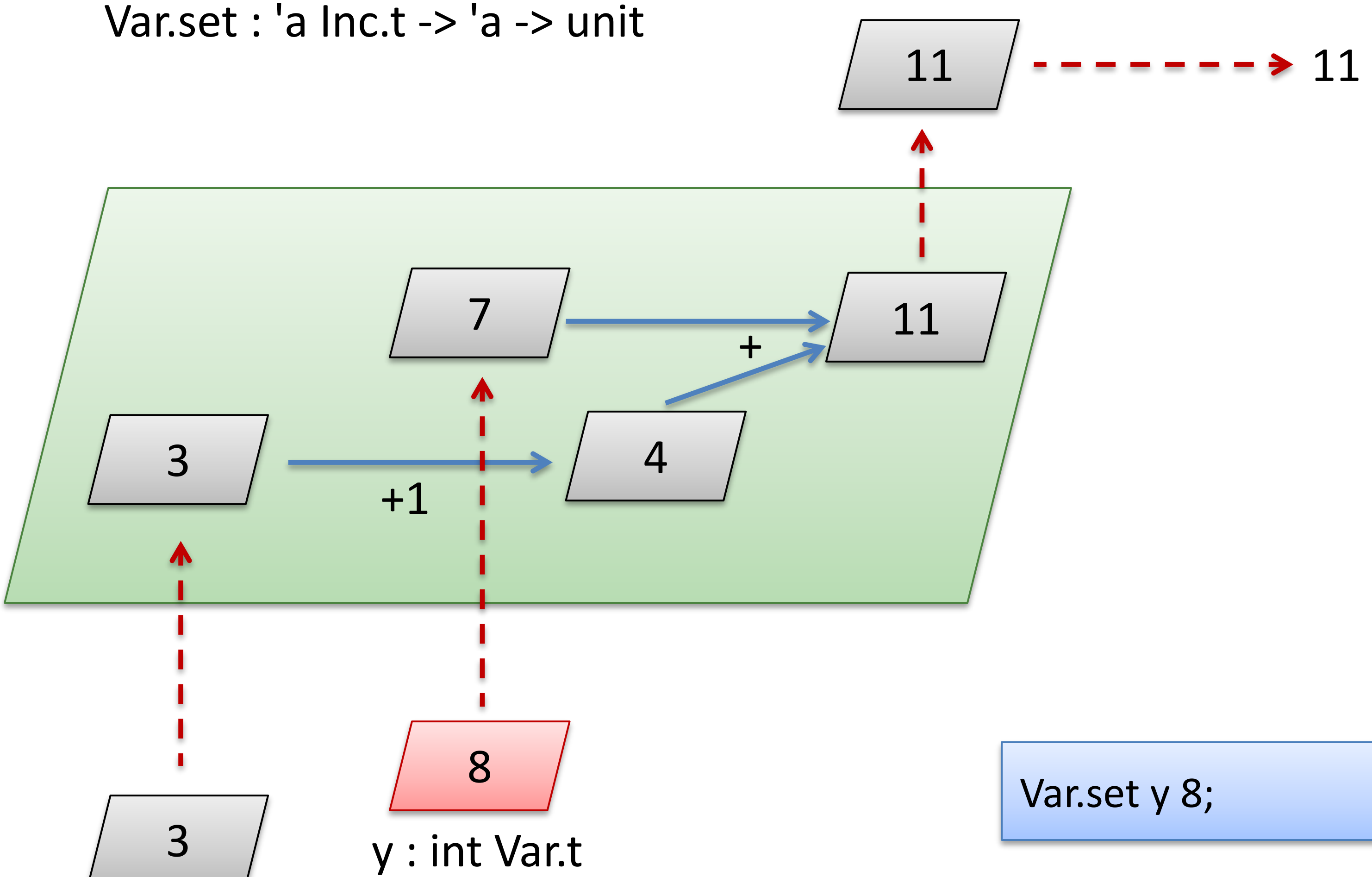
Summary

```
let x = Var.create 3 in
let y = Var.create 7 in

let xi = Var.watch x in
let yi = Var.watch y in

let zi = Inc.map  xi
           ~f:(fun x -> x + 1) in
let ri = Inc.map2 zi yi
           ~f:(fun x y -> x + y) in

let ro = Inc.observe ri in

stabilize();
let v = Obs.value_exn ro in
```

Building an Incremental Computation

7. *Update* source variables.

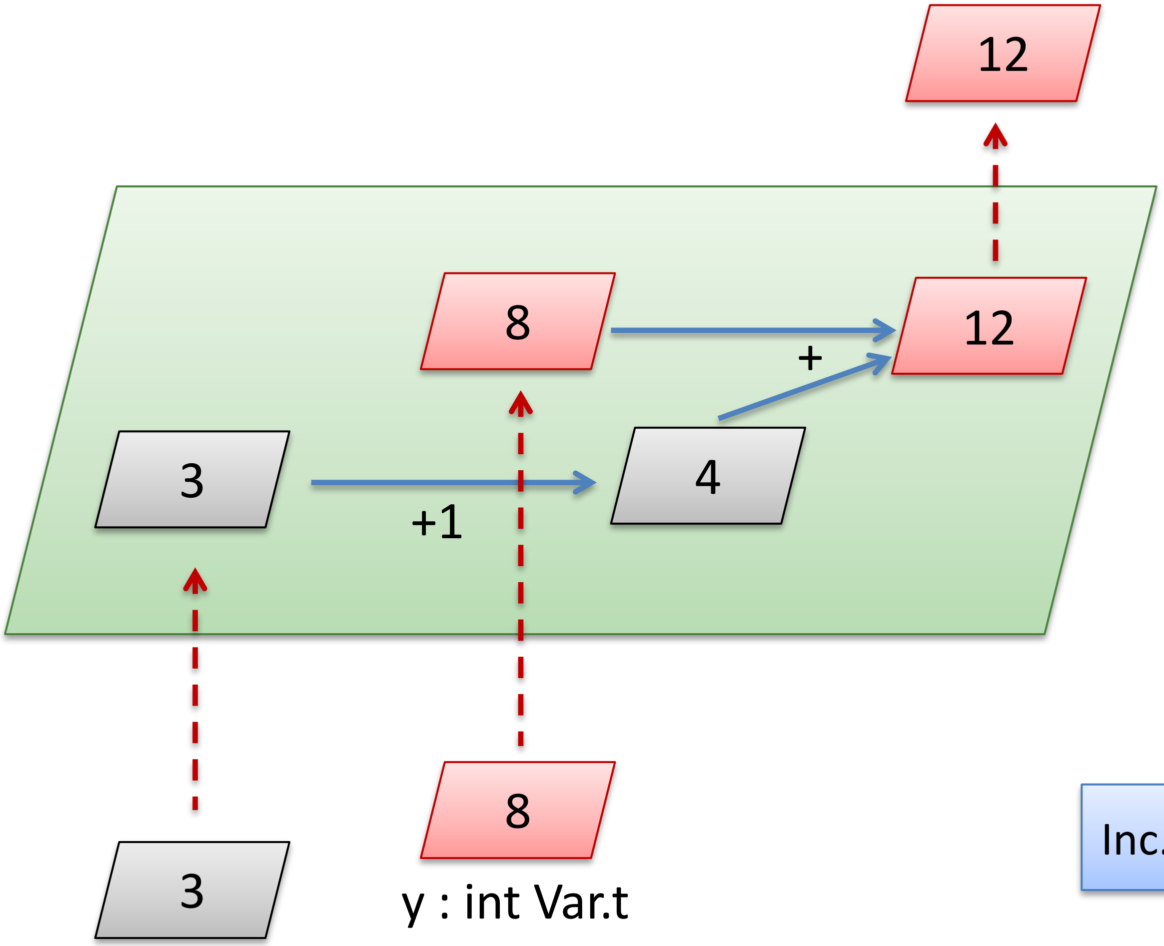
Var.set : 'a Inc.t -> 'a -> unit




```
Var.set y 8;
```

