Lecture 10: Mostly Recap Keystone examples

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Thursday 17 Oct 2012

Questions?

- Have you looked at the detailed syllabus?
- And the course plans? (CTH/GU)

Extended rendezvous

- Client(input) =
  request(input)! reply(result)? Client(result)
- Server = request(input)? reply(f(input))! Server
- Notes
  - Server computes function f of whatever input it gets
  - Client waits till its gets the result
  - Both deaf to others until the whole interaction is over
  - We are using a public reply channel, so the reply can be stolen by another process. To avoid this, pass the reply channel too to the server along with the input, as in the resource allocationexample

Extended rendezvous (Ben-Ari style)

```
channel  request
Client
   var local
   loop forever
         request <= local;
         reply => local
Server
    var local
    loop forever
         request => local;
         reply <= f(local)
```

- Notes
  - Server computes function f of whatever input it gets
  - Client waits till its gets the result
  - Both deaf to others until the whole interaction is over

Message passing

- Many ideas missing from Ben-Ari’s chapter
- Let alone broadcast!
  - (generally neglected topic)
- Growing networks (see Sieve example)
- Use of channels as queues (see resource alloc)
- Timeouts (assume possible communication will take place)

Sieve of Eratosthenes

- – Ints (qin, n) = qin!n. Ints(qin, n+1)
- – Outs (qout) = qout?n. print! n. Outs(qout)
- – Filter (p, qin, qout) = qin?n.
  if (n ndiv p) then qout!n. Filter(p,qin,qout)
- – Sift(qin,qout) = qin?p. qout!p.
  new q. (Filter(p, qin, q) | Sift(q, qout))
- – Sieve(qi, qo) = Ints(qi,2)|Sift(qi,qo)|Outs(qo)

- Note the ”new” makes growing network
- Works with synchronous m.p. Asynch?

Sieve of Eratosthenes (Ben-Ari style)

```
Array 1..100 of channel c
process Ints
   integer n := 2
     loop forever
            c(1) <= n;  n := n+1

process Filter (i)  (for 1 = 1..99)
   integer myprime, n
   c(i) => myprime;  print <= myprime;
   loop forever
         c(i) => n;
         if (myprime does not divide n)
               then c(i+1) <= n
```

```
Process Filter(100)
     integer myprime
     c(100) => myprime;
     print <= myprime
```

A program to print the first 100 primes. So Filter(1) filters out even numbers, Filter(2) multiples of 3, etc.

The first input to Filter(i) is the i'th prime. Later inputs are numbers not multiples of the smaller primes.

This program decides in advance to find 100 primes, and runs all 101 processes all the time.

Resource Allocation

- S(dep, pend) =
  ```
  alloc?r. (if dep=0 then S(dep,r:pend)
  else r!ok. S(dep-1,pend))
  +
  rel?. if pend=[] then S(dep+1,[])
  else hd(pend)!ok. S(dep,tl(pend))
  ```
- Resources allocated by server S in fixed size, 1.
- Pend is a queue of return channels
- Alloc and rel are separate channels (requests)

Resource Allocation (Ben-Ari style)

```
process Server
    integer dep; channel alloc, rel, r;
    queue of channels pend;
  loop forever
        alloc => r;
           if dep=0 then pend = r: pend
           else r <= ok;  dep--;
        or
        rel => dummy;
            if pend=[] then dep++
            else hd(pend) <= ok;
                  pend := tl(pend)
```

- Resources allocated by server S in fixed size, 1.
- Pend is a queue of return channels
- Alloc(ate) and rel(ease) are separate channels (requests)
- The code to be run on each request is indented under the input that triggers it.

Resource Allocation with retry channel

- S(dep) =

  ```
  alloc?r. (if dep=0 then pend!r. S(dep)
  else r!ok. S(dep-1))
  +
  rel?
  (pend?r.  r!ok. S(dep)
  timeout. S(dep+1))
  ```
- Retry channel maintains the queue
- Works only for asynchronous m.p.
- Note timeout – triggered if channel retry empty

Resource Allocation with retry channel (Ben-Ari style)

```
process Server
    integer dep; channel alloc, rel, r;
    queue of channels pend;
  loop forever
        alloc => r;
           if dep=0 then pend <= r
           else r <= ok;  dep-- ;
        or
        rel => dummy;
            pend => r;
                r <=ok;
            or
            timeout =>  dep++
```

- Resources allocated by server S in fixed size, 1.
- Alloc(ate) and rel(ease) are separate channels (requests)
- The code to be run on each request is indented under the input that triggers it.
- Pend is a queue of return channels
- The timeout happens if the pend channel is empty (there is no other way to detect an empty channel)

Mutex proof for Dekker's algorithm

- Invariants
  - [] t1 v t2   (ti means turn = i)
  - (p3..p5 v p8..p10) iff wp   (wp=wantp) similar for q
  - (p1 v p2 v p6 v p7) iff !wp   (! = not) similar for q
  - Prove invariance by induction
  - Imply mutex: p8 ^ q8 iff wp ^ wq but p8 => !wq

Dekker progress proof, 1 (variant of Twente proof)

