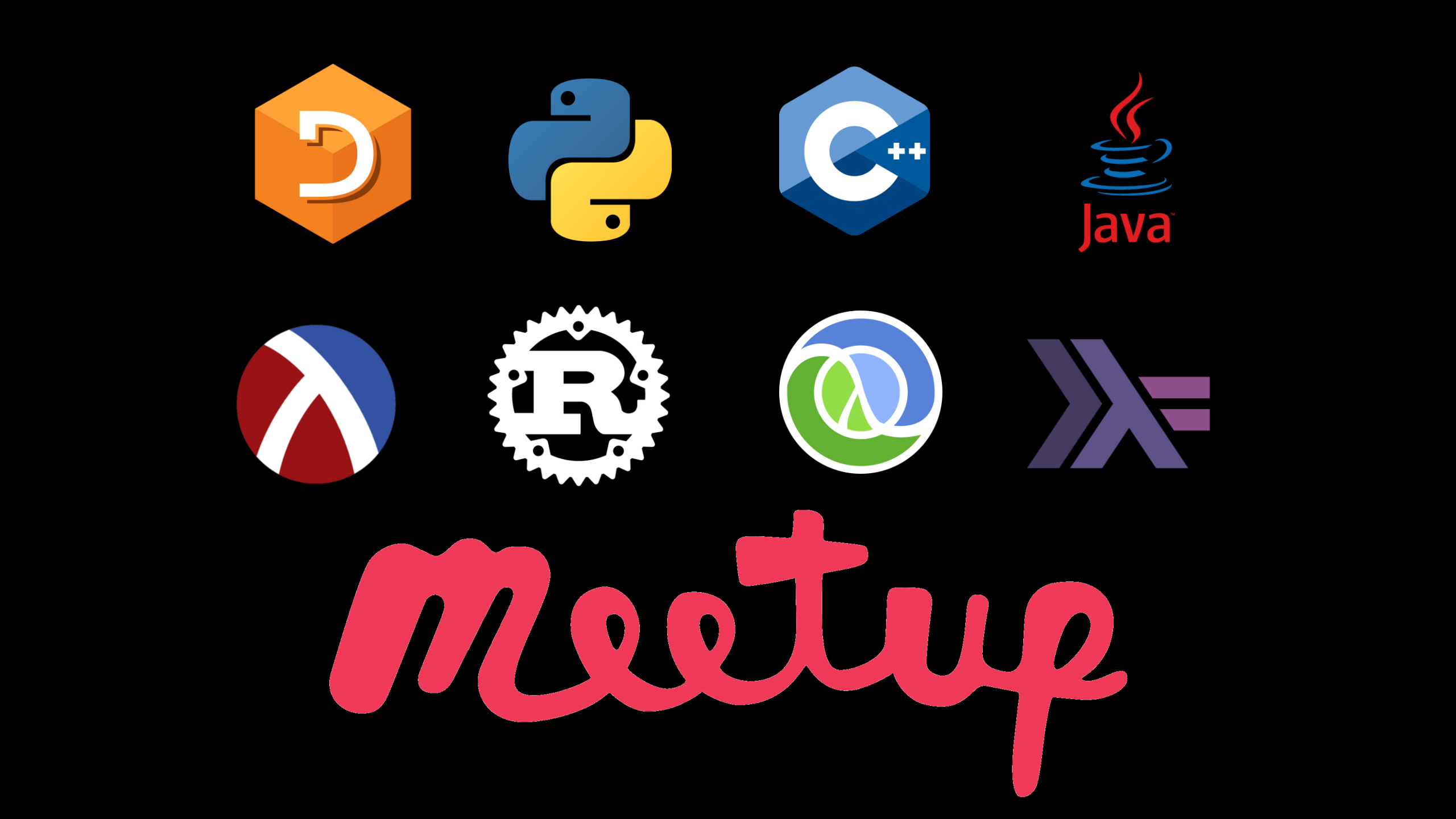