Building an Incremental Computation

7. *Stabilize* again

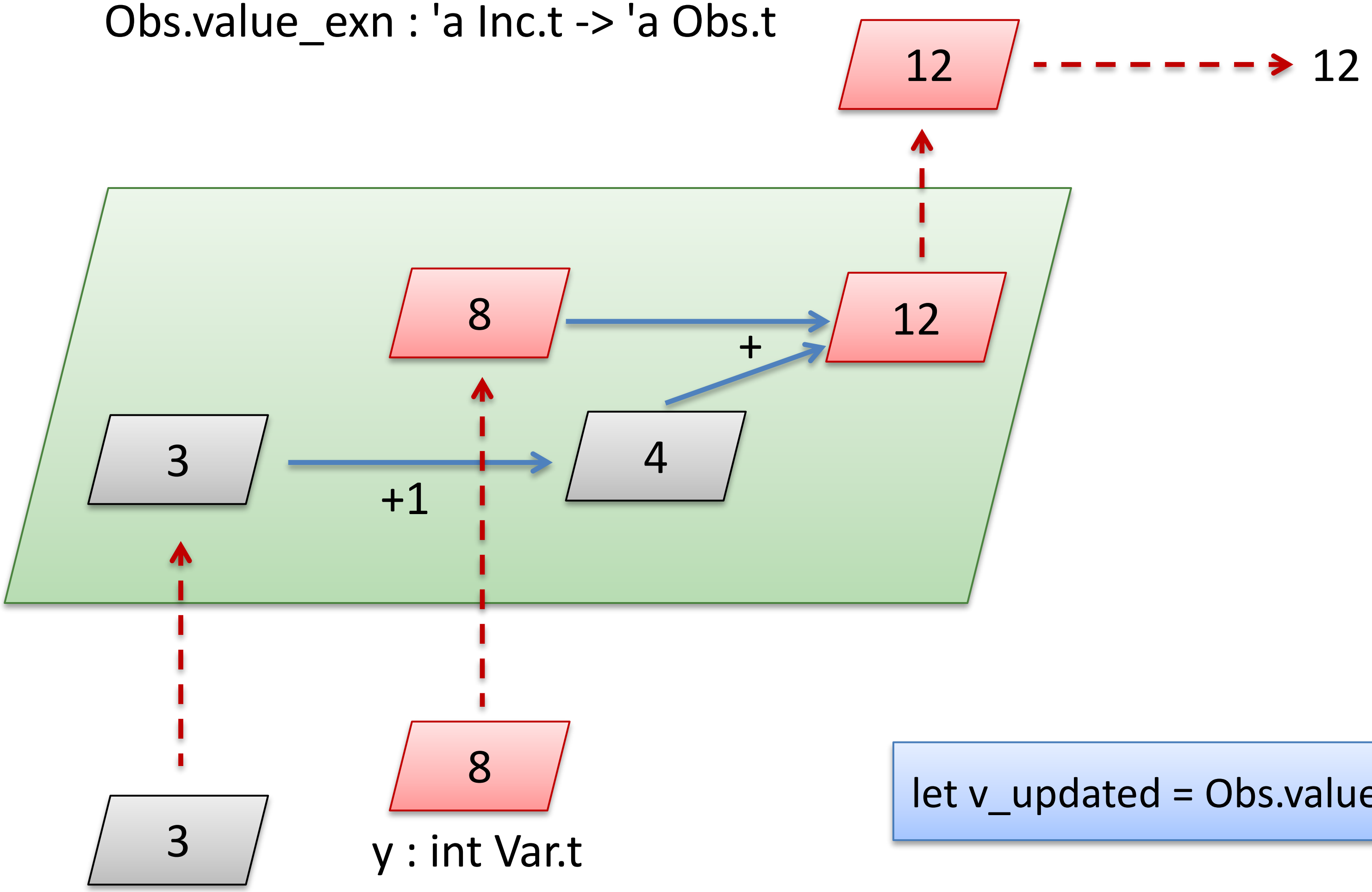


Inc.stabilize ();

