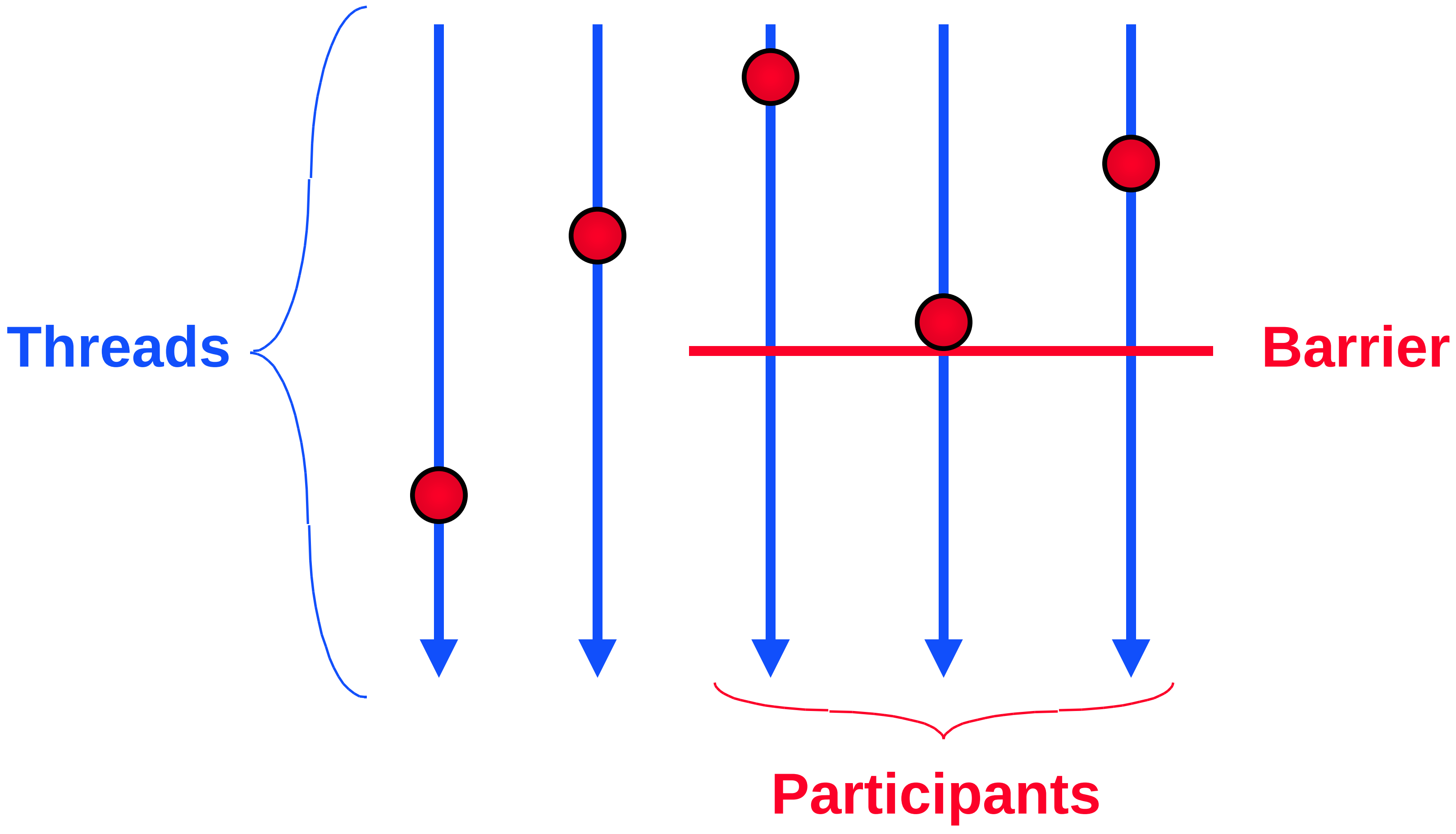
First-Class Synchronization Barriers

Franklyn Turbak
Wellesley College

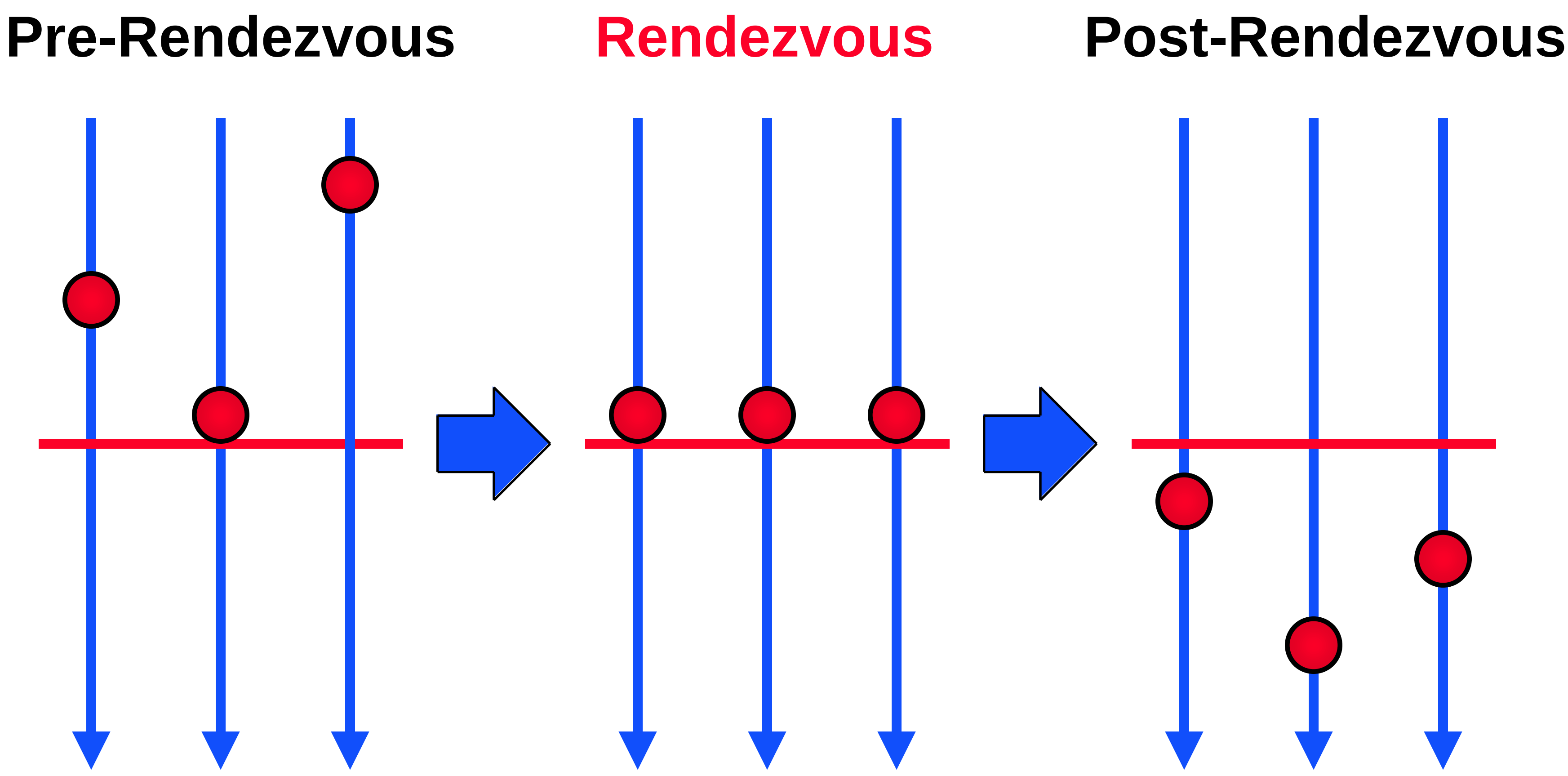
Overview

- What is a Synchronization Barrier?
- Dimensions of Barriers
- Synchrons: First-Class Barriers with a Variable Number of Participants
- Motivating Example:
  Space-Efficient Aggregate Data Programs
- Discussion