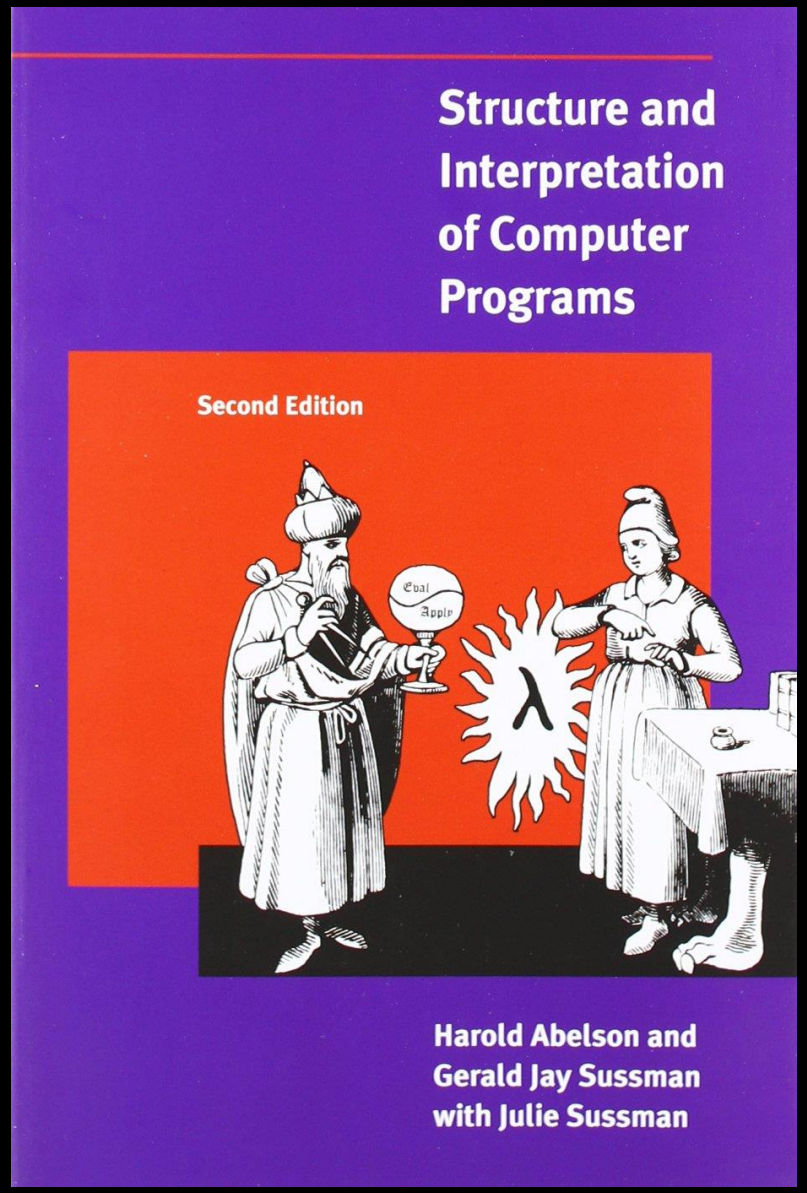