Building an Incremental Computation

8. Get *plain value* from observable after stabilizing.

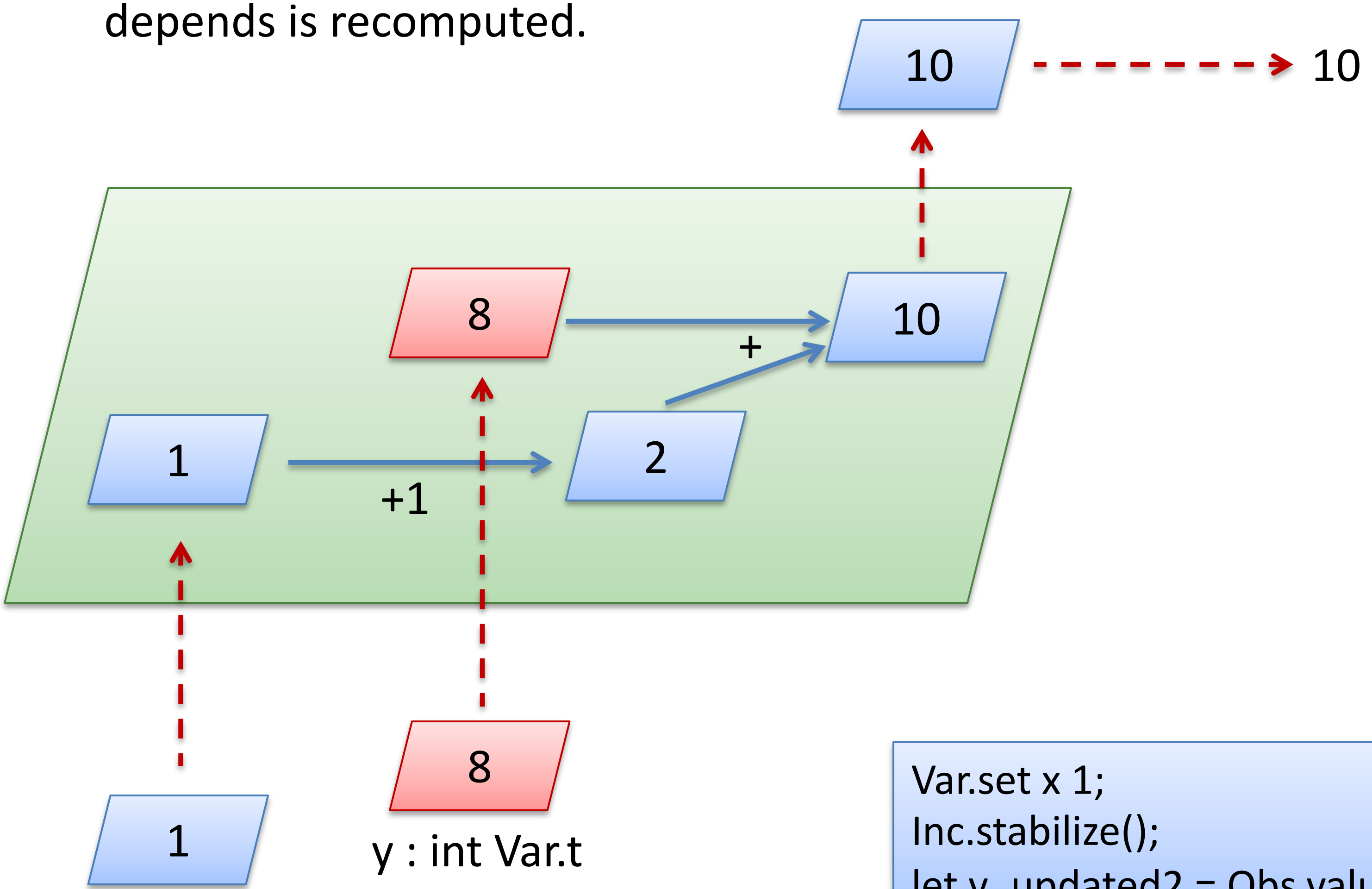
Obs.value_exn : 'a Inc.t -> 'a Obs.t



let v_updated = Obs.value_exn ro in

Building an Incremental Computation

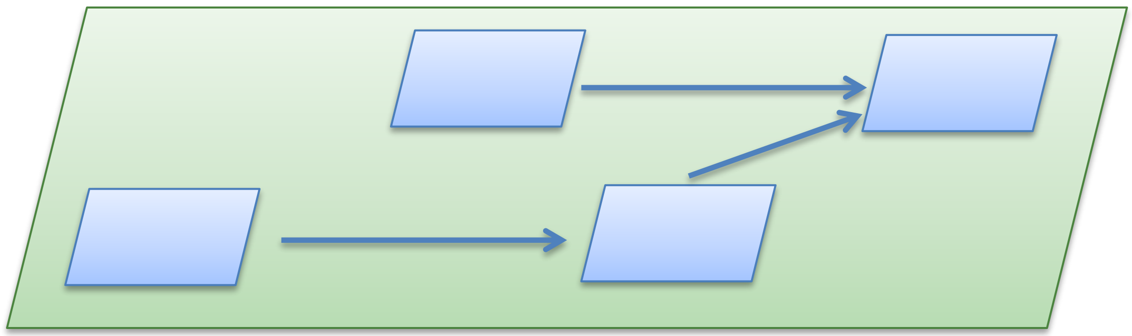
9. *Repeat*: *Set var --> Stabilize --> Get observed value*
   Each time, the subgraph that changed and on which the answer depends is recomputed.



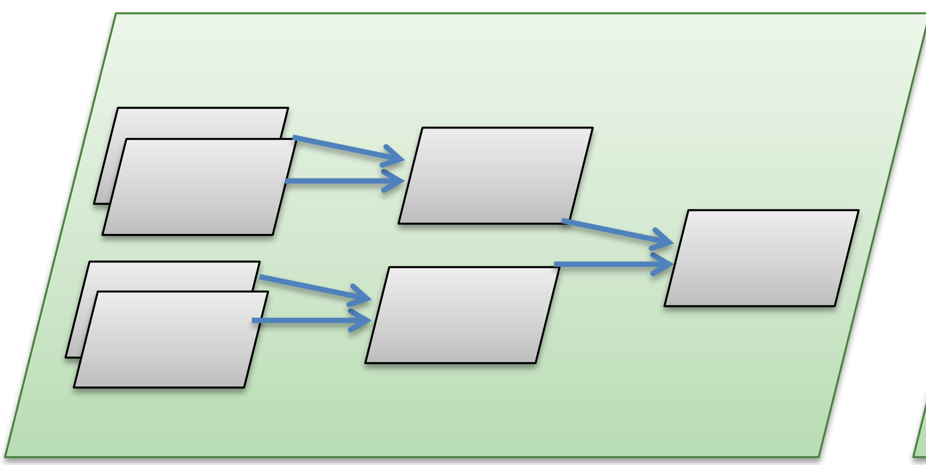

```
Var.set x 1;
Inc.stabilize();
let v_updated2 = Obs.value_exn ro in
```

Structured Graphs

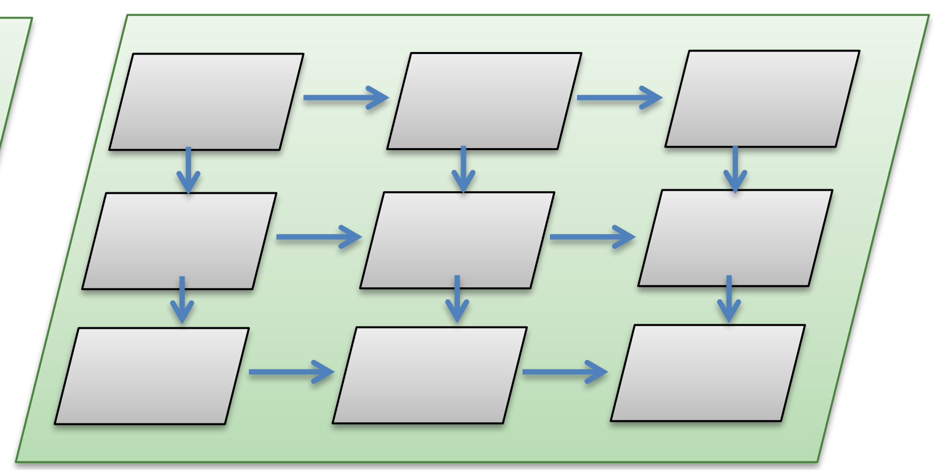
So Far: Unstructured, *ad hoc* graphs



Next: Structured graphs



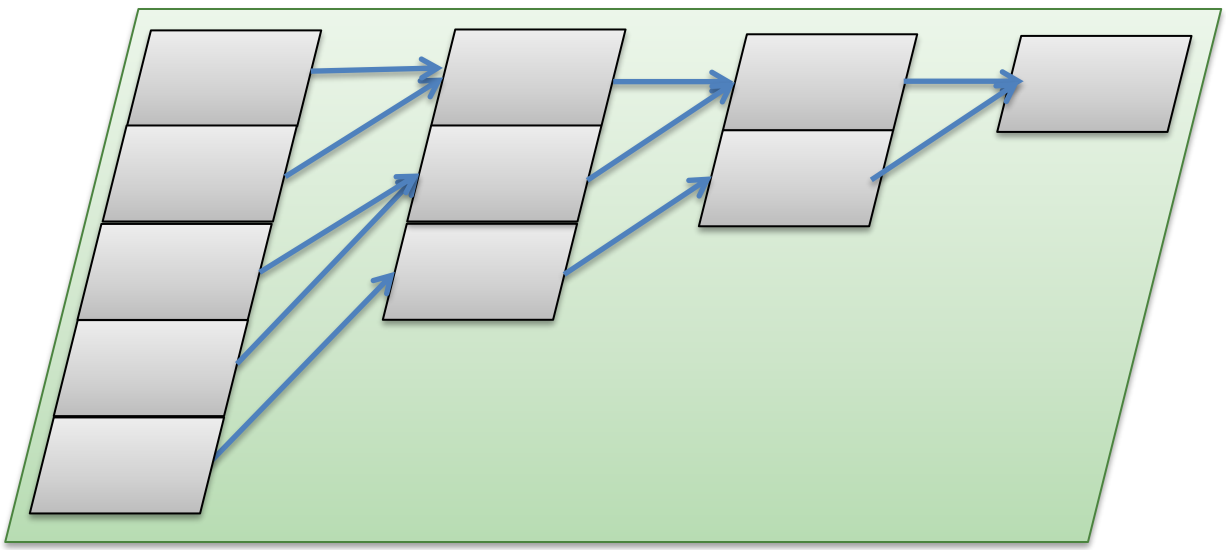
trees



tables/spread sheets

Structured Graphs

Implement a reduce of function f over a sequence.



