(Concurrent Imp

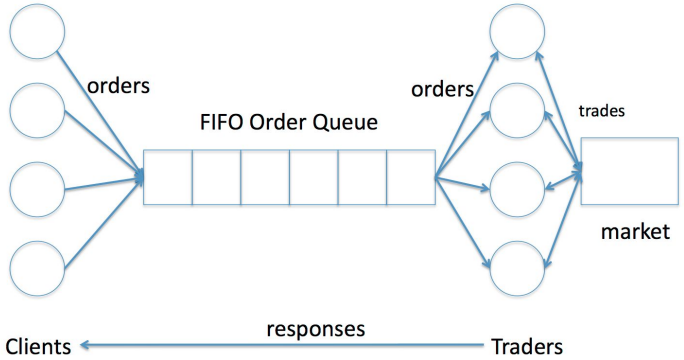
(Imp but Concurrent))

Harlan, Zach

Background and Review

- TA-ing CS170 Operating Systems
  - Focuses on concurrent designs, using mutexes, condition variables, and semaphores
  - Personal interest in how you reason about (a)synchronous/parallel behavior
- In the Programming Languages Lab (with Ben Hardekopf)
- Synchronization toolbox consists of a few “primitives”
  - while loops: code spins while waiting for stuff to become true, available, etc.
  - locks/mutexes: mutual exclusion scheme where no two entities may access the same code simultaneously
  - condition variables: augment mutexes, a way around while loops, jump off CPU until entity can access a mutex’ed region
  - semaphores: integer counters with two operations P (decrement) and V (increment), entities P/V the semaphore to indicate its availability; if counter < 0, block

Example Problem (mutexes)




```
        /*
         * queue it for the traders
         */
        queued = 0;
        while(queued == 0) {
                pthread_mutex_lock(&(ca->order_que->lock));
                next = (ca->order_que->head + 1) % ca->order_que->size;
                /*
                 * is the queue full?
                 */
                if(next == ca->order_que->tail) {
                        pthread_mutex_unlock(&(ca->order_que->lock));
                        continue;
                }

                /*
                 * there is space in the queue, add the order and bump
                 * the head
                 */
                if(ca->verbose == 1) {
                        now = CTimer();
                        printf("%10.0f client %d: ",now,ca->id);
                        printf("queued stock %d, for %d, %s\n",
                                order->stock_id,
                                order->quantity,
                                (order->action ? "SELL" : "BUY"));
                }

                ca->order_que->orders[next] = order;
                ca->order_que->head = next;
                queued = 1;
                pthread_mutex_unlock(&(ca->order_que->lock));

                /*
                 * spin waiting until the order is fulfilled
                 */
                while(order->fulfilled == 0);

                /*
                 * done, free the order and repeat
                 */
                FreeOrder(order);
        }
}
```

Problem Statement (Expectation)

1. “Synthesize C code with semaphores for some user-written specification”
    a. Inspired by *Combinatorial sketching for finite programs* and other synthesis tasks
    b. User spec would be some DSL fed into Rosette
    c. Semaphores can be hard to get right but are often cleaner
2. Project Goals & Requirements
    a. DSL needs a notion of concurrency, be it threads etc.
    b. Needs a notion of locking, so the user can tame the concurrency
    c. “Higher-order functions” to express what parts of the spec (intended code) are to be atomic
    d. Parser to turn DSL into C code
    e. Play with performance on class assignments, Dining Philosophers, etc.

Problem Progress (Reality)

- Really only made it to about step 2b/c
- Turns out taking the classic, imperative, language Imp from a toy language that does some boolean and arithmetic computations to a toy language that does those computations in threads is non-trivial
- Most of our time was spent on continually revising the original DSL to get closer to a concurrent Imp

A Concurrent Imperative DSL

Arithmetic Expression	e	$::=$	$e + e \mid e - e \mid e * e \mid v \mid n$
Boolean Expression	b	$::=$	$\neg\, b \mid b \wedge b \mid e = e \mid e \leq e \mid \text{true} \mid \text{false}$
Command	c	$::=$	$v := e \mid \text{load } g\ v \mid \text{store } v\ g$
		$\mid$	$\text{if } b \text{ then } c \text{ else } c \mid \text{while } b \text{ do } c$
		$\mid$	$c\ ;\ c \mid \text{atomic } c \mid \text{par } c \mid \text{skip} \mid \text{halt}$
Local Variables	v	$\in$	$\mathcal{V}$
Global Variables	g	$\in$	$\mathcal{G}$
Integer	n	$\in$	Z

Table 1: Concurrent Imp DSL.

How to take Imp to the next level (1)

```
(define (interpret p st)
  (match p
    [(id x)       (hash-ref st x)]
    [(plus a b)   (+ (interpret a st)       (interpret b st))]
    [(mul a b)    (* (interpret a st)       (interpret b st))]
    [(square a)   (expt (interpret a st)                2)]

    [(top) #t]
    [(bottom) #f]
    [(eq a0 a1)   (= (interpret a0 st)      (interpret a1 st))]
    [(leq a0 a1)  (<= (interpret a0 st)     (interpret a1 st))]
    [(non b)      (not (interpret b st))]
    [(et b0 b1)   (and (interpret b0 st))   (interpret b1 st)]
    [(ou b0 b1)   (or (interpret b0 st)     (interpret b1 st))]

    [(skip) st]
    [(assign var val)     (update-hash-table st var (interpret val st))]
    [(ite b c0 c1)   (if (interpret b st) (interpret c0 st) (interpret c1 st))]
    [(follows c0 c1) (let ([st1 (interpret c0 st)]) (interpret c1 st1))]

    ; allows non-terminating functions, could unroll only finitely many times
    [(while b c)   (if (interpret b st)
                       (interpret (follows c (while b c)) st)
                       st)]

    [_ p]))
```