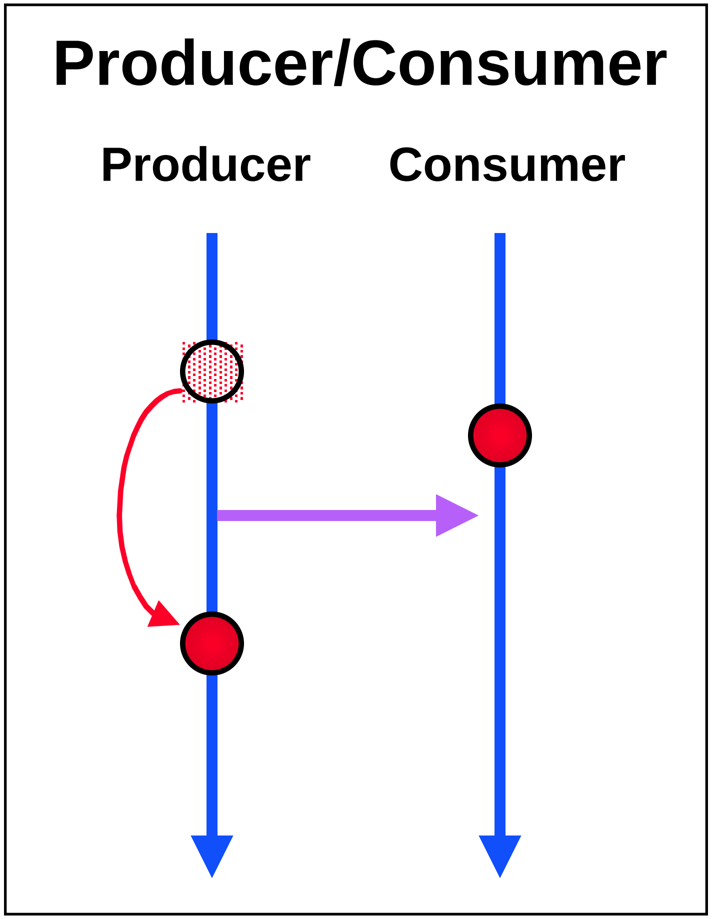
Synchronization Barriers

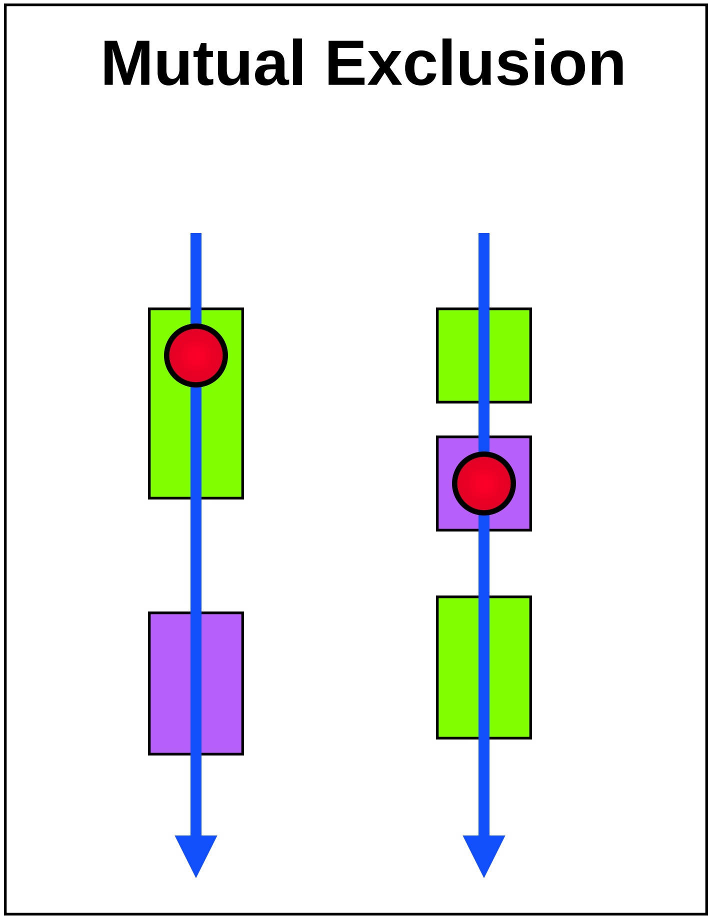


Barrier Rendezvous



Non-Barrier Synchronization





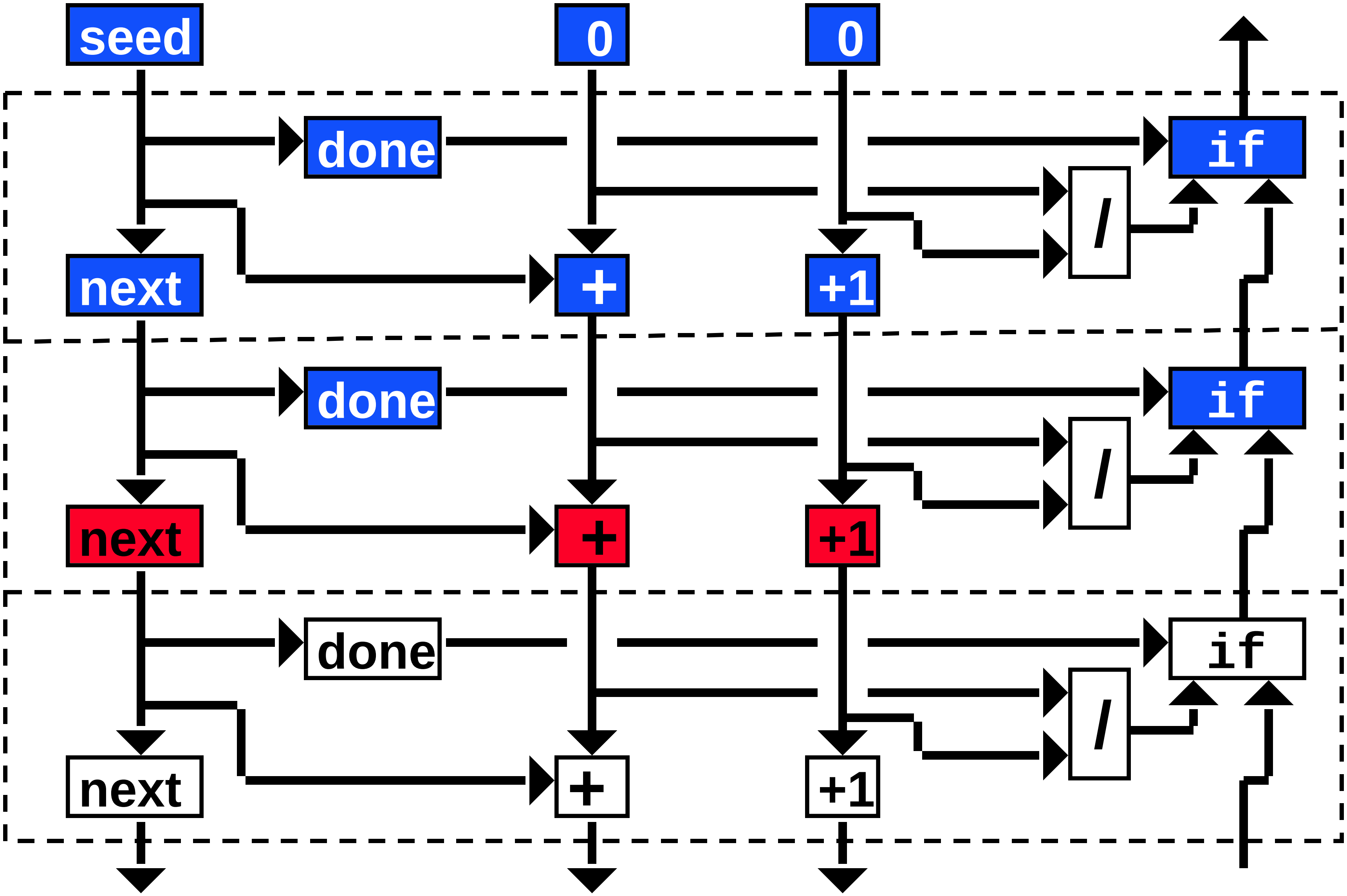
What Are Barriers Good For?