- To prove: [](p2 -> <>p8)
  - Every path from a p2 will lead to a p8
- First, note that [](p2 -> <>p3) by fairness
- Will show [] (p3 -> <> p8)
  - Case 1: <> [] q1 (q gets stuck in NCS)
    - q1 iff !wq, so []q1 => [] !wq
    - [](p3 ^ []q1) => [](p3 ^ [] !wq) => [] <> p8

      by while loop

Dekker progress proof, 2 (variant of Twente proof)

- To show [] (p3 -> <> p8), continued
  - Case 2: [] <> !q1
    - the other case, q never gets stuck in NCS
    - Proof by contradiction, assume p3 ^ !p8, i.e., []p3..p7
    - Lemma1: [] <> t1
      - Again, by contradiction, assume []t2
      - [](p3..p7 ^ t2) => <> [] p6

```
                  => <> !wp
                  => q9              (by progress of q)
                  => t1              (Contradiction!)
```

So [] p3..p7 => <> t1

Dekker progress proof, 3 (variant of Twente proof)

- To show [] (p3 -> <> p8) continued
  - Case 2: [] <> !q1 continued
  - [] p3..p7 => <> t1 prev page
    - => <> [] t1 (never reach p9)
    - => <> [] (p3 v p4) (p3..p7)
    - => <> [] wp (by invariant)
    - => <> [] q6
    - => <> [] !wq (also by invariant)
    - => <> p8 (contradiction!)
  - Hence ![] p3..p7 and [] (p3 ^ []<>!q1 => <> p8)
  - Putting both cases together (q and NCS),

    [](p3=><>p8)

Exchange

- (q4 v q5) iff Lq
- So (q1 v q2) iff !Lq
- Similarly, (p1 v p2) iff !Lp
- To show [] (p2 => <> p4)
  - Assume not. Then [] (p2 v p3) and so [] !Lp
    - Then also ![]q1 (q stuck in NCS because then !Lq and C)
    - So []<>q4 and by progress []<>q1
    - Either p2 immediately after q5, when p will progress
    - or p3 in parallel with q5. If p3 is very slow, PROBLEM.

    NEED ASSUMPTION that one instruction is at least faster than all of q's NCS.

On progress proofs

- Delicate (many cases, did we miss any?)
- Labour intensive
- Error prone (even Ben-Ari’s book?)
- Need machine check
- Then why study them at all by hand?
  - To know what to assert
    - Build the right system
    - The system will check that the system is built right
  - A “true” assertion will always pass
  - If []c doesn’t hold, p ^ []c -> <> p’ will always hold!

CS by fetch-and-add

- faa(c,v) = atomic {v:=c; c++}
  - number of people waiting= c, we wait till v=0.
  - faa(c,v,x) = atomic{v:=c; c:=c+x}; above, x=1
  - wait = repeat faa(c,v) until v=0
  - signal = c:=0

Temporal algebra

- [] distributes over ^, i.e., []A ^ []B iff [](A ^ B)
  - Both sides say that both A and B hold for t>=0
- Why doesn't [] distribute over v ?
  - []A v []B = either A holds from now on, or B does
  - [](A v B) = either A or B holds from now on
- <> distributes over v, i.e., <>A v <>B iff <>(A v B)
  - Both say there is a time when either A or B holds
- Why doesn't <> distribute over ^ ?
  - <>A ^ <>B = exist t1, t2 s.t. A(t1) and B(t2)
  - <>(A ^ B) = exist t s.t. A^B holds at t

More temporal algebra

- [][]A iff []A
- <><>A iff <>A
  - For some r,s,t>=0, lhs says A(r+s) and rhs says A(t)
- <>[]<>A iff []<>A
  - Rhs = “A will be true infinitely often”
  - Lhs = “at some time, A will be true infinitely often”
- Sketched the ideas here. Formally, use the definitions 4.6 and 4.7 in the book (p72,73)

Insertion sort

- P_i(n) = c_(i-1)? x. if x>n then c_i!x. P_i(n)
                                        else c_i!n. P_i(x)
- E = c_n?x. E
- Begin with P_1 through P_n, each holding T's.
  - T is a pseudo-integer, larger than any input integer
- Assuming (for simplicity) n distinct integers to be sorted
  - Masochists can remove the restrictions
- E, the "end process", simply throws away the last part of the wave (what wave?)
- Invariant – the array of integers held is always sorted
- (A)synchronous? (Do you need E in both cases?)

Insertion sort (Ben-Ari style)

- process P_i
```
    integer n
    loop forever
        c_(i-1) => x;
        if x>n then c_i <= x
              else  c_i <= n
```
- process E
```
    integer x
    loop forever
        c_n => x
```
- Notes: as on previous slide

The readers and writes monitor (notes for reading Sec 7.7)

- Remember IRR
- Invariants
  - R >= 0 and W >= 0
  - (R>0 -> W=0) ^ (W =< 1) ^ (W=1 -> R=0)
- Remember to check *eight* operations
  - Each of the four entries run to completion
  - Each of the four resumed entries
- Starvation-freeness applies to waiting processes, not user code outside the monitor