MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

**Lecture Notes – October 29, 1996**

## 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 ... pₖ)
```

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: $P_1$ 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` times `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`.

But with serialization

```
(define x 10)

(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. The other possibilities are eliminated, because the execution of $P_1$ and $P_2$ cannot be interleaved.

We can fix our bank account example:

```
(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))
```

A procedure to swap balances in two accounts

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                       (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

Suppose Paul swaps $a1$ and $a2$ at the same time that Peter swaps $a1$ and $a3$.

Peter might compute difference between $a1$ and $a2$ but then Paul might change the balance in $a1$ 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 (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "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))))
    dispatch))
```

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)))  ; 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)))
```