Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 8, 2018

1 Mathematical Artifacts

$$f(x) = x^2$$

```
    type thread_id
    val spawn: (unit -> unit) -> thread_id

    type ’a chan
    val channel : unit -> ’a chan
    val recv : ’a chan -> ’a
    val send : (’a chan * ’a) -> unit

```

```

    signature SERV = sig
      type serv
      val make : unit -> serv
      val call : serv * int -> int
    end

    structure Serv : SERV = struct
      datatype serv = S of (int * int chan) chan

      fun make () = let
        val reqCh = channel ()
        fun loop state = let
          val (v, replCh) = recv reqCh
        in
          send (replCh, state);
          loop v
        end
      in
        spawn (fn () => loop 0);
        S reqCh
      end

      fun call (server, v) = let
        val S reqCh = server
        val replCh = channel ()
      in
        send (reqCh, (v, replCh));
        recv replCh
      end
    end

```

```

type ’a event
val sync : ’a event -> ’a
val recvEvt : ’a chan -> ’a event
val sendEvt : 'a chan * 'a -> unit event
val choose: 'a event * 'a event -> 'a event

fun send (ch, v) = sync (sendEvt (ch, v))
fun recv v = sync (recvEvt v)

val thenEvt: 'a event * ('a -> 'b event)  -> 'b event
```

```


    val server = Serv.make ()
val _ = spawn (fn () => Serv.call (server, 35))
val _ = spawn (fn () =>
        Serv.call (server, 12);
  Serv.call (server, 13)
)
val _ = spawn (fn () => Serv.call (server, 81))
val _ = spawn (fn () => Serv.call (server, 44))
```

```



    structure Serv :> SERV = struct
  datatype serv = S of (int * int chan) chan

  fun make () = let
    val reqCh = FanIn.channel()
    fun loop state = let
      val (v, replCh) = FanIn.recv reqCh
    in