How to take Imp to the next level (2)

- CEK is an “Abstract machine”
  a. (C)ode (E)nvironment (K)ontinuation
  b. Behave sort of like registers
  c. Each a stack, information pushed and popped between them

```
;; step function
(define (step st)
  (match st

    ;; values
    [(state (list-rest (? number? n) c) e s k)          (state c e s (cons n k))]
    [(state (list-rest (? boolean? b) c) e s k)         (state c e s (cons b k))]

    ;; arithmetic operations
    [(state (list-rest (add E1 E2) c) e s k)            (state (cons E1 (cons E2 (cons '+ c))) e s k)]
    [(state (list-rest (sub E1 E2) c) e s k)            (state (cons E1 (cons E2 (cons '- c))) e s k)]
    [(state (list-rest (mul E1 E2) c) e s k)            (state (cons E1 (cons E2 (cons '* c))) e s k)]

    ;; while B do C
    [(state (list-rest (while B C) c) e s k)                        (state (cons B (cons 'while c)) e s (cons B (cons C k)))]
    [(state (list-rest 'while c) e s (cons #t (cons B (cons C k))))  (state (cons C (cons (while B C) c)) e s k)]
    [(state (list-rest 'while c) e s (cons #f (cons B (cons C k))))  (state c e s k)]
```

Par and Atomic

1. Par is a custom struct in our DSL that signals to the interpreter it is within a parallel discipline and thus may choose non-deterministically which of a set of branches it can step next
2. Unless the code is wrapped in an Atomic struct in which case the interpreter is forced to run all wrapped code before returning to the parallel discipline

$$[par^{1}_{sos}] \quad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1 \ \texttt{par} \ S_2, s\rangle \Rightarrow \langle S_1' \ \texttt{par} \ S_2, s'\rangle}$$

$$[par^{2}_{sos}] \quad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \ \texttt{par} \ S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$[par^{3}_{sos}] \quad \frac{\langle S_2, s\rangle \Rightarrow \langle S_2', s'\rangle}{\langle S_1 \ \texttt{par} \ S_2, s\rangle \Rightarrow \langle S_1 \ \texttt{par} \ S_2', s'\rangle}$$

$$[par^{4}_{sos}] \quad \frac{\langle S_2, s\rangle \Rightarrow s'}{\langle S_1 \ \texttt{par} \ S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}$$

Rosette Integration

1. Modeled off first homework assignment
2. We would like to:
   a. Solve: obtain the answers to computations internal to our threaded code, e.g. if a thread computes a sum, have Rosette compute that self-same sum
   b. Verify: assuming I know a priori the result of a threaded computation, replace the variables in question with symbolic vars and verify the result is possible
   c. Synthesize: replace chunks of my code with holes (??) and have Rosette fill them in
3. Synthesis is much harder because our DSL at this point is relatively complicated and doesn't have a clear, nearby logic. Whereas Imp by itself is very close to the logics of unquantified boolean expressions and linear integer arithmetic, our DSL is based on lots of commands, mutations, and concurrency

Future Steps

1. Synthesis
    a. Synthesize some integer, boolean expressions
    b. Synthesize the placement of an atomic region
    c. Try to explore the space of how Rosette does the above, where it stops
    d. Formal semantics a la E.A. Emerson’s region graphs + Hoare logic
2. C code generation
    a. Assuming a tighter DSL/semantics, “machine translate” that to C code

```
par C1 C2 with G = { A -> 5, B -> 50 }

C1 = load A a ;
     a := a + 1 ;
     store a A ;
     atomic (
         load A a ;
         a := a + 1 ;
         store a A ;
         load B b ;
         b := b + 1 ;
         store b B ;
    ) ;
    halt

C2 = atomic (
         load A a ;
         a := a + 1 ;
         store a A ;
         load B b ;
         b := b + 1 ;
         store b B ;
    ) ;
    halt
```

Figure 1: Example concurrent program.

```
par C1 C2 with G = { A -> 5, B -> 50 }

C1 = load A a ;
     a := a + $x ;
     store a A ;
     atomic (
         load A a ;
         a := a + $x ;
         store a A ;
         load B b ;
         b := b + $y ;
         store b B ;
    ) ;
    halt

C2 = atomic (
         load A a ;
         a := a + $x ;
         store a A ;
         load B b ;
         b := b + $y ;
         store b B ;
    ) ;
    halt
```

Figure 2: Example concurrent program with symbolic values `$x` and `$y`.

```
par C1 C2 with G = { A -> $x, B -> $y }

C1 = load A a ;
     a := a + 1 ;
     store a A ;
     atomic (
         load A a ;
         a := a + 1 ;
         store a A ;
         load B b ;
         b := b + 1 ;
         store b B ;
    ) ;
    halt

C2 = atomic (
         load A a ;
         a := a + 1 ;
         store a A ;
         load B b ;
         b := b + 1 ;
         store b B ;
    ) ;
    halt
```

Figure 3: Example concurrent program with symbolic values `$x` and `$y`.