Iteration: 0 1 2 3

Let **prev** be the array created in previous iteration
Each cell i of the **curr**ent array will be defined as follows:

$$\begin{cases} \mathbf{curr}[i] = \mathbf{f}\ (\mathbf{prev}[2*i])\ (\mathbf{prev}[2*i+1]) & \text{if prev}[2*i+1] \text{ exists} \\ \mathbf{curr}[i] = \mathbf{prev}[2*i] & \text{otherwise} \end{cases}$$

Structured Graphs

Let **prev** be the array created in previous iteration
Each cell i of the **curr**ent array will be defined as follows:

$$\begin{cases} \textbf{curr}[i] = \textbf{f}\ (\textbf{prev}[2*i])\ (\textbf{prev}[2*i+1]) & \text{if prev}[2*i+1] \text{ exists} \\ \textbf{curr}[i] = \textbf{prev}[2*i] & \text{otherwise} \end{cases}$$

Standard Functional Algorithm:

```
let rec merge (prev: 'a array) (f:'a -> 'a -> 'a) : 'a =
  if Array.length prev <= 1 then prev.(0)
  else
    let len = Array.length prev in
    let len' = (len/2) + (len mod 2) in
    let cell i =
      if i * 2 + 1 < len then f prev.(2*i) prev.(2*i+1)
                             else prev.(2*i)
    in
    let curr = Array.init len' cell in
    merge curr f
```

compute new cell value from previous cell values

prev array of values thrown away after its one use

Structured Graphs

Let **prev** be the array created in previous iteration
Each cell i of the **curr**ent array will be defined as follows:

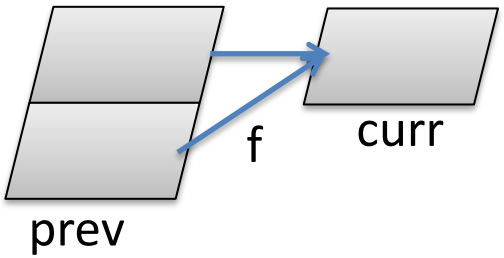
$$
\begin{cases}
\textbf{curr}[i] = \textbf{f}\ (\textbf{prev}[2*i])\ (\textbf{prev}[2*i+1]) & \text{if prev}[2*i+1] \text{ exists} \\
\textbf{curr}[i] = \textbf{prev}[2*i] & \text{otherwise}
\end{cases}
$$

Standard Functional Algorithm:

```
let rec merge (prev: 'a Inc.t array) (f:'a -> 'a -> 'a) : 'a =
  if Array.length prev <= 1 then prev.(0)
  else
    let len = Array.length prev in
    let len' = (len/2) + (len mod 2) in
    let cell i =
      if i * 2 + 1 < len then Inc.map2 prev.(2*i) prev.(2*i+1) ~f:f
                              else prev.(2*i)
    in
    let curr = Array.init len' cell in
    merge curr f
```

pass in array of incrementals

create incremental graph:



Moral of the Story

Functional algorithms are easily transformed into incremental functional algorithms.

<u>Stack of Functional Recursive Calls:</u>

f x1

↑

calls f x2

↑

calls f x3

<u>Build Incremental graph Calls:</u>

f x1

↑

calls Inc.map x2 ~f

↑

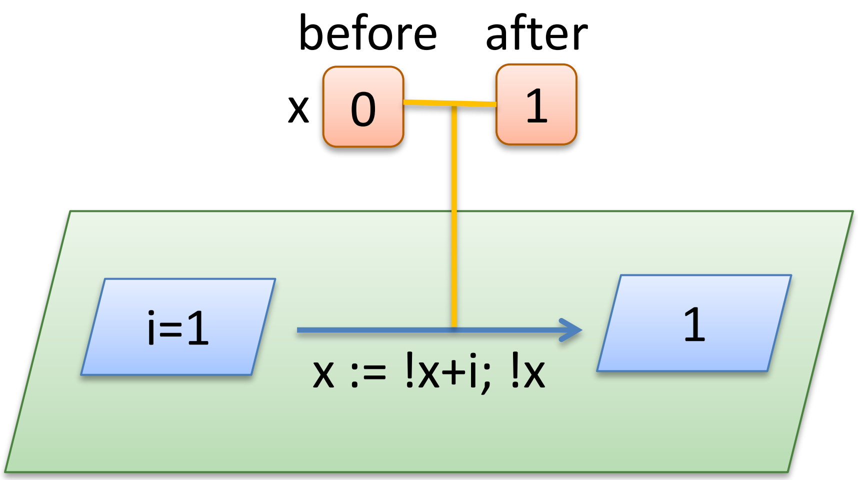
calls Inc.map x3 ~f

1. convert argument from 'a to 'a Inc.t
2. convert result computed from 'b to 'b Inc.t by using Inc.map
3. fix up initial call to supply 'a Inc.t rather than 'a (use Var.create, Var.watch)
   fix up result returned to extract 'a from 'a Inc.t (use Inc.observe, Obs.value_exn)

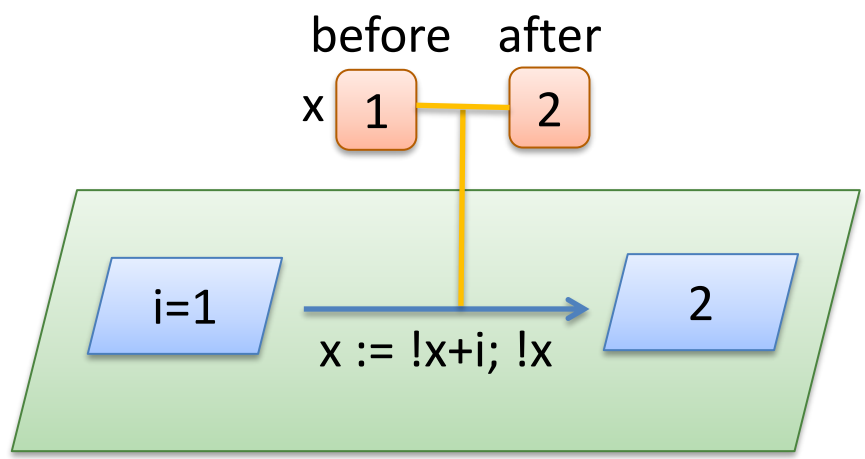
Mutation

What happens if your algorithm is not function? Uses mutable references?

Issue 1: The output is immediately "out of date"



original run



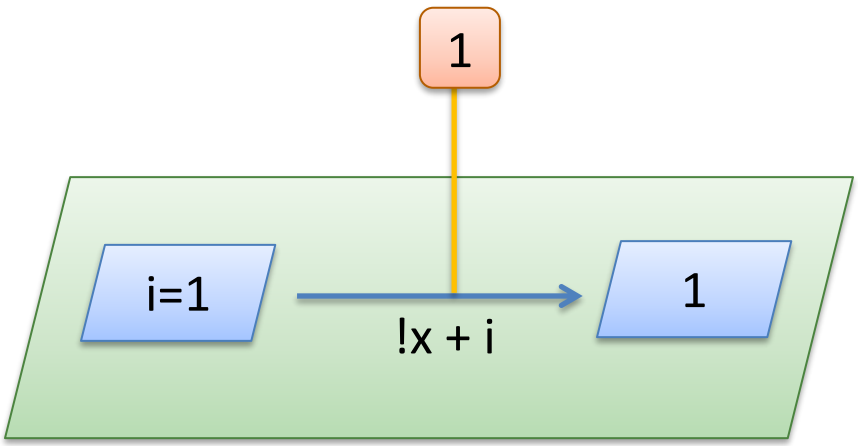
If you run it again,
you get a different answer

Very difficult to reason about (and draw!)
Avoid at almost all costs.