- CSP-Style Handshake
- Coordinating Side Effects
  - Mutable Variables
  - I/O
- Managing Resources
  - Number of Threads
  - Space

Strict `average` program (Scheme)

```
(define (average seed next done)
  (define (loop n sum count)
    (if (done n)
        (/ sum count)
        (loop (next n)
              (+ sum n)
              (+ count 1))))
  (loop seed 0 0))
```

`average` (strict)

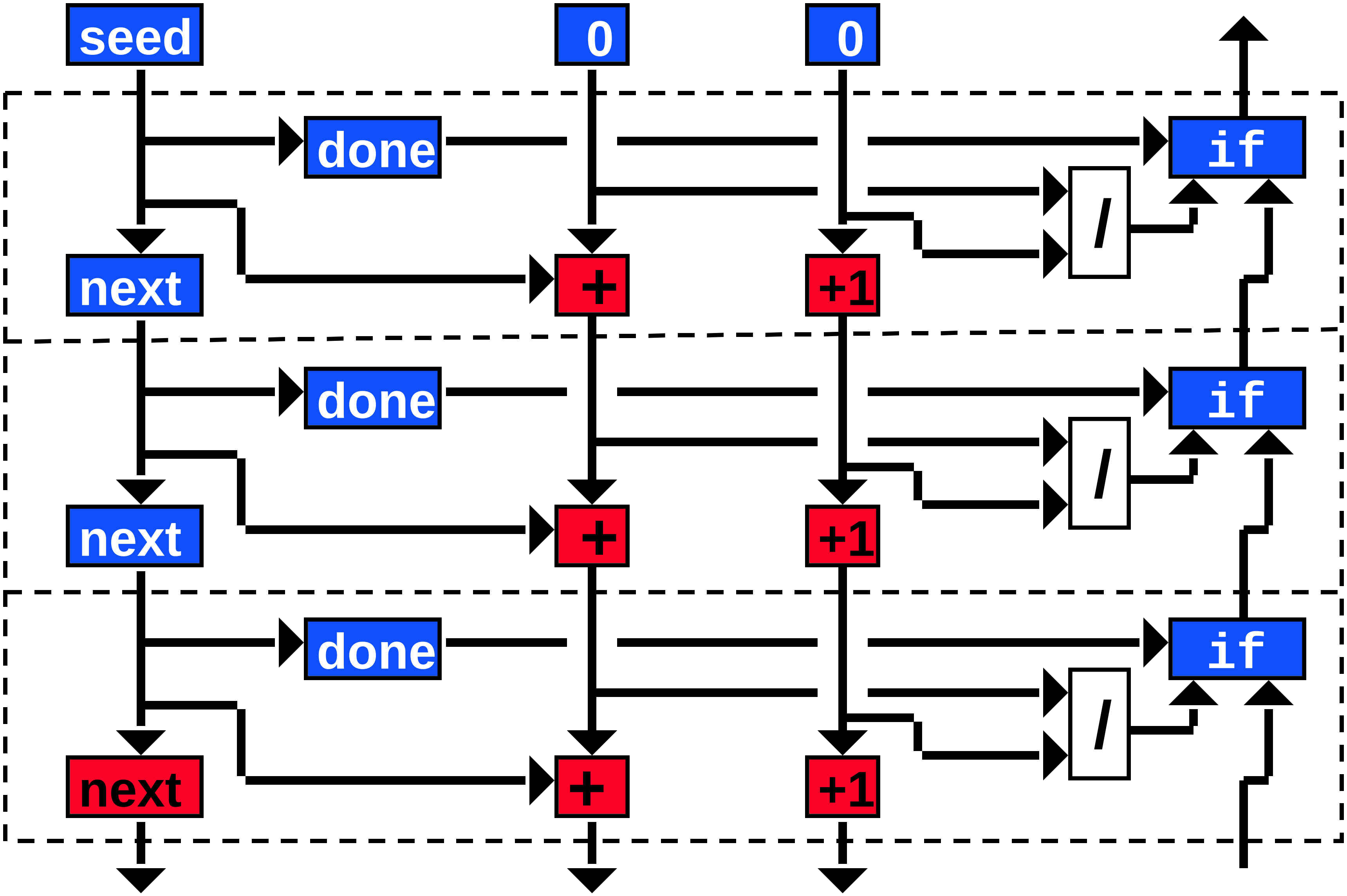


Eager `average` program (Id)

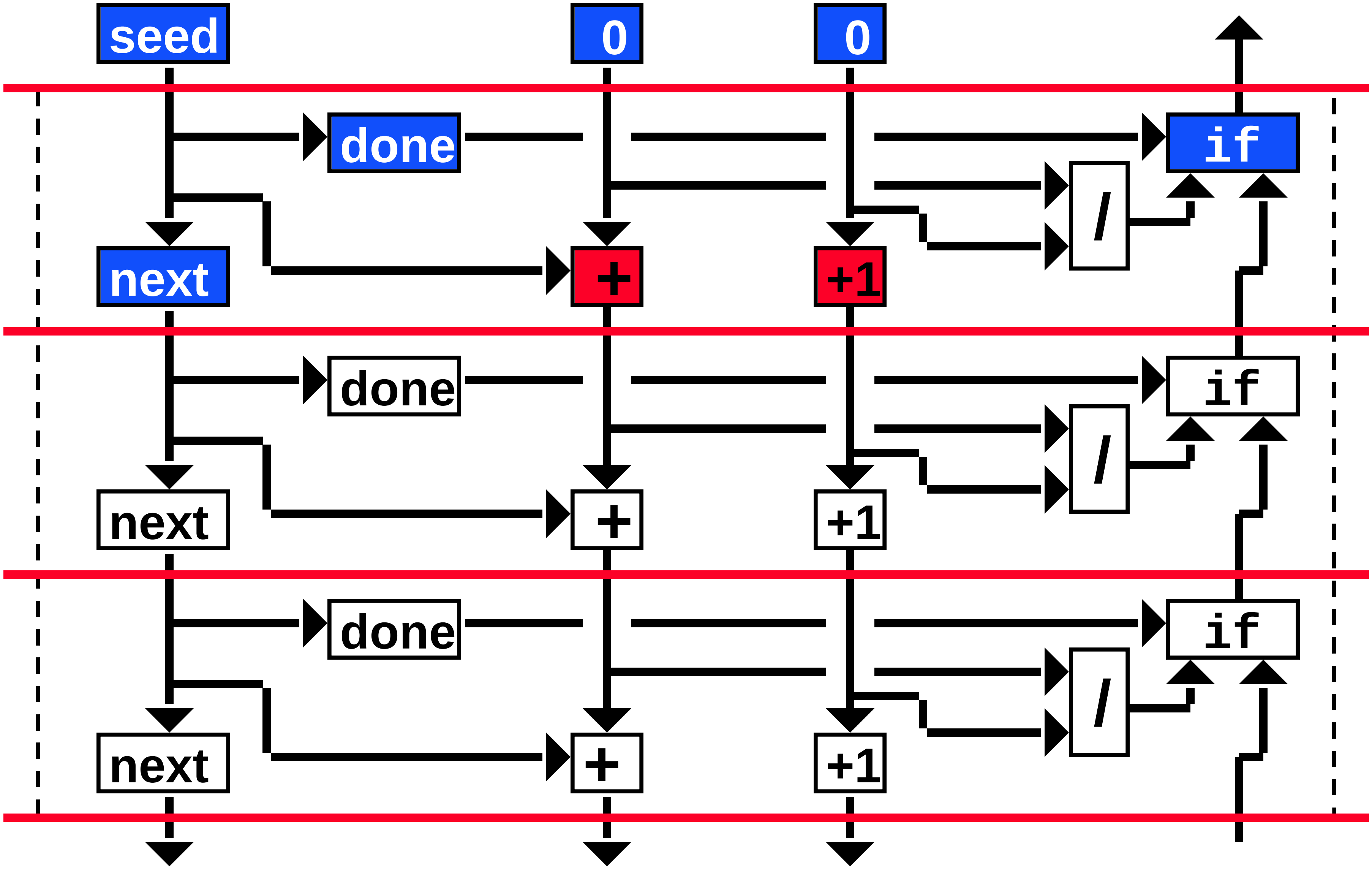
concurrent

```
def average seed next done =
  {def loop n sum count =
    if (done n) then
     sum/count
    else
     { new_n = (next n);
       new_sum = sum + n;
       new_count = count + 1;

       in loop new_n new_sum new_count };

 in loop seed 0 0};
```

average (eager)



`average` (eager with barriers)



Eager `average` program (Id)
with barriers

concurrent

```
def average seed next done =
  {def loop n sum count =
    if (done n) then
     sum/count
    else
     { new_n = (next n);
       new_sum = sum + n;
       new_count = count + 1;
       ------------------------
       in loop new_n new_sum new_count };

  in loop seed 0 0};
```

Overview

- What is a Synchronization Barrier?
- Dimensions of Barriers
- Synchrons: First-Class Barriers with a Variable Number of Participants
- Motivating Example:
  Space-Efficient Aggregate Data Programs
- Discussion

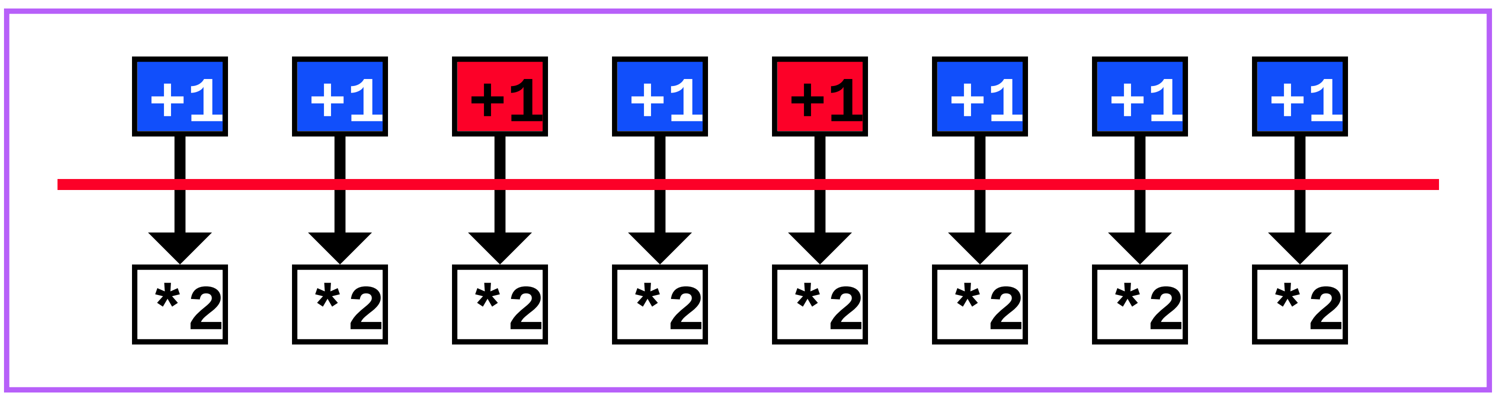
Barrier Dimensions

		Dynamically varying number of participants? no	Dynamically varying number of participants? yes
First Class?	no		
First Class?	yes		

Barrier Dimensions

Dynamically varying number of participants?