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 3.4

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

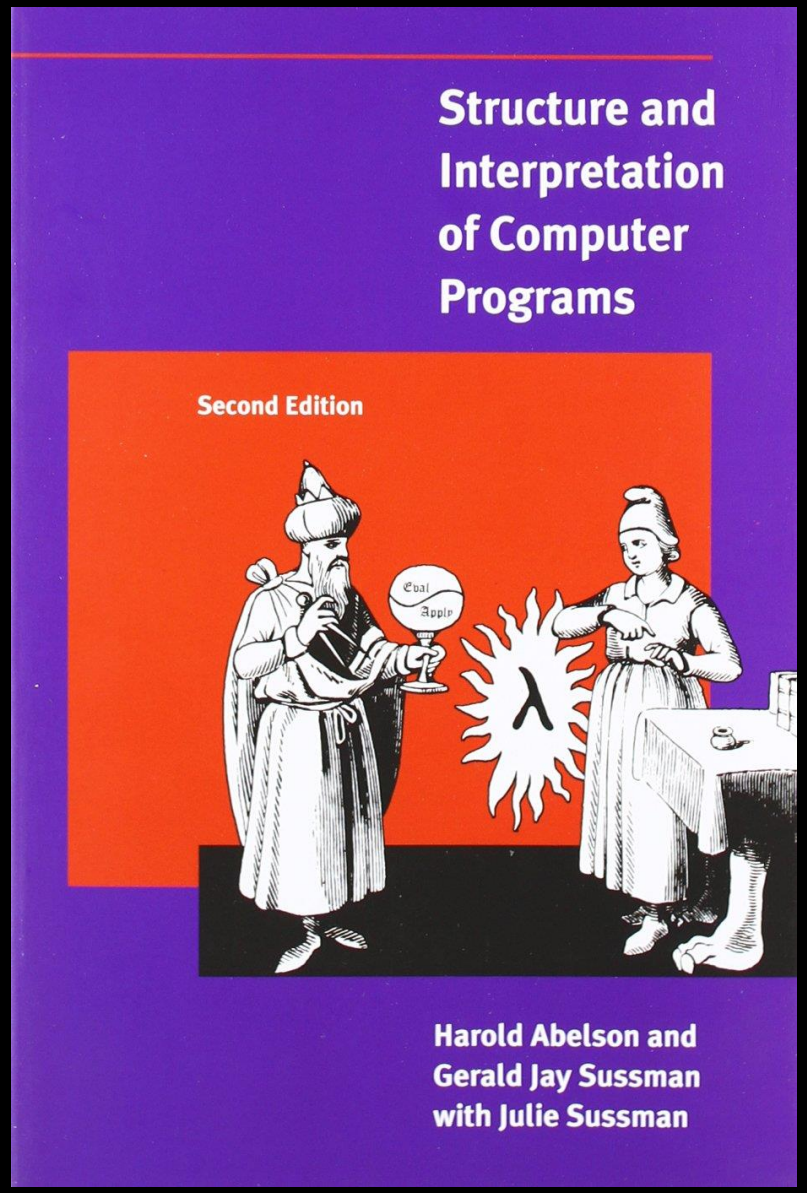
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 3.4

We've seen the power of computational objects with local state as tools for modeling. Yet, as Section 3.1.3 warned, this power extracts a price: the loss of referential transparency, giving rise to a thicket of questions about sameness and change, and the need to abandon the substitution model of evaluation in favor of the more intricate environment model.

The central issue lurking beneath the complexity of state, sameness, and change is that by introducing assignment we are forced to admit *time* into our computational models. Before we introduced assignment, all our programs were timeless, in the sense that any expression that has a value always has the same value.

Here successive evaluations of the same expression yield different values. This behavior arises from the fact that the execution of assignment statements (in this case, assignments to the variable `balance`) delineates *moments in time* when values change. The result of evaluating an expression depends not only on the expression itself, but also on whether the evaluation occurs before or after these moments. Building models in terms of computational objects with local state forces us to confront time as an essential concept in programming.

On the surface, time seems straightforward. It is an ordering imposed on events.[35] For any events A and B, either A occurs before B, A and B are simultaneous, or A occurs after B. For instance, returning to the bank account example, suppose that Peter withdraws \$10 and Paul withdraws \$25 from a joint account that initially contains \$100, leaving \$65 in the account. Depending on the order of the two withdrawals, the sequence of balances in the account is either \$100 $\rightarrow$ \$90 $\rightarrow$ \$65 or \$100 $\rightarrow$ \$75 $\rightarrow$ \$65 . In a computer implementation of the banking system, this changing sequence of balances could be modeled by successive assignments to a variable `balance`.

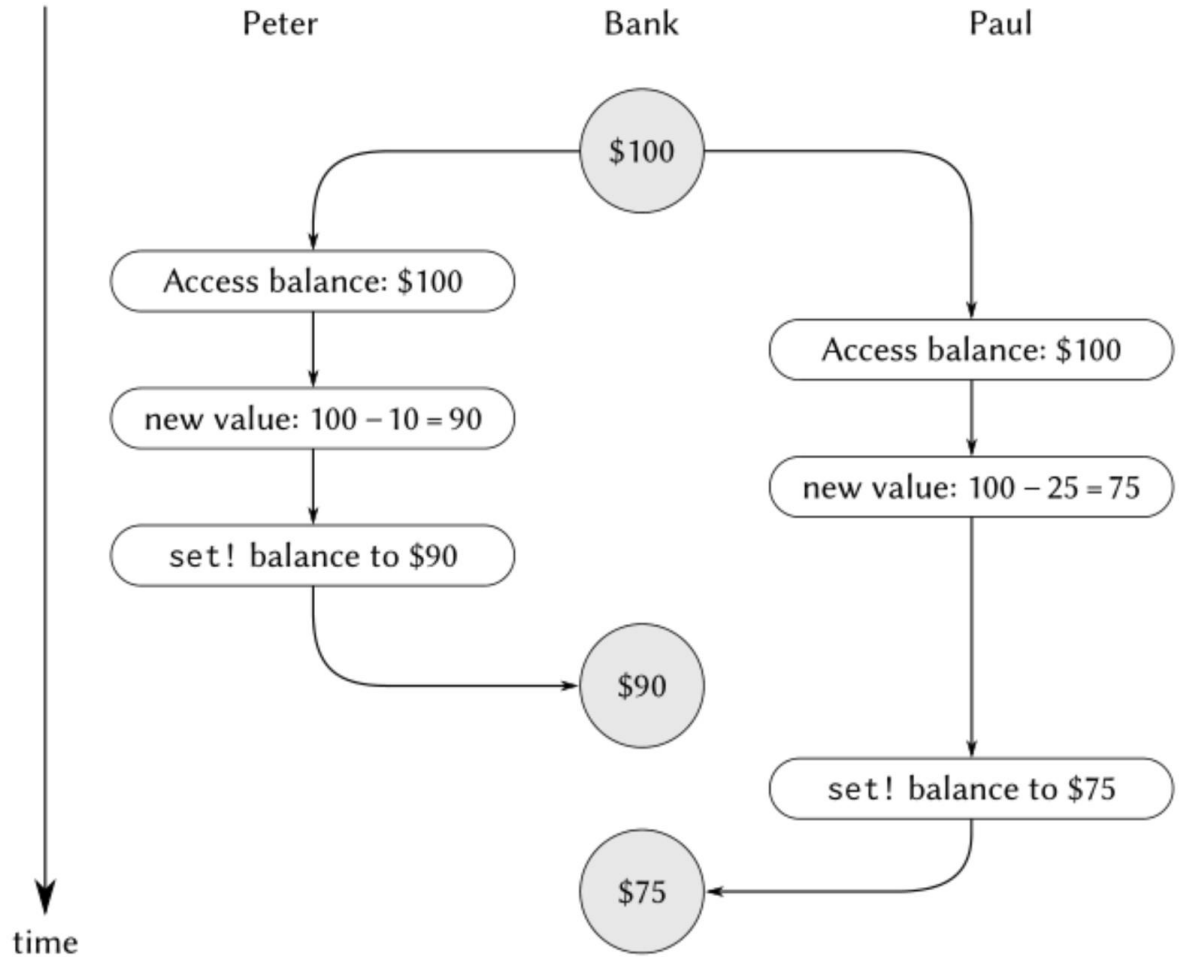


Figure 3.29: Timing diagram showing how interleaving the order of events in two banking withdrawals can lead to an incorrect final balance.

Correct behavior of concurrent programs

The above example typifies the subtle bugs that can creep into concurrent programs. The root of this complexity lies in the assignments to variables that are shared among the different processes. We already know that we must be careful in writing programs that use `set!`, because the results of a computation depend on the order in which the assignments occur.[37] With concurrent processes we must be especially careful about assignments, because we may not be able to control the order of the assignments made by the different processes. If several such changes might be made concurrently (as with two depositors accessing a joint account) we need some way to ensure that our system behaves correctly. For example, in the case of withdrawals from a joint bank account, we must ensure that money is conserved. To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.

One possible restriction on concurrency would stipulate that no two operations that change any shared state variables can occur at the same time. This is an extremely stringent requirement.

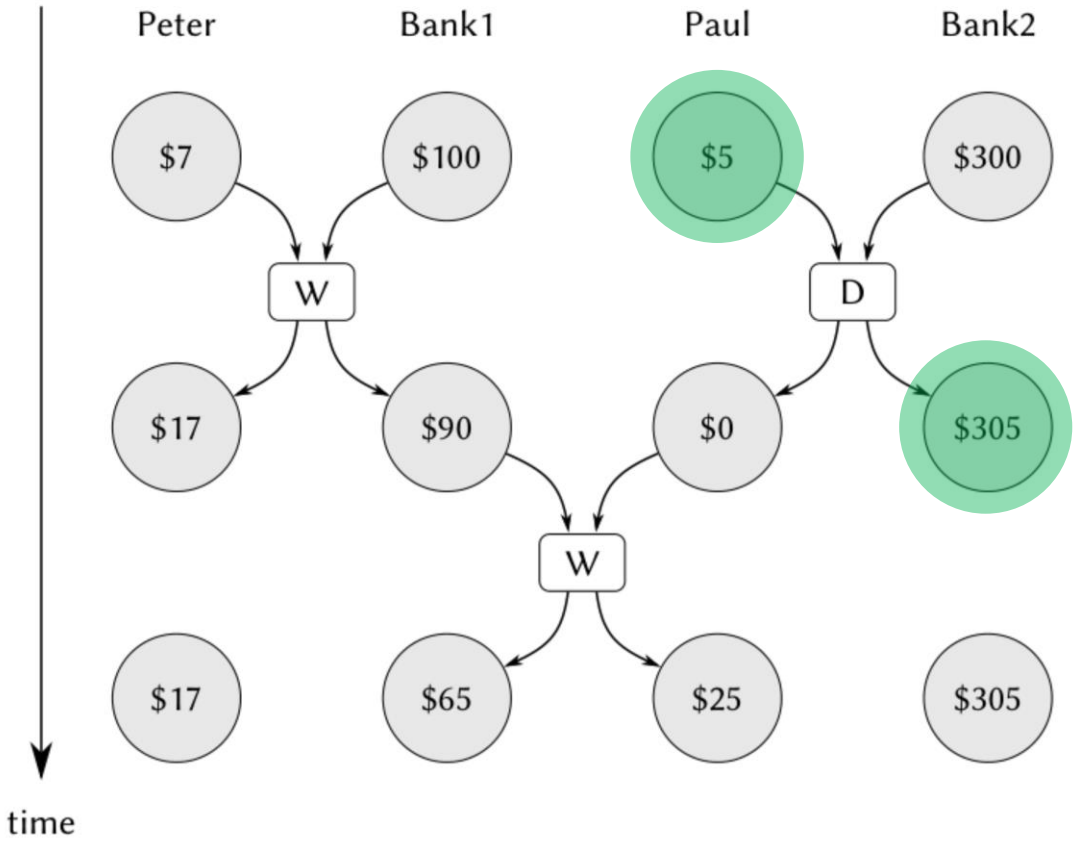


Figure 3.30: Concurrent deposits and withdrawals from a joint account in Bank1 and a private account in Bank2.

A less stringent restriction on concurrency would ensure that a concurrent system produces the same result as if the processes had run sequentially in some order.

Exercise 3.38: Suppose that Peter, Paul, and Mary share a joint bank account that initially contains $100. Concurrently, Peter deposits $10, Paul withdraws $20, and Mary withdraws half the money in the account, by executing the following commands:

```
Peter: (set! balance (+ balance 10))
Paul:  (set! balance (- balance 20))
Mary:  (set! balance (- balance (/ balance 2)))
```

a. List all the different possible values for `balance` after these three transactions have been completed, assuming that the banking system forces the three processes to run sequentially in some order.

b. What are some other values that could be produced if the system allows the processes to be interleaved? Draw timing diagrams like the one in Figure 3.29 to explain how these values can occur.

```
;; Exercise 3.38 (page 409-10)
```

a)

```
10+÷∘2⊣¯20+100 ⍝ 50
÷∘2⊣¯20+10+100 ⍝ 45
¯20+10+÷∘2⊣100 ⍝ 40
¯20+÷∘2⊣10+100 ⍝ 35
```

b) worst case would be:

```
¯20+÷∘2⊣100 ⍥ 30
```

Serializing access to shared state

Serialization implements the following idea: Processes will execute concurrently, but there will be certain collections of procedures that cannot be executed concurrently. More precisely, serialization creates distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time. If some procedure in the set is being executed, then a process that attempts to execute any procedure in the set will be forced to wait until the first execution has finished.

```
(define x 10)
(parallel-execute
 (lambda () (set! x (* x x)))
 (lambda () (set! x (+ x 1))))
```

This creates two concurrent processes—P_1, which sets x to x times x, and P_2, which increments x. After execution is complete, x will be left with one of five possible values, depending on the interleaving of the events of P_1 and P_2:

101: P_1 sets x to 100 and then P_2 increments x to 101.
121: P_2 increments x to 11 and then P_1 sets x to x * x.
110: P_2 changes x from 10 to 11 between the two times that P_1 accesses the value of x during the evaluation of `(* x x)`.
11: P_2 accesses x, then P_1 sets x to 100, then P_2 sets x.
100: P_1 accesses x (twice), then P_2 sets x to 11, then P_1 sets x.

```
(define s (make-serializer))
(parallel-execute
 (s (lambda () (set! x (* x x))))
 (s (lambda () (set! x (+ x 1)))))
```

can produce only two possible values for x, 101 or 121.

Exercise 3.39: Which of the five possibilities in the parallel execution shown above remain if we instead serialize execution as follows:

```
(define x 10)
(define s (make-serializer))
(parallel-execute
 (lambda () (set! x ((s (lambda () (* x x))))))
 (s (lambda () (set! x (+ x 1)))))
```

```
;; Exercise 3.39 (page 414)

;; YES -> 101: P 1 sets x to 100 and then P 2 increments x to 101.
;; YES -> 121: P 2 increments x to 11 and then P 1 sets x to x * x .
;; NO  -> 110: P 2 changes x from 10 to 11 between the two times that
;;             P 1 accesses the value of x during the evaluation of (* x x) .
;; YES -> 11:  P 2 accesses x , then P 1 sets x to 100, then P 2 sets x . -> lambda could eval (* x x) and then +1
;; NO  -> 100: P 1 accesses x (twice), then P 2 sets x to 11, then P 1 sets x .
```

Exercise 3.40: Give all possible values of `x` that can result from executing

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (* x x x))))
```

Which of these possibilities remain if we instead use serialized procedures:

```
(define x 10)
(define s (make-serializer))
(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (* x x x)))))
```

```
;; Exercise 3.40 (page 414)

;; a) 100 1K 10K 100K 1M
;; b) 1M
```

Exercise 3.41: Ben Bitdiddle worries that it would be better to implement the bank account as follows (where the commented line has been changed):

because allowing unserialized access to the bank balance can result in anomalous behavior. Do you agree? Is there any scenario that demonstrates Ben's concern?

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance
                     (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance)
             ((protected
               (lambda () balance)))) ; serialized
            (else
             (error "Unknown request: MAKE-ACCOUNT"
                    m))))
    dispatch))
```

```
;; Exercise 3.41 (page 414-5)

;; Not a mutator so no need to serialize
```

Exercise 3.42: Ben Bitdiddle suggests that it's a waste of time to create a new serialized procedure in response to every `withdraw` and `deposit` message. He says that `make-account` could be changed so that the calls to `protected` are done outside the `dispatch` procedure. That is, an account would return the same serialized procedure (which was created at the same time as the account) each time it is asked for a withdrawal procedure.

Is this a safe change to make? In particular, is there any difference in what concurrency is allowed by these two versions of `make-account`?

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance) balance)
            (else (error "Unknown request: MAKE-ACCOUNT"
                         m))))
    dispatch))
```

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (let ((protected-withdraw (protected withdraw))
          (protected-deposit (protected deposit)))
      (define (dispatch m)
        (cond ((eq? m 'withdraw) protected-withdraw)
              ((eq? m 'deposit) protected-deposit)
              ((eq? m 'balance) balance)
              (else
               (error "Unknown request: MAKE-ACCOUNT"
                      m))))
      dispatch)))
```

```
;; Exercise 3.42 (page 415-6)

;; They extra let expression will create an additional frame in the environment model
;; - but other than that they are the same
```

Implementing serializers

We implement serializers in terms of a more primitive synchronization mechanism called a *mutex*. A mutex is an object that supports two operations—the mutex can be *acquired*, and the mutex can be *released*.

[44]The term "mutex" is an abbreviation for *mutual exclusion*. The general problem of arranging a mechanism that permits concurrent processes to safely share resources is called the mutual exclusion problem. Our mutex is a simple variant of the *semaphore* mechanism (see Exercise 3.47), which was introduced in the "THE" Multiprogramming System developed at the Technological University of Eindhoven and named for the university's initials in Dutch (Dijkstra 1968a). The acquire and release operations were originally called P and V, from the Dutch words *passeren* (to pass) and *vrijgeven* (to release), in reference to the semaphores used on railroad systems. Dijkstra's classic exposition (Dijkstra 1968b) was one of the first to clearly present the issues of concurrency control, and showed how to use semaphores to handle a variety of concurrency problems.

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))
      serialized-p)))
```

```
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire))) ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex))

(define (clear! cell) (set-car! cell false))

(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

Exercise 3.47: A semaphore (of size n) is a generalization of a mutex. Like a mutex, a semaphore supports acquire and release operations, but it is more general in that up to n processes can acquire it concurrently. Additional processes that attempt to acquire the semaphore must wait for release operations. Give implementations of semaphores

a. in terms of mutexes

```
;; Exercise 3.47 a) (page 425)

;; original (incorrect) solution

(define (make-semaphore n)
  (let ((count 0))
    (define (the-semaphore msg)
      (cond ((eq? msg 'acquire)
             (if (>= count n)
                 (the-semaphore 'acquire)
                 (set! count (+ count 1))))
            ((eq? msg 'release)
             (set! (count (- count 1))))))
    the-semaphore))
```

```
;; modified (correct) solution

(define (make-semaphore n)
  (let ((count 0)
        (lock (make-mutex)))
    (define (the-semaphore msg)
      (cond ((eq? msg 'acquire)
             (lock 'acquire)
             (if (>= count n)
                 (begin (lock 'release) (the-semaphore 'acquire))
                 (begin (set! count (+ count 1) (lock 'release)))))
            ((eq? msg 'release)
             (lock 'aquire)
             (set! (count (- count 1)))
             (lock 'release))))
    the-semaphore))
```

Deadlock

[48]The general technique for avoiding deadlock by numbering the shared resources and acquiring them in order is due to Havender (1968). Situations where deadlock cannot be avoided require *deadlock-recovery* methods, which entail having processes "back out" of the deadlocked state and try again. Deadlock-recovery mechanisms are widely used in database management systems, a topic that is treated in detail in Gray and Reuter 1993.

Mechanisms such as `test-and-set!` require processes to examine a global shared flag at arbitrary times. This is problematic and inefficient to implement in modern high-speed processors, where due to optimization techniques such as pipelining and cached memory, the contents of memory may not be in a consistent state at every instant. In contemporary multiprocessing systems, therefore, the serializer paradigm is being supplanted by new approaches to concurrency control.[49]

[49] One such alternative to serialization is called *barrier synchronization*. The programmer permits concurrent processes to execute as they please, but establishes certain synchronization points ("barriers") through which no process can proceed until all the processes have reached the barrier. Modern processors provide machine instructions that permit programmers to establish synchronization points at places where consistency is required. The PowerPC, for example, includes for this purpose two instructions called SYNC and EIEIO (Enforced In-order Execution of Input/Output).

std::barrier

Defined in header <barrier>

```
template<class CompletionFunction = /* see below */>
class barrier;
```

(since C++20)

The class template `std::barrier` provides a thread-coordination mechanism that allows at most an expected number of threads to block until the expected number of threads arrive at the barrier. Unlike `std::latch`, barriers are reusable: once the arriving threads are unblocked from a barrier phase's synchronization point, the same barrier can be reused.

A barrier object's lifetime consists of a sequence of barrier phases. Each phase defines a *phase synchronization point*. Threads that arrive at the barrier during the phase can block on the phase synchronization point by calling `wait`, and will be unblocked when the phase completion step is run.

A *barrier phase* consists following steps:

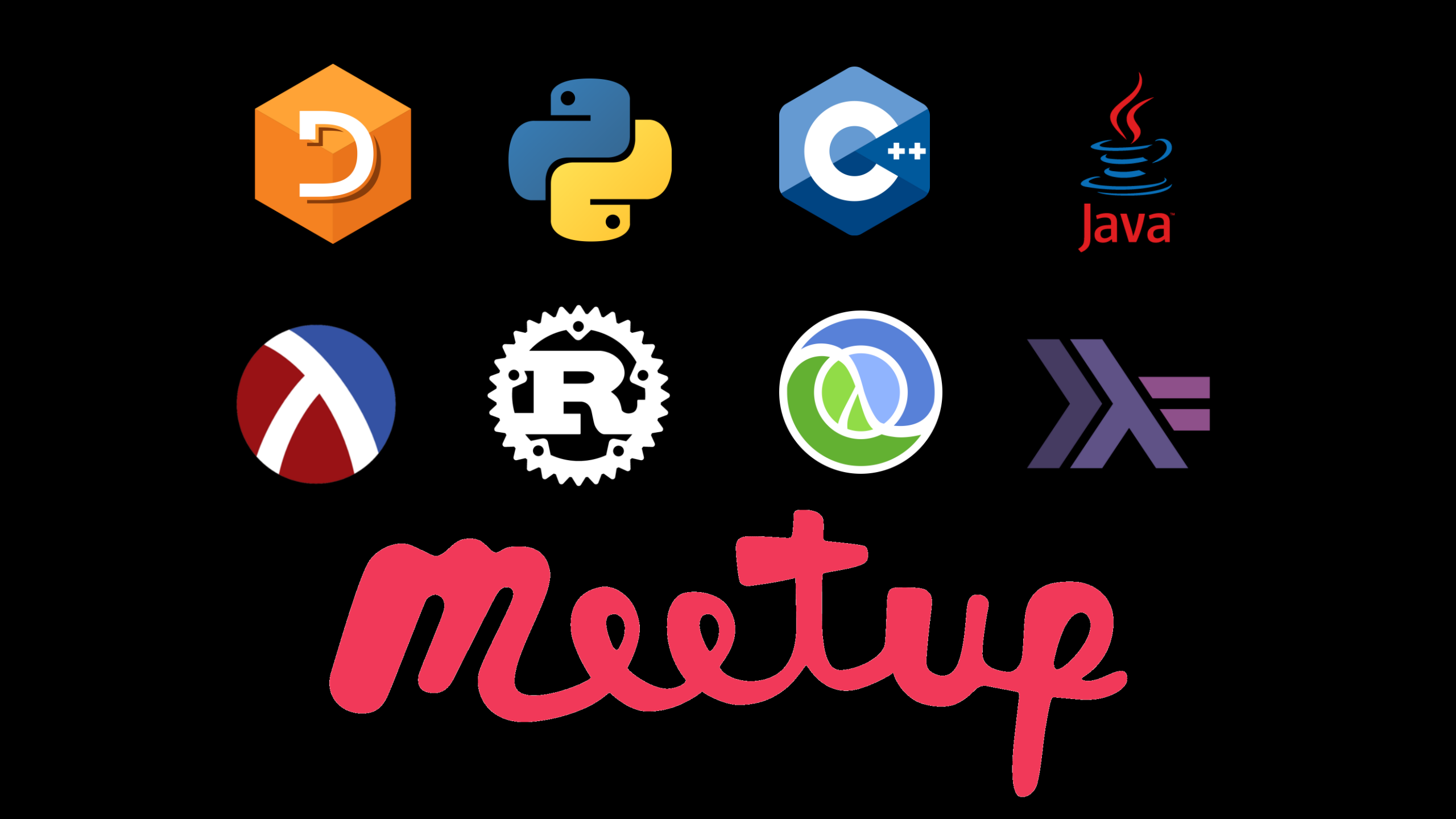
1. The *expected count* is decremented by each call to `arrive` or `arrive_and_drop`.
2. When the expected count reaches zero, the *phase completion step* is run. The completion step invokes the completion function object, and unblocks all threads blocked on the phase synchronization point. The end of the completion step strongly happens-before the returns from all calls that were unblocked by the completion step.
   - For the specialization `std::barrier<>` (using the default template argument), the completion step is run as part of the call to `arrive` or `arrive_and_drop` that caused the expected count to reach zero.
   - For other specializations, the completion step is run on one of the threads that arrived at the barrier during the phase. And the behavior is undefined if any of the barrier object's member functions other than `wait` are called during the completion step.
3. When the completion step finishes, the expected count is reset to the value specified in construction, possibly adjusted by calls to `arrive_and_drop`, and the next phase starts.

Concurrent invocations of the member functions of `barrier`, except for the destructor, do not introduce data races.

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman


Java
meetup