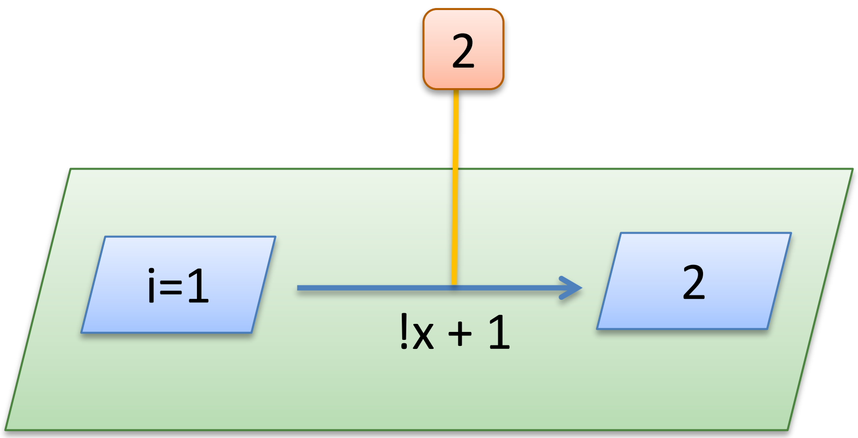
Mutation

What happens if your algorithm is not function? Uses mutable references?

Issue 2: An external agent modifies your reference



original run

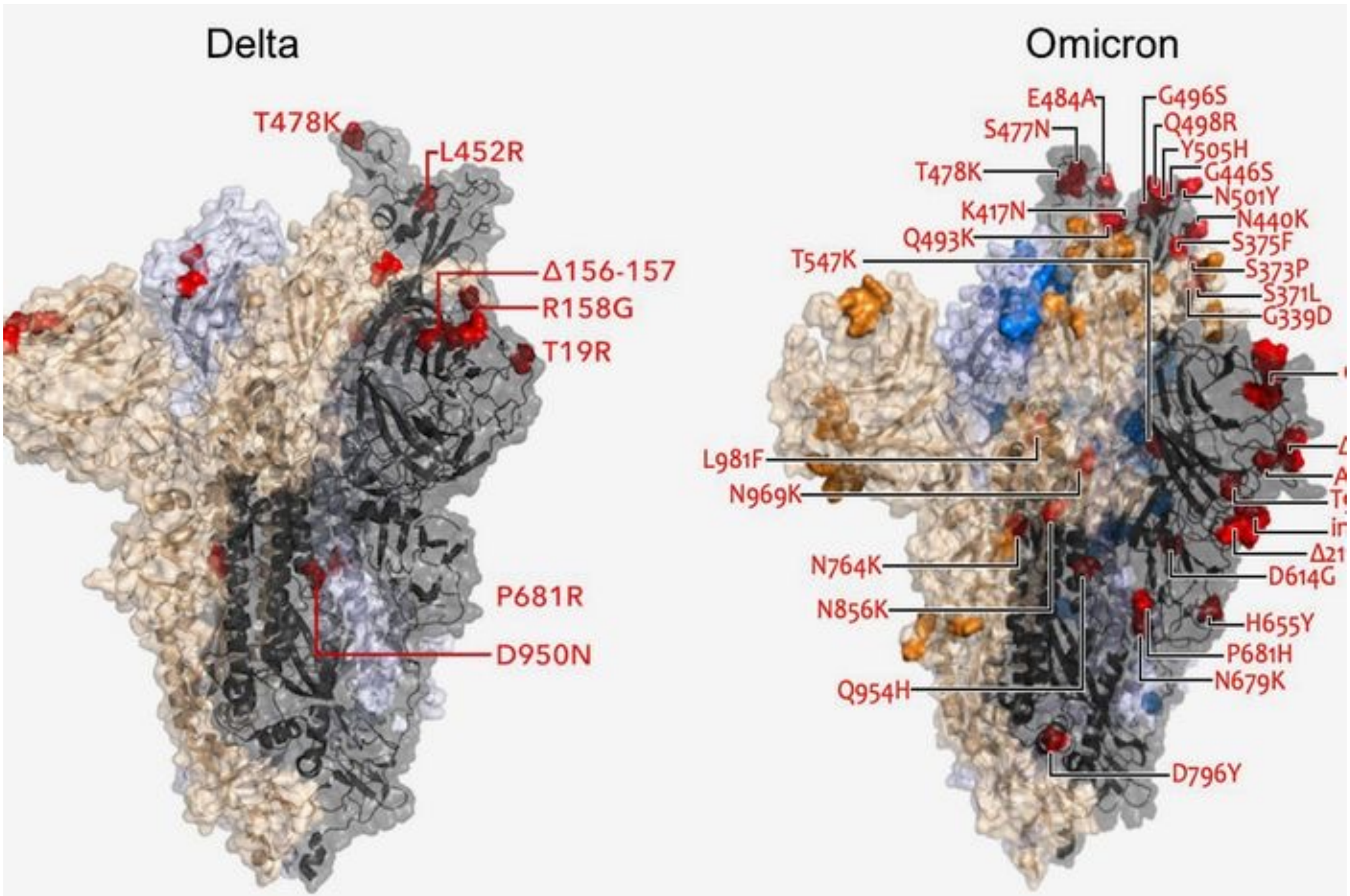


stabilize() will not rerun
the computation

You usually want your inputs to have type 'a Var.t so you can watch them.

AN APPLICATION: INCREMENTAL LONGEST COMMON SUBSEQUENCE ALGORITHMS

Comparative Genomics



DNA Sequences

A C T G C A ....

Nucleotides

A(denine)
C(ytosine)
G(uanine)
T(hymine)

Longest Common Subsequence

X is a *Subsequence* of Y if X can be obtained from Y by deleting some of the elements of Y.

A C T G C A

subsequences

A ... A C T A ... A C T G C A

A *Longest Common Subsequence* between Z and W is a subsequence S of both Z and W that is as long or longer than any other subsequence of Z and W.

A C T G C A

C A T

LCS

C A

A T

C T

Longest Common Subsequence: Rule 1.

Rule 1: first letters match

Input Sequences	Longest Common Subsequence
A :: [... rest1 ...] A :: [... rest2 ...]	A :: LCS (rest1, rest2)

Example	Longest Common Subsequence
A C G T A C T A	A C T

LCS of rest is C T

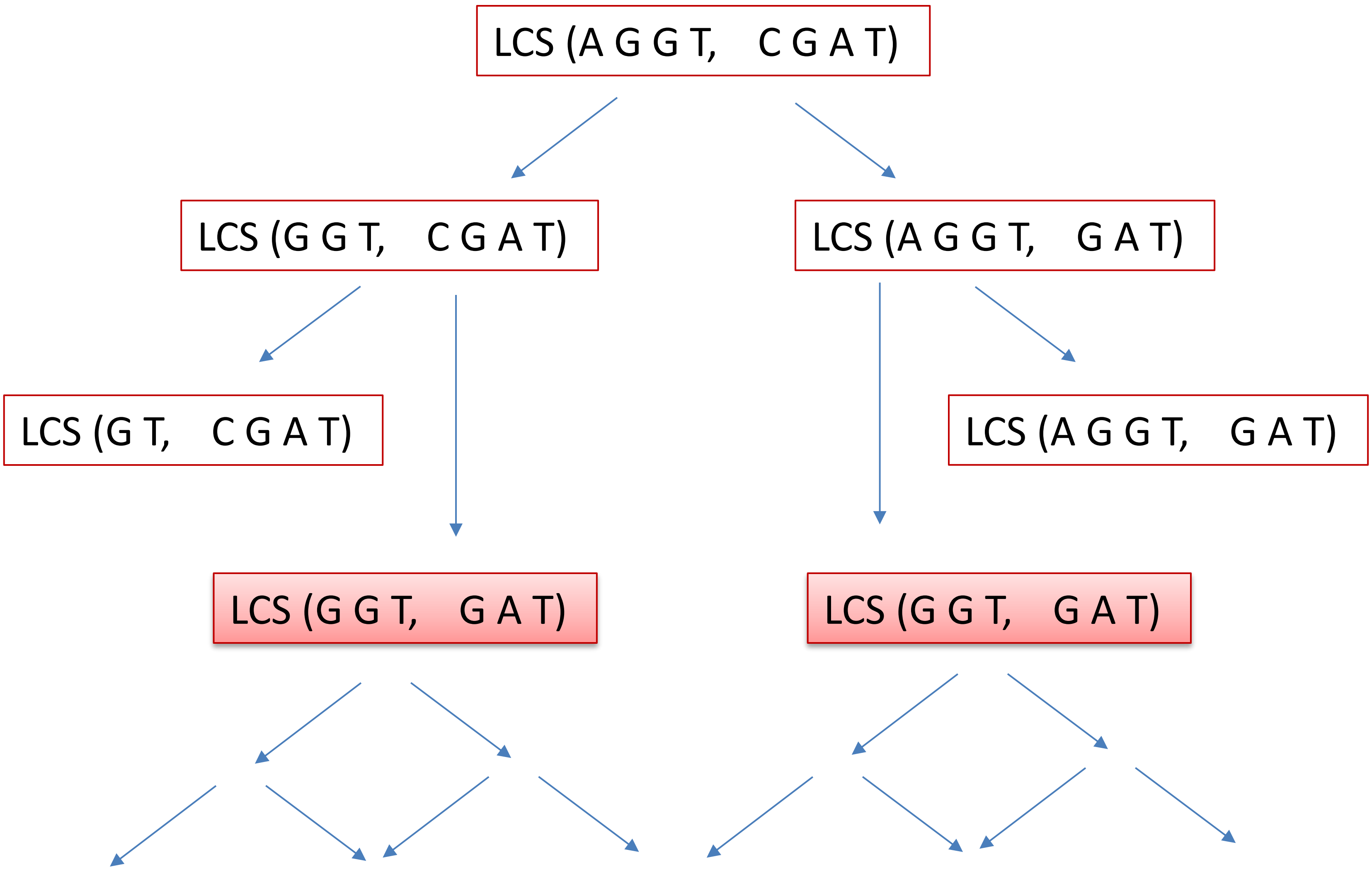
Longest Common Subsequence: Rule 2.

Rule 2: first letters can't match

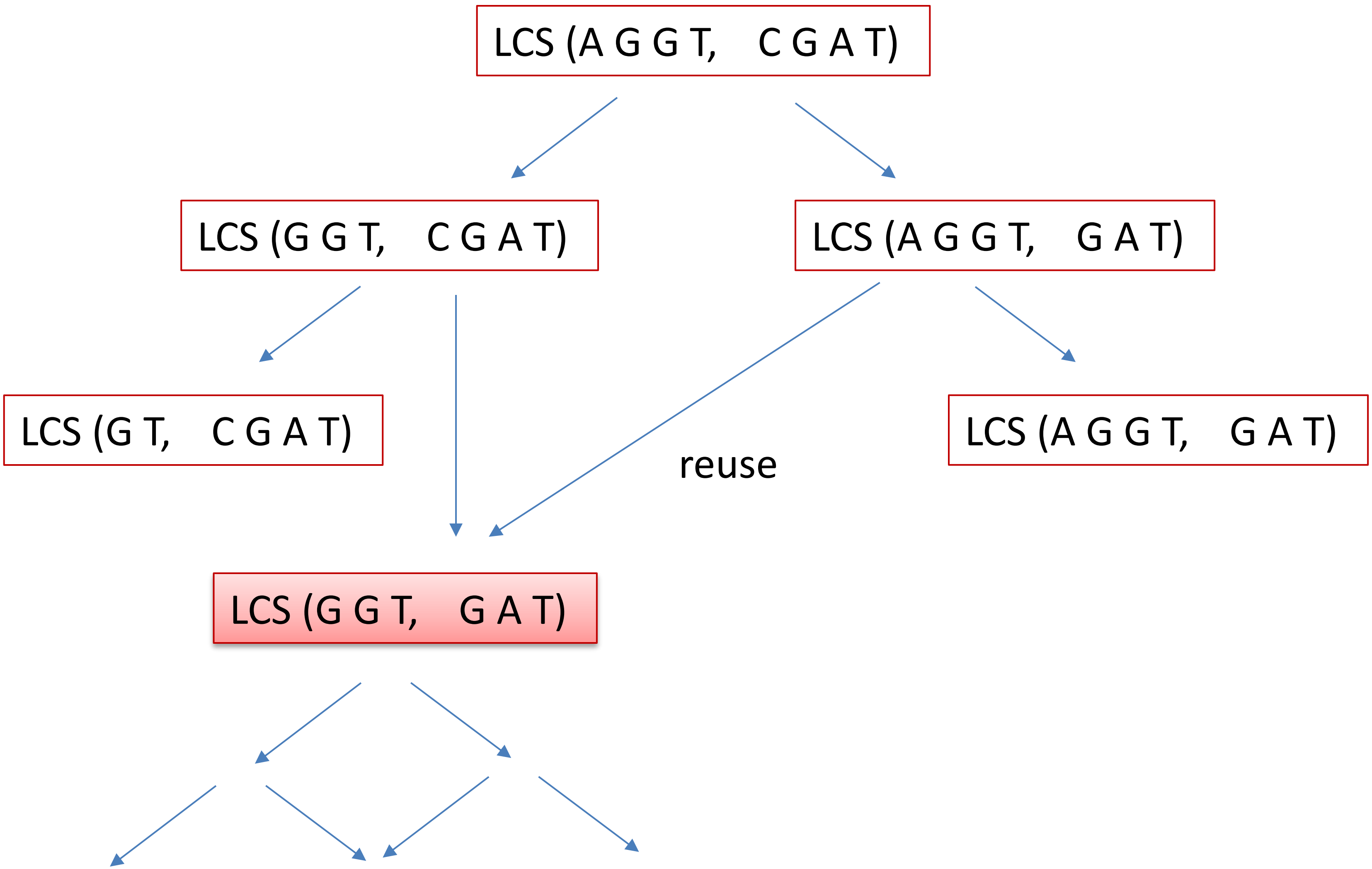
Input Sequences	Longest Common Subsequence
A :: [... rest1 ...]	LCS (A::rest1, rest2)
	or
C :: [... rest2 ...]	LCS (rest1, C::rest2)
	(whichever is longest)

Example	Longest Common Subsequence	
A C G T	A C G T	A C G T
C G A T	C G A T	C G A T
	LCS is C G T	LCS is A T (or G T)
	C G T	

Redundant Computation



Redundant Computation

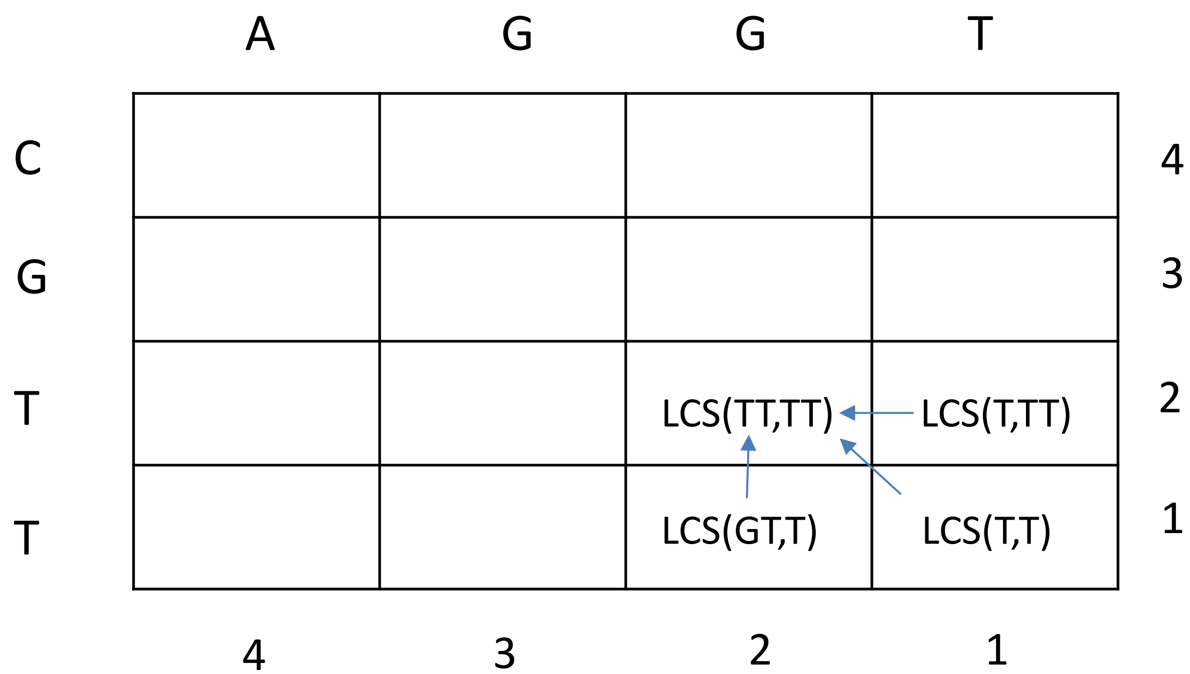


Memoize Results (Dynamic Programming)

	A	G	G	T	
C	LCS(AGGT, CGTT)				4
G				LCS(T,GTT)	3
T			LCS(TT,TT)		2
T				LCS(T,T)	1
	4	3	2	1	

Cell (i, j) contains LCS (input1[i..], input2[j..])

Memoize Results (Dynamic Programming)



Cell (i, j) depends on

Cell (i-1, j-1), or

Cell (i, j-1) and Cell(i-1, j-1)

Memoize Results (Dynamic Programming)

	A	G	G	T
C	GT	GT	GT	T
G	GT	GT	GT	T
T	T	T	T	T
T	T	T	T	T

Memoize Results (Dynamic Programming)

	A	G	G	T
C	GT	GT	GT	T
G	GT	GT	GT	T
T	T	T	T	T
T	T	T	T	T

Implementation Data Structure

	A	G	G	T
C	GT	GT	GT	T
G	GT	GT	GT	T
T	T	T	T	T
T	T	T	T	T

Create a key-value map to store intermediate results:

- keys have type dna * dna
- values have type dna * length
- Dict.find (dna1, dna2) = LCS(dna1, dna2)

Implementation Data Structure: Phase 1

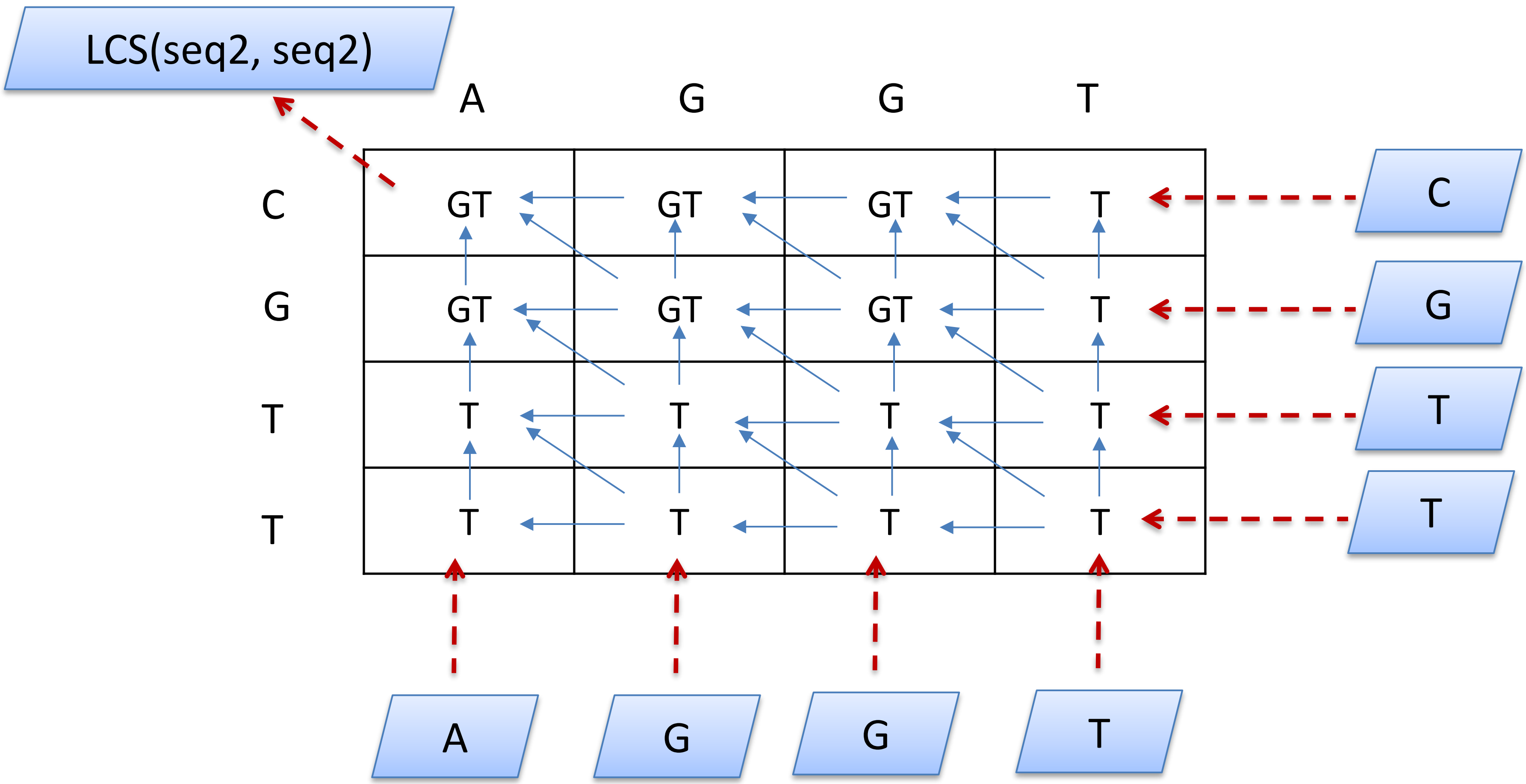
	A	G	G	T
C	GT	GT	GT	T
G	GT	GT	GT	T
T	T	T	T	T
T	T	T	T	T

You will actually create a generic memoizer

- A functor generates a memoizer for *any* function!
- You'll apply it to the LCS algorithm

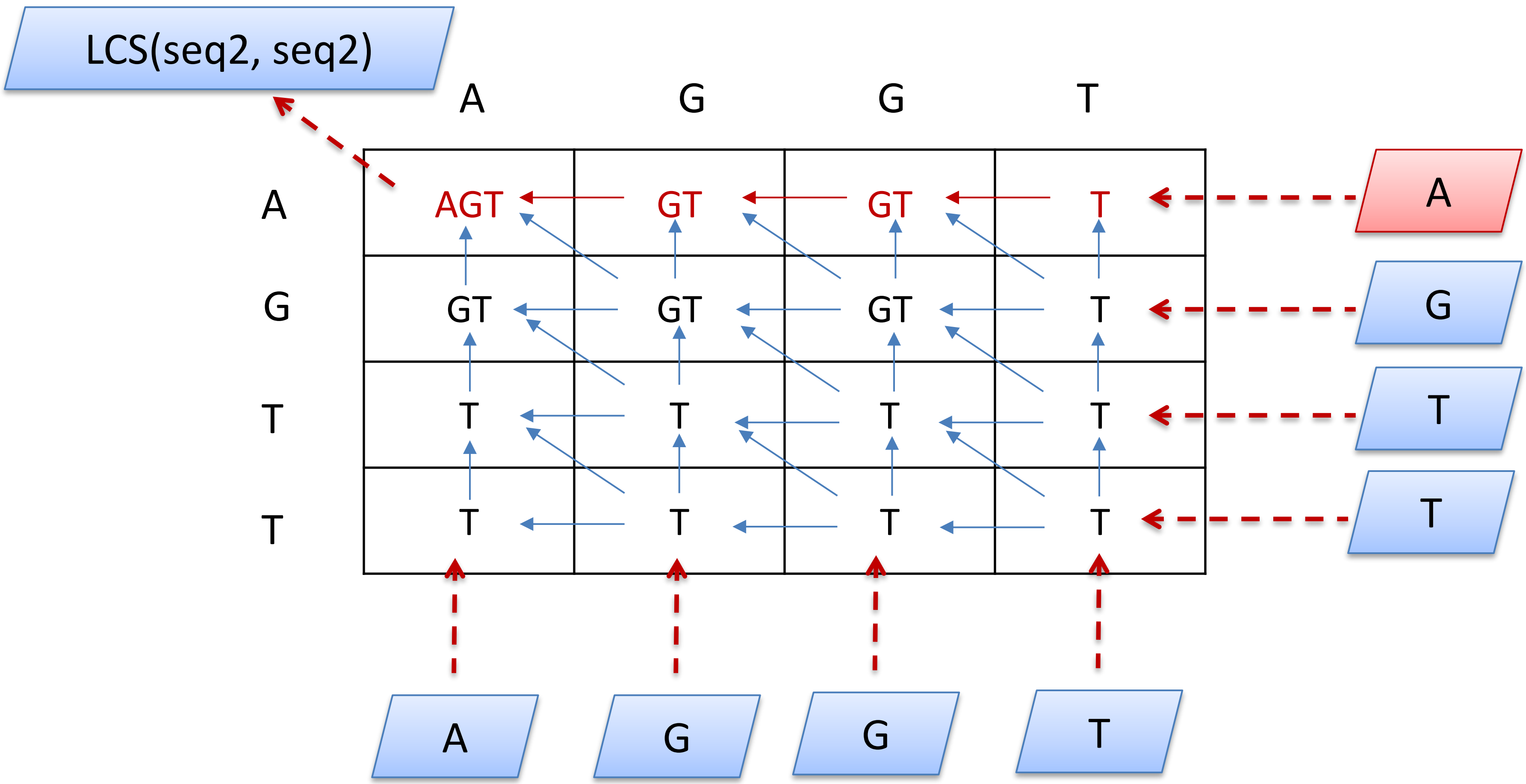
Implementation Data Structure: Phase 2

Build an incremental dependency graph



Implementation Data Structure: Phase 2

Mutate input cells; Stabilize incremental graph; Obtain result



Assignment Summary: Caching N Ways

- Lazy computation with infinite data structures (streams.ml)
  - lazy results get cached
  - infinite speedup when you process infinite data structures!
- Manually Memoizing Fibonacci (memo.ml)
  - fib n = fib (n-1) + fib (n-2) if n > 1
  - recursive calls are cached to avoid exponential blow-up
- Auto-memoizing (memo.ml)
  - build a functor to cache results for any function
  - build a dictionary that maps function inputs to outputs
  - an automatic dynamic programmer
  - apply to LCS algorithm
- Incremental computation (lcs.ml)
  - build a dictionary of incrementals
  - incrementally recompute LCS

What did you get out of this course?

HOT (Higher-order, Typed) programming gives great code reuse

All languages should have lambdas

All languages should have ML data types. Concise code + exhaustiveness checks for the win

Get work done by creating new data not always by changing old data

Immutable data preserves invariants, simplifies reasoning about your code

If you have inductive data, think inductively! Assume your IH and use it to compute your answer

Functions are data structures

Indeed, represent functions using data structures (ie: ASTs)

OCaml

Prove things about entire programming languages via induction over their ASTs!

Substitution model of execution defines program semantics

But implement interpreters with an environment-based model and closures

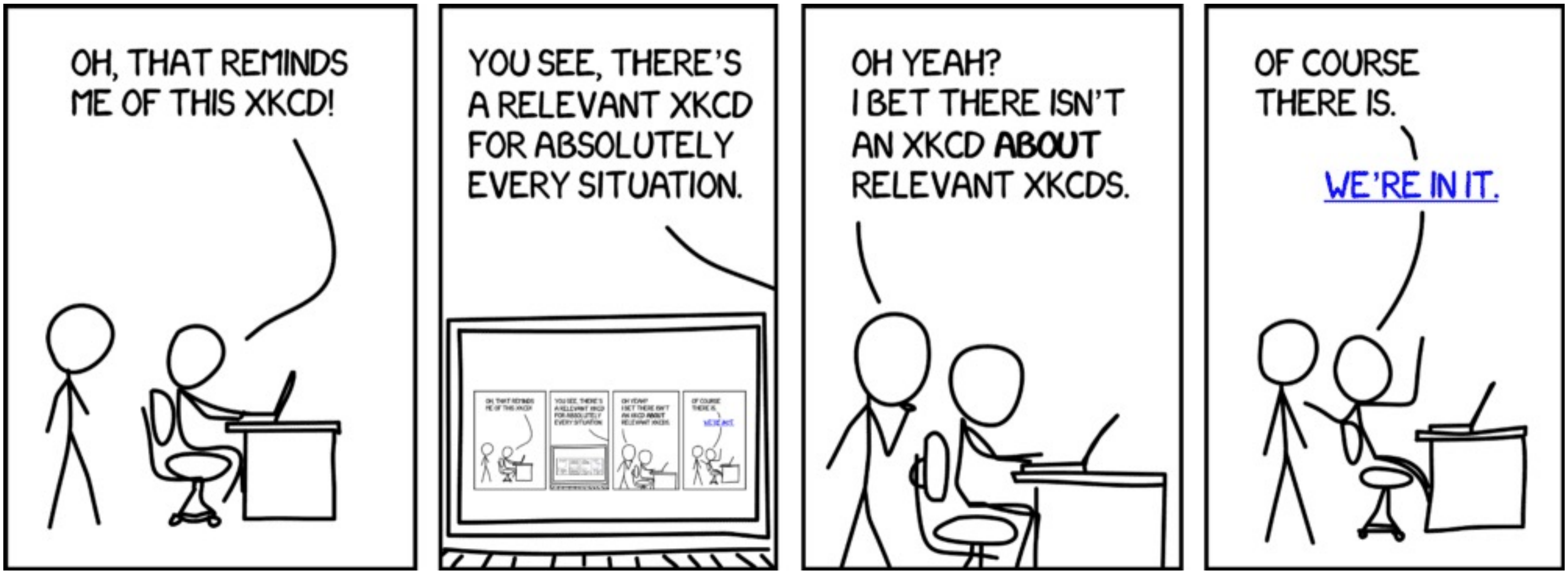
Check your representation invariants when data released from a module

Trees and sequences are good for parallel (and incremental) computation

Parallelism via Map, Reduce, Scan

Lazy evaluation allows you to program with the abstraction of infinite data

Java Programs are insanely verbose. Honestly, why go back?



Recursive XKCD