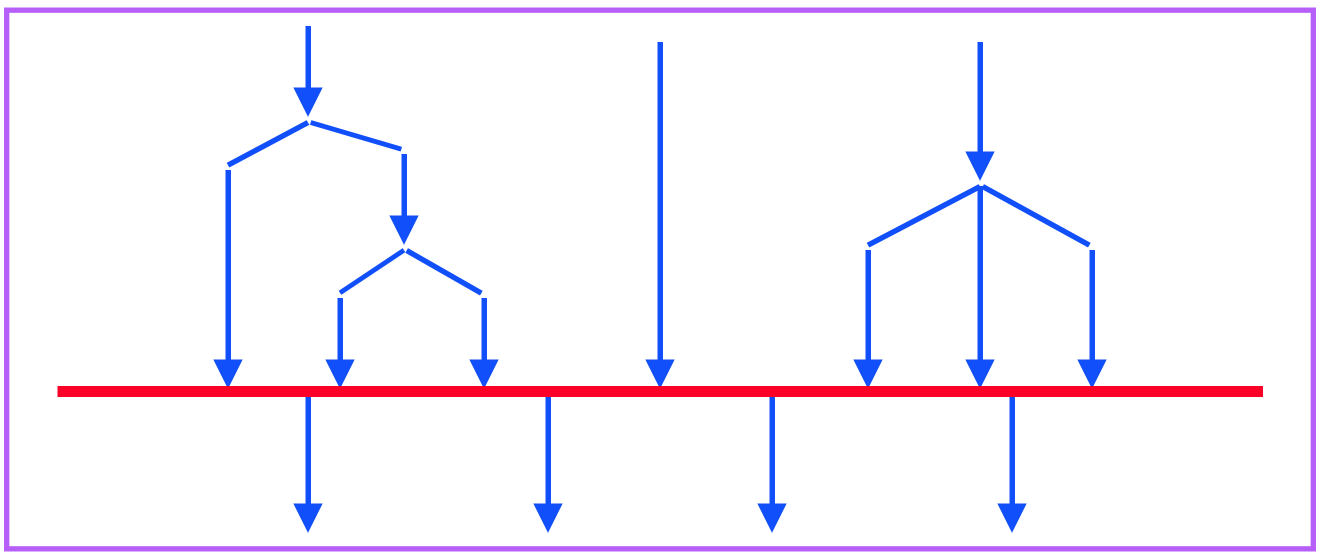
First Class?	no	yes
no	data parallel barriers	
yes		



Barrier Dimensions

Dynamically varying number of participants?

First Class?	no	yes
no	data parallel barriers	Id barriers
yes		



Barrier Dimensions

		Dynamically varying number of participants?	
		no	**yes**
First Class?	**no**	**data parallel barriers**	**Id barriers**
	yes	**fixed size barriers**	

`barrier` `:` *`int -> barrier`*

Return a first-class barrier for a rendezvous of *`int`* threads.

`pause` `:` *`barrier -> unit`*

Suspend the current thread until the rendezvous at *`barrier`*, after which `pause` resumes by returning *`unit`*.

Barrier Dimensions

First Class? \ Dynamically varying number of participants?	no	yes
no	data parallel barriers	Id barriers
yes	fixed size barriers	synchrons

Overview

- What is a Synchronization Barrier?
- Dimensions of Barriers
- Synchrons: First-Class Barriers with a Variable Number of Participants
- Motivating Example:
  Space-Efficient Aggregate Data Programs
- Discussion

Synchron Interface

`synchron` : *`unit -> synchron`*

Return a first-class barrier for a rendezvous of an as yet undetermined number of threads.

`wait` : *`synchron -> unit`*

Suspend the current thread until the rendezvous at the synchron, after which `wait` resumes by returning `unit`.

`simul` : *`synchron * synchron -> unit`*

Constrain the two synchrons to be the same barrier.

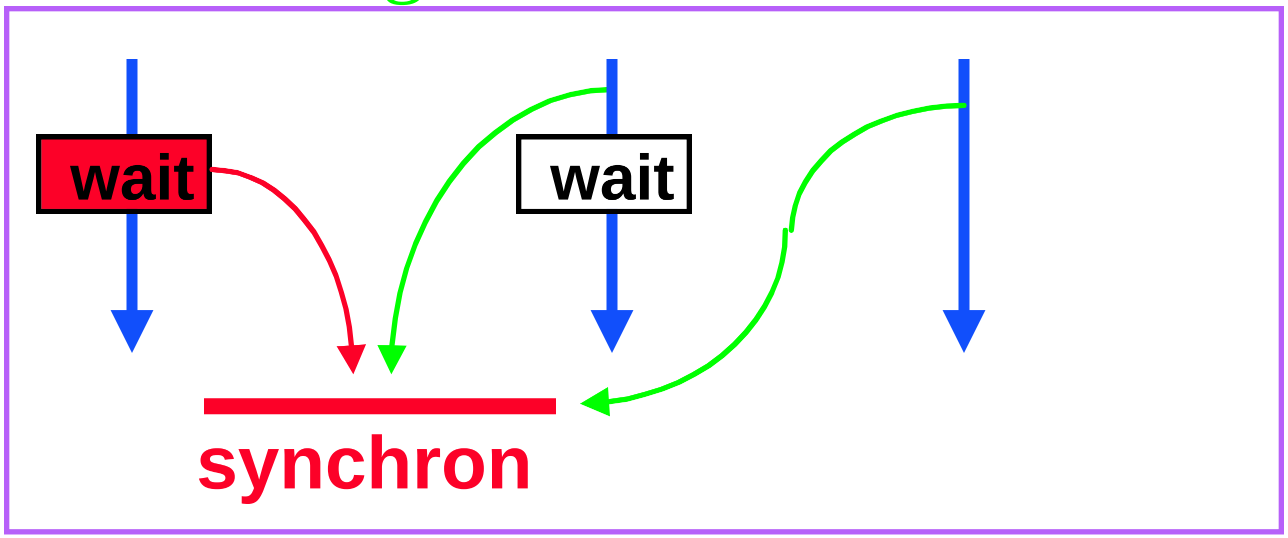
Synchron Rendezvous Condition

A rendezvous occurs at a synchron when all threads that could ever wait at the synchron are waiting at the synchron.

In practice, approximate rendezvous condition by tracking pointers to a synchron via automatic storage manager.

Rendezvous Semantics

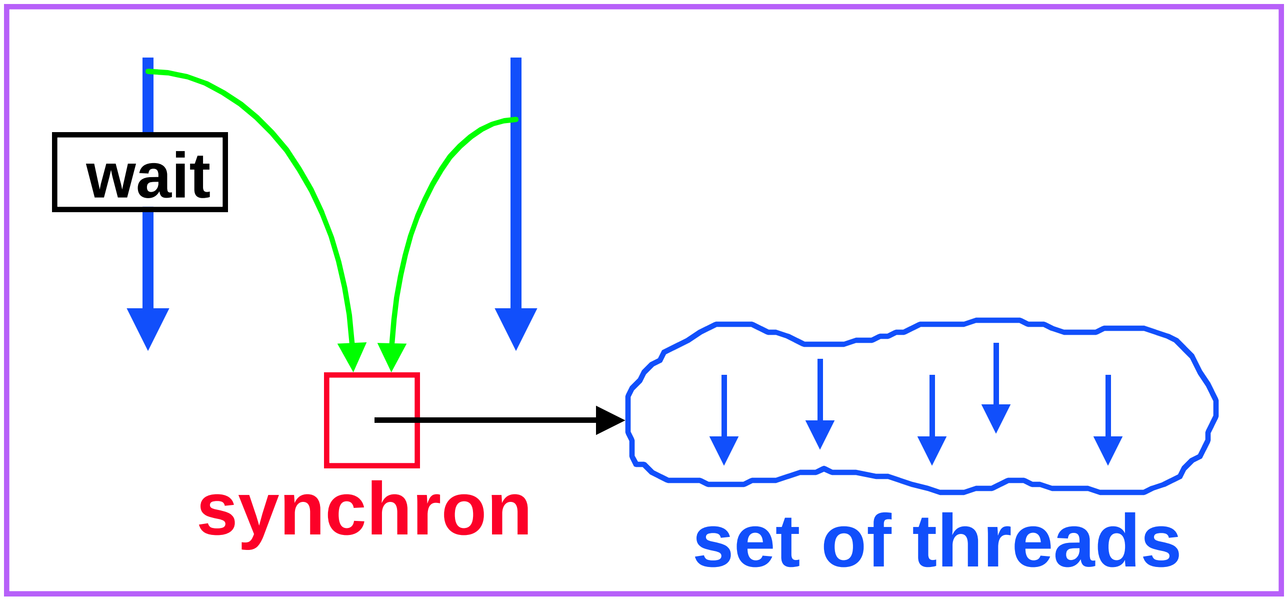
- Classify pointers to a synchron as waiting or non-waiting.



- Rendezvous occurs at a synchron when all pointers to it are waiting.
- Non-waiting pointers block rendezvous.
- No pointers left after rendezvous.

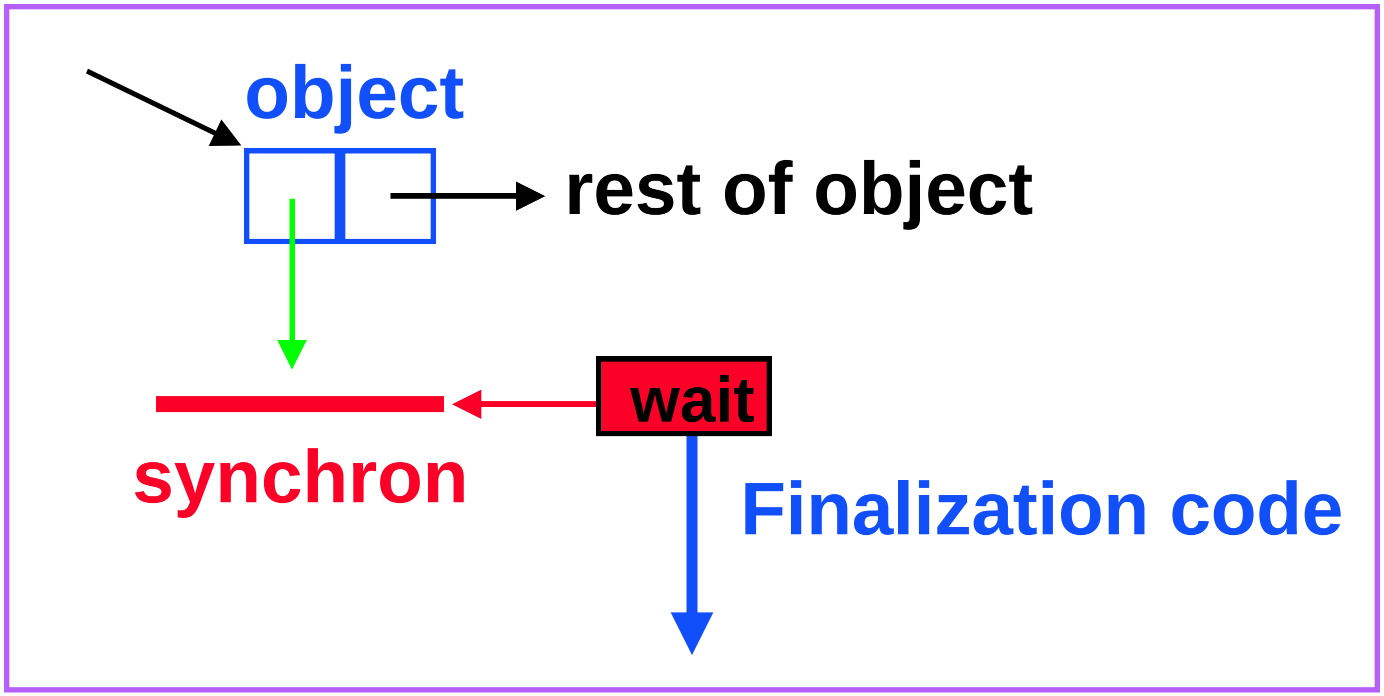
Synchrons are Equivalent to Object Finalization

- Can implement synchrons in terms of object finalization:



Synchrons are Equivalent to Object Finalization

- Can implement object finalization in terms of synchrons:

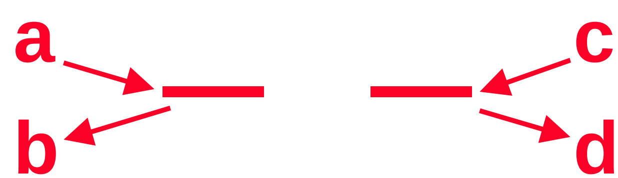


Synchron Examples

```
(define (f s)
  (begin (display ‘a) (wait s) (display ‘b)))
(define (g s)
  (begin (display ‘c) (wait s) (display ‘d)))
```

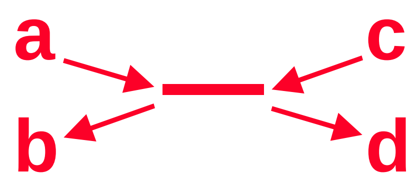
1

```
(par (f (synchron))
     (g (synchron))
```



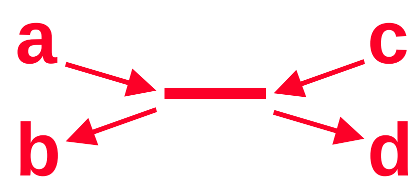
2

```
(let ((s (synchron)))
  (par (f s) (g s)))
```



3

```
(let ((s1 (synchron))
      (s2 (synchron)))
  (par (f s1) (g s2) (simul s1 s2)))
```



Synchron Subtleties

Need detailed model to reason about liveness.
We use Appel's Safe for Space Complexity model.

1

```
(let ((s (synchron)))
  (begin (wait s)
         (wait s)))   deadlock!
```

2

```
(let ((p (cons (synchron) 17)))
  (begin (wait (car p))
         (cdr p)))    deadlock!
```

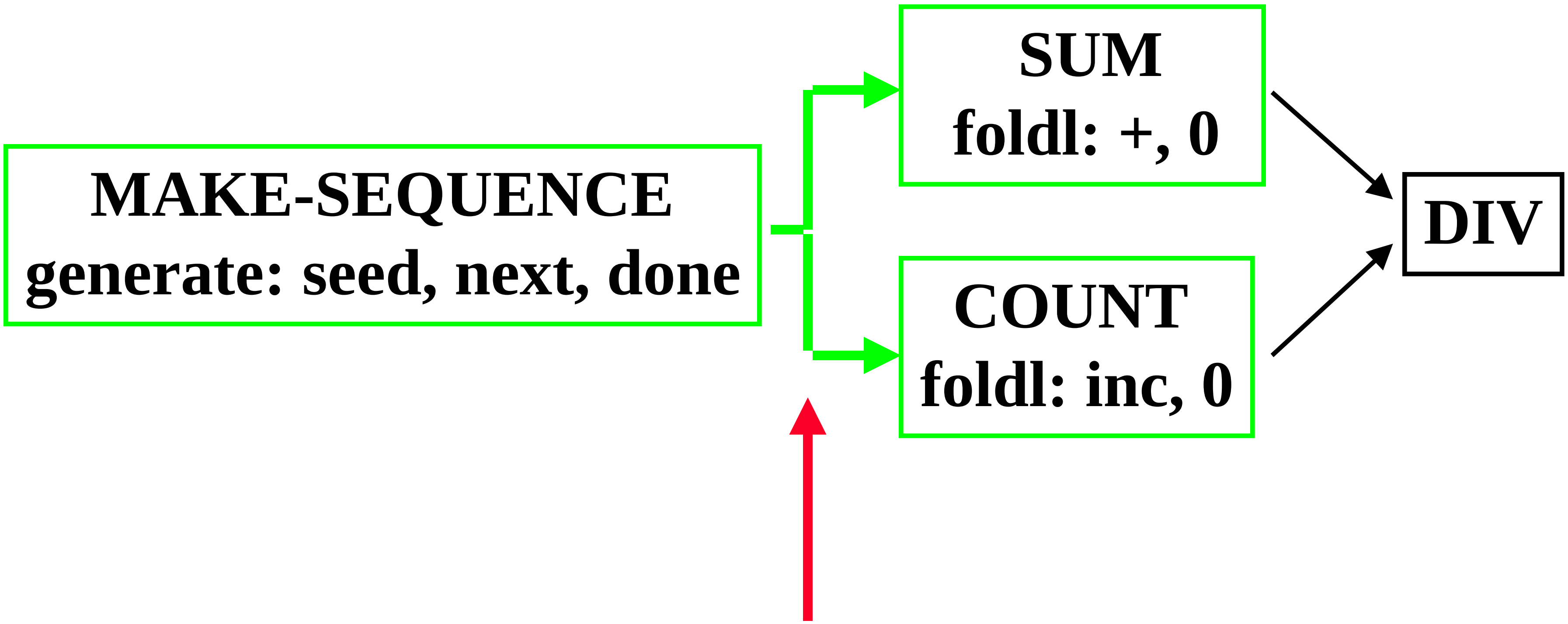
3

```
(let ((p (cons (synchron) 17)))
  (let ((s (car p))
        (n (cdr p)))
    (begin (wait s) n)))   17
```

Overview

- What is a Synchronization Barrier?
- Dimensions of Barriers
- Synchrons: First-Class Barriers with a Variable Number of Participants
- Motivating Example:
  Space-Efficient Aggregate Data Programs
- Discussion

A Modular `average`



How to represent sequence
to guarantee constant space?
(Fan-out is a problem!)

Modular `average` program (Scheme)

```
(define (average seed next done)
  (let ((nums (generate seed
                        next
                        done)))
    (/ (foldl + 0 nums)
       (foldl (lambda (x y) (+ 1 y))
              0
              nums))))
```

generate and foldl

```
(define (generate seed next done)
  (if (done? seed)
      ‘()
      (pack init
            (generate (next seed)
                      next
                      done))))

(define (foldl op acc seq)
  (if (null? seq)
      acc
      (unpack seq
        (lambda (hd tl)
          (foldl op (op hd acc) tl)))))
```

Strict list strategy

(pack *E1 E2*) desugars to (cons *E1 E2*)

```
(define (unpack seq f)
  (f (car seq) (cdr seq)))
```

Lazy list strategy

```
(pack E1 E2)
  desugars to (cons E1 (delay E2))

(define (unpack seq f)
  (f (car seq) (force (cdr seq))))
```

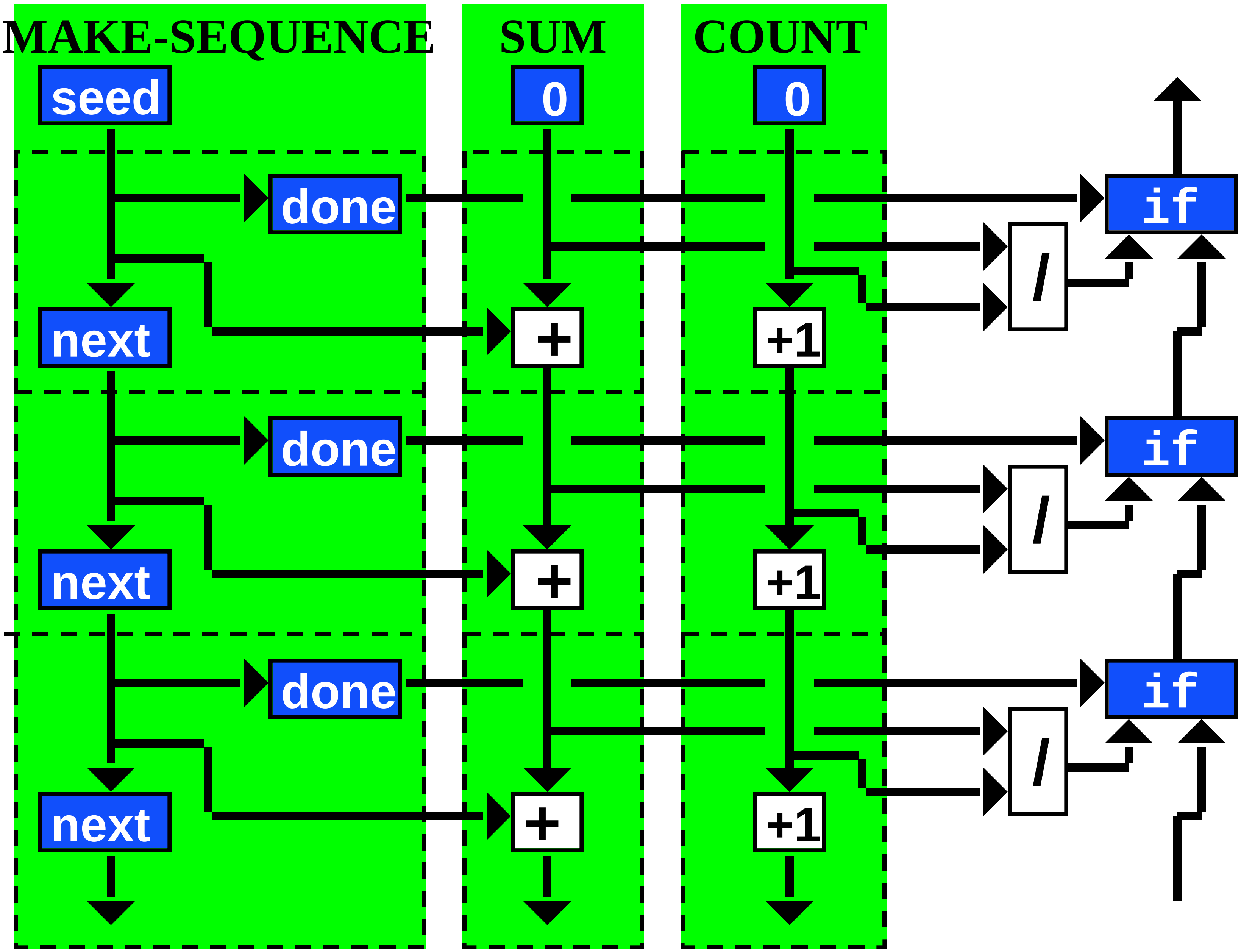
Synchronized lazy list strategy

```
(pack E1 E2)
  desugars to (list (synchron)
                    E1
                    (delay E2))

(define (unpack seq f)
  (let ((sync (first seq))
        (hd (second seq))
        (tl (third seq)))
    (begin (wait sync)
           (f hd (force tl)))))
```

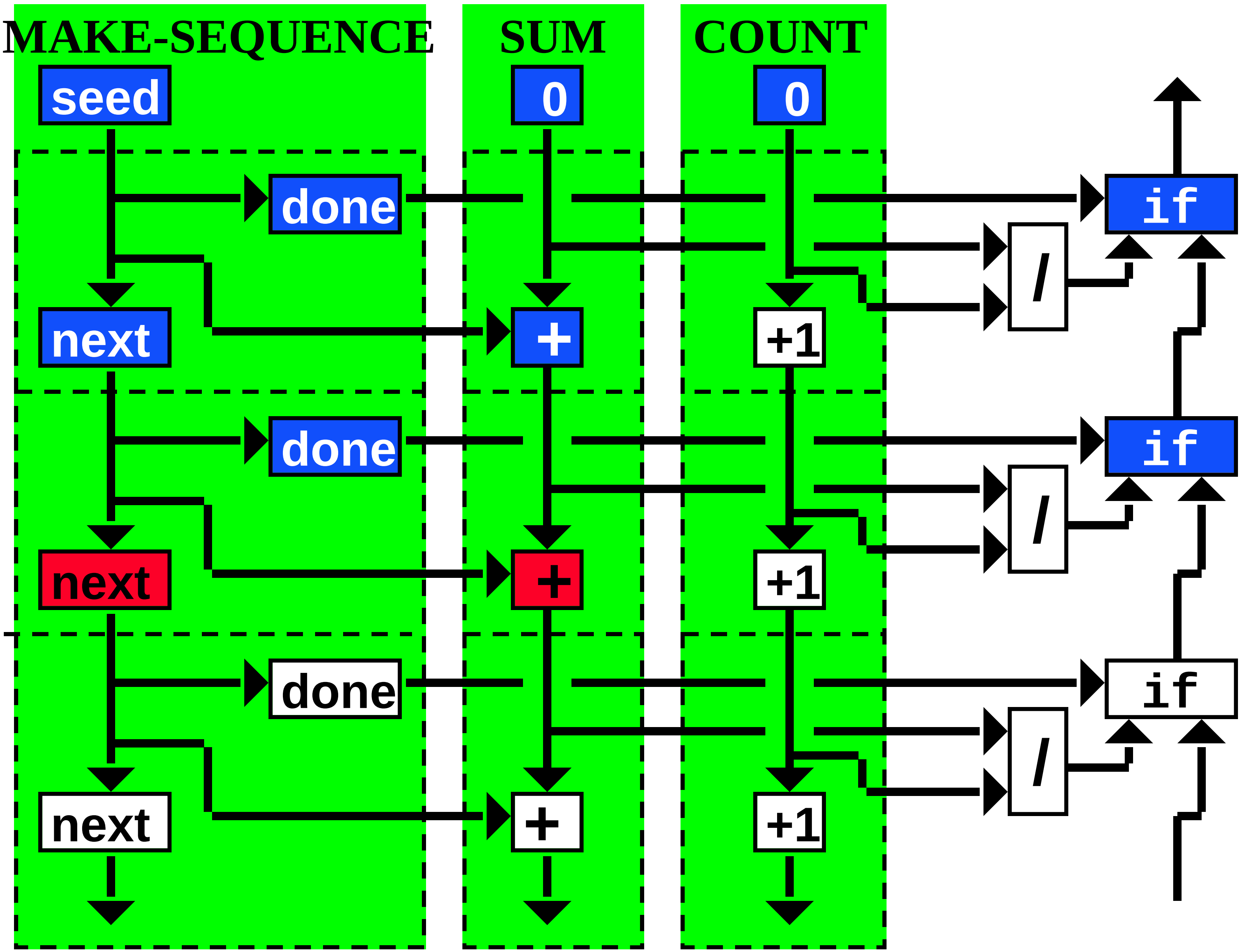
Strategies for modular `average`

- Strict Lists: $\Theta(n)$ space
- Lazy Lists: $\Theta(n)$ space
- Hughes (1984): Any sequential implementation will take $\Theta(n)$ space
- Concurrency alone is not enough -- Eager Lists: $\Theta(n)$ space
- Synchronized Lazy Lists: $\Theta(1)$ space

Modular `average` (strict lists)



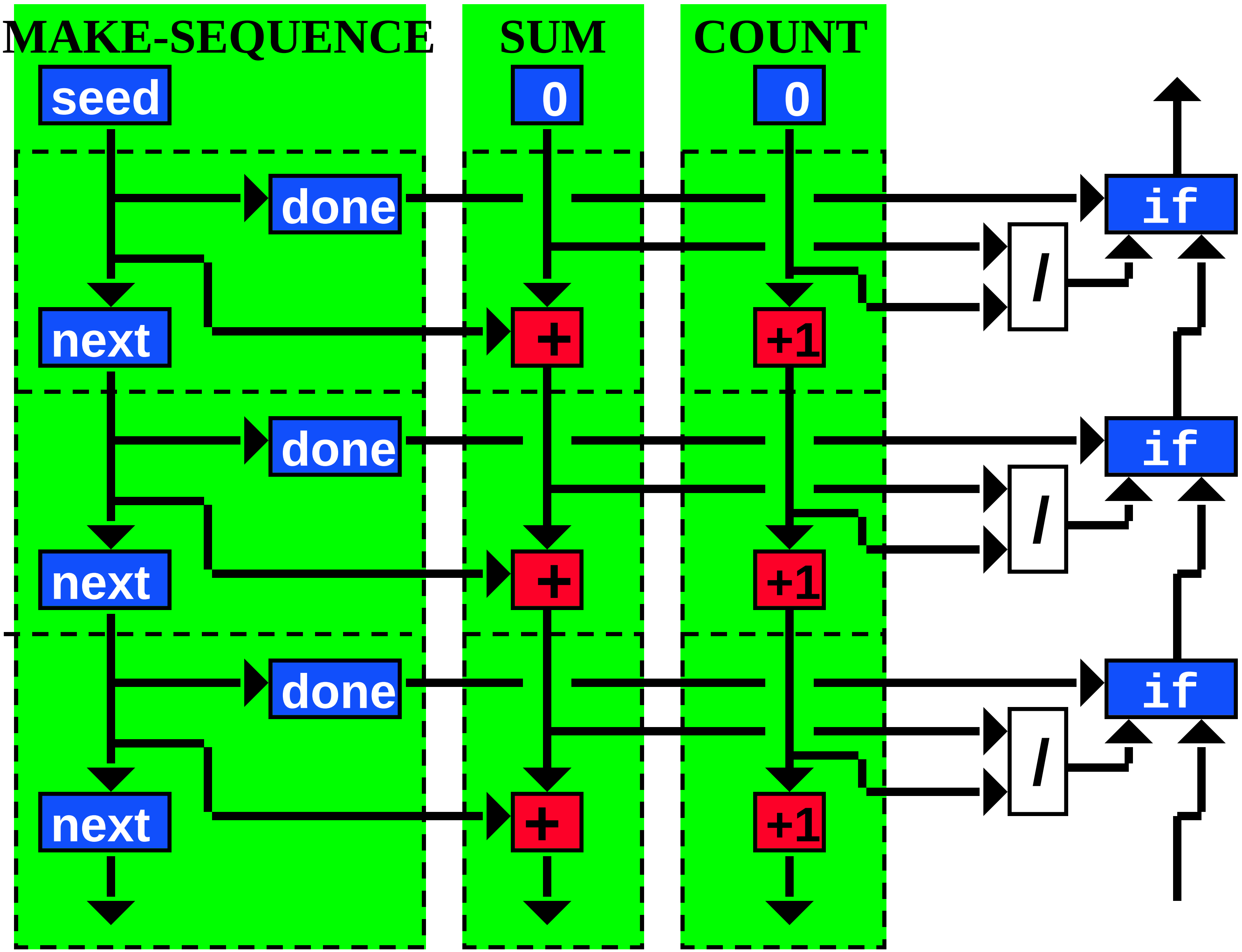
Modular `average` (lazy lists)



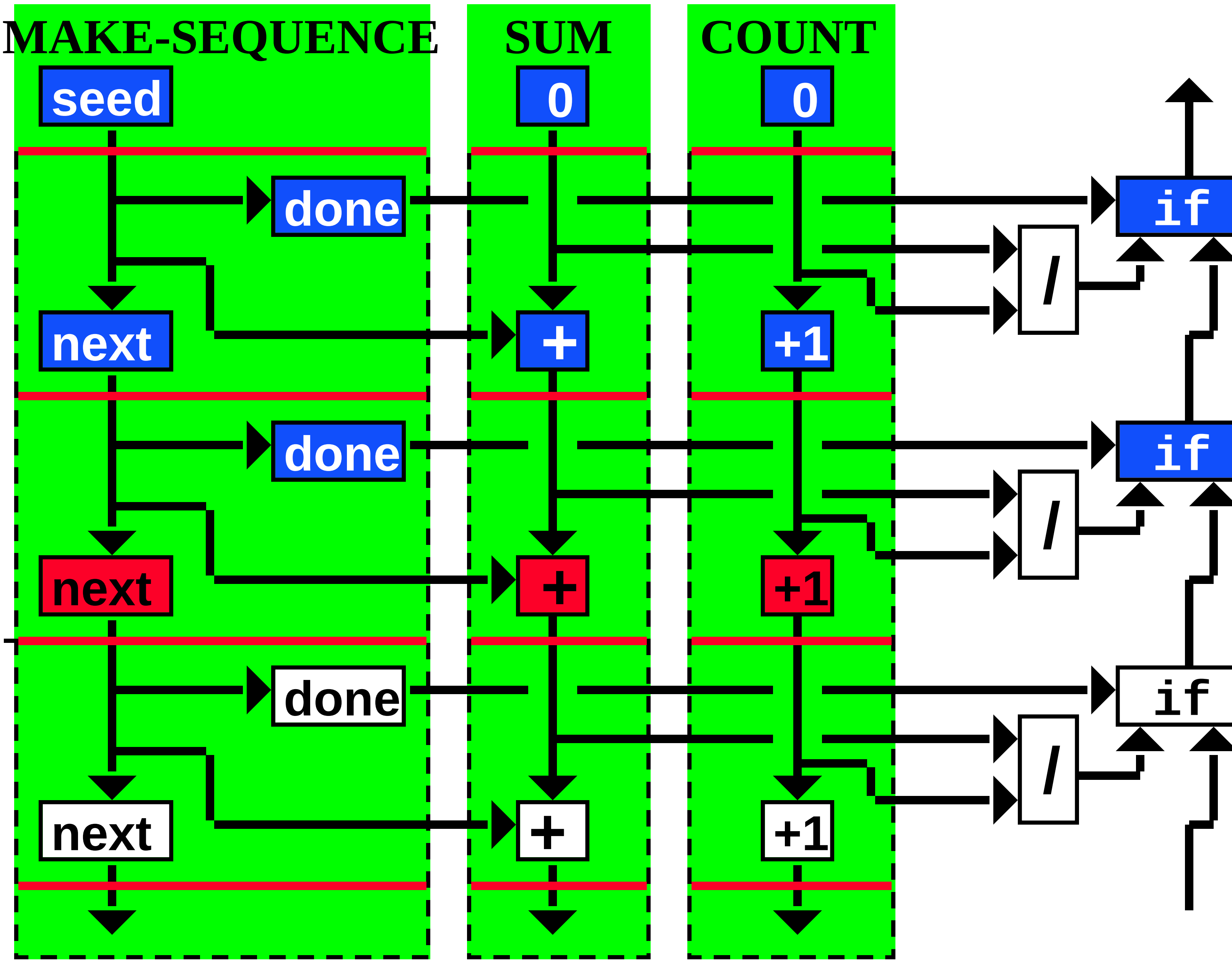
Modular average cannot be space-efficient in a sequential language

- Hughes: “Parallel Functional Languages Use Less Space” (1984)
- Need some form of concurrency and synchronization for modular space-efficient aggregate data programs.

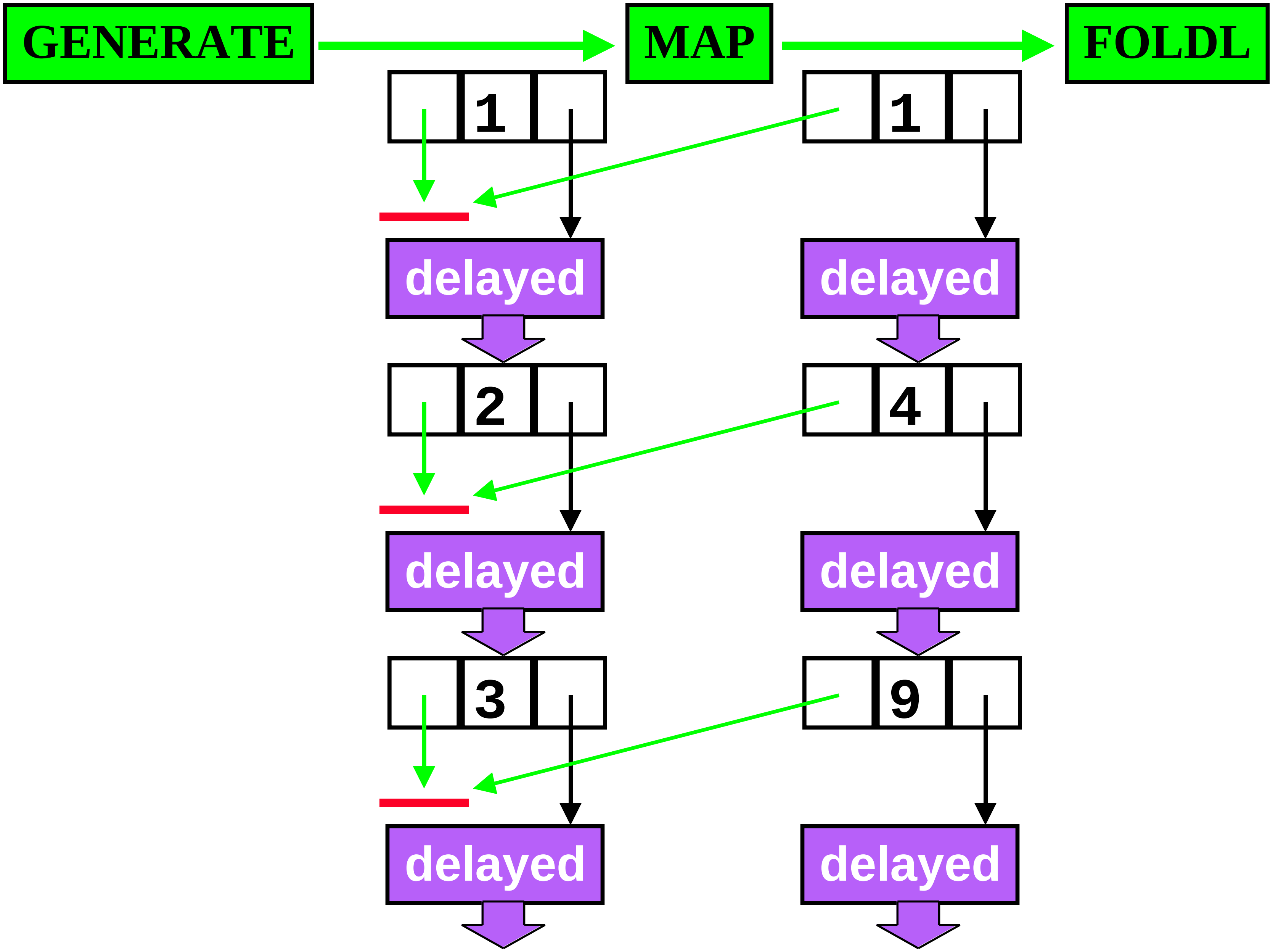
Modular `average` (eager lists)



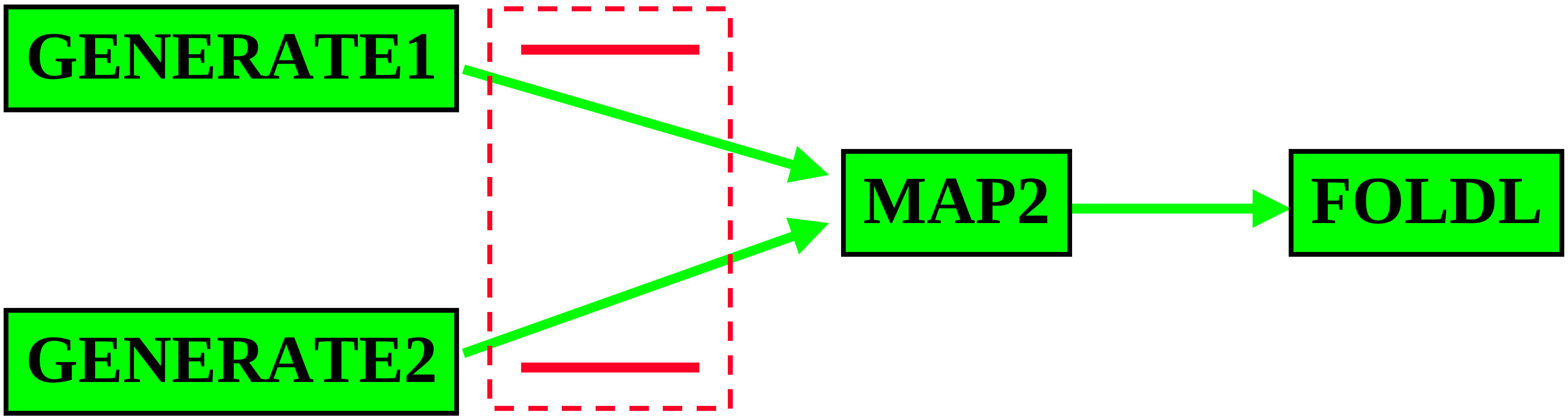
Modular `average` (synchronized lazy lists)



Synchronized Lazy Lists



`simul` needed for fan-in



Synchronized Lazy Aggregates

- We have developed a suite of routines for modularizing list and tree algorithms.
- Can modularize recursive as well as iterative list algorithms by using two synchrons per node (one down, one up).
- Can handle tail-recursion in a modular way by extending synchrons.
- Can modularize recursive tree algorithms in same fashion.
- Can handle some forms of filtering.

Overview

- What is a Synchronization Barrier?
- Dimensions of Barriers
- Synchrons: First-Class Barriers with a Variable Number of Participants
- Motivating Example:
  Space-Efficient Aggregate Data Programs
- Discussion

Implementation Strategies

- Can implement terms of object finalization, but then every rendezvous requires full GC.
- Our prototype implementation uses reference counts to reduce rendezvous costs.
- Ripe area for static analysis:
  - Use types to reduce reference count costs
  - Completely remove some synchrons
  - Convert some synchrons to `barrier/pause`.

Related Work

- Hughes's `par/synch` non-modular
- Wadler's Listlessness iterative lists only
- Wadler's Deforestation fan-out, fan-in problems
- Waters's Series iterative lists only
- Other Transformations no guarantees, ad hoc

- Wadler's Fixing a Space Leak with GC.
- Other GC-dependent language mechanisms:
  - object finalization
  - weak pointers
  - reference counting cells (Espinosa)

Future Directions

- Goal: Express space-efficient algorithms in modular way.
- Synchron semantics.
- More efficient synchron implementations.
- Static analysis to remove synchrons.
- Static deadlock detection.
- More idioms to encapsulate synchrons.
- Other mechanisms for space-efficient modular program decompositions.

Conclusion

- Synchrons (+ concurrency) are first mechanism to support space-efficient aggregate data programs in modular fashion.
- Need better idioms for GC-dependent language mechanisms.
- Need better techniques for reasoning about space.
- Need better techniques for modularizing space-efficient algorithms.