Concurrency and Time/State

Introducing mutation (e.g. `set!`) into our language forces us to confront what we mean by equality and change.

An example of a simple procedure that is referentially transparent:

An example of a simple procedure involving mutation that is NOT referentially transparent:

Mutation has introduced issues of `time` directly into our language.

Consider two withdrawals from a joint bank account. Sketch below an example of why concurrent procedures can cause problems:
Possible restrictions on concurrent programming that will fix the problem of accessing shared variables:

**Serialization**

Suppose we extend Scheme to include a procedure called `parallel-execute`:

```
(parallel-execute p₁ p₂... pₙ)
```

Each `p` must be a procedure of no arguments. `Parallel-execute` creates a separate process for each `p`, which applies `p` (to no arguments). These processes all run concurrently.

As an example of how this is used, consider

```
(define x 10)

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

Here are the possible outcomes

- 101: `P₁` sets `x` to 100 and then `P₂` increments `x` to 101.
- 121: `P₂` increments `x` to 11 and then `P₁` sets `x` to `x` times `x`.
- 110: `P₂` changes `x` from 10 to 11 between the two times that `P₁` 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 \( x \).

But with serialization

\[
\text{(define } x \ 10) \\
\text{(define } s \ (\text{make-serializer})) \\
\text{(parallel-execute} \\
\quad (s \ (\lambda () \ (\text{set! } x \ (* \ x \ x)))) \\
\quad (s \ (\lambda () \ (\text{set! } x \ (+ \ x \ 1))))
\]

\text{can produce only two possible values for } x, 101 \text{ or 121. The other possibilities are eliminated, because the execution of } P_1 \text{ and } P_2 \text{ cannot be interleaved.}

We can fix our bank account example:

\[
\text{(define } (\text{make-account } \text{balance})) \\
\text{(define } (\text{withdraw amount})) \\
\quad \text{(if } (> = \text{balance amount}) \\
\qquad \text{(begin} \ (\text{set! balance } (- \text{balance amount})) \\
\qquad \text{balance})) \\
\quad \text{"Insufficient funds"}) \\
\text{(define } (\text{deposit amount})) \\
\text{(set! balance } (+ \text{balance amount})) \\
\text{balance}) \\
\text{(let } ((\text{protected } (\text{make-serializer}))) \\
\text{(define } (\text{dispatch } m)) \\
\quad (\text{cond} \ ((\text{eq? } m \ '\text{withdraw}) \\
\qquad (\text{protected withdraw})) \\
\qquad ((\text{eq? } m \ '\text{deposit}) \\
\qquad (\text{protected deposit})) \\
\qquad ((\text{eq? } m \ '\text{balance}) \text{balance}) \\
\quad \text{(else } (\text{error } \text{"Unknown request} \\
\qquad \text{-- MAKE-ACCOUNT"} \\
\qquad m)))))) \\
\text{dispatch})
\]

A procedure to swap balances in two accounts

\[
\text{(define } (\text{exchange account1 account2})) \\
\text{(let } ((\text{difference } (- \text{account1 } \text{balance}) \\
\quad (\text{account2 } \text{balance}))) \\
\quad ((\text{account1 } \text{withdraw} \text{difference}) \\
\qquad ((\text{account2 } \text{deposit} \text{difference}))
\]

Suppose Paul swaps \( a_1 \) and \( a_2 \) at the same time that Peter swaps \( a_1 \) and \( a_3 \).

Peter might compute difference between \( a_1 \) and \( a_2 \) but then Paul might change the balance in \( a_1 \) before Peter is able to complete the exchange.

So instead we can export a serializer:
(define (make-account-with-serializer balance)
  (define (withdraw amount)
    (if (> amount balance)
        (begin (set! balance (- balance amount))
              "Insufficient funds")))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer)
              balance-serializer)
            (else (error "Unknown request -- MAKE-ACCOUNT" m))))))

Now each user must explicitly manage serialization.

(define (deposit account amount)
  (let ((s (account 'serializer))
        (d (account 'deposit))
        ((s d) amount)))

But exchanging is now straightforward.

(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer))
        ((serializer1 (serializer2 exchange))
         account1
         account2)))

An implementation of a serializer:

(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)))
            (eq? m 'release) (clear! cell)))))
(define (clear! cell)
  (set-car! cell false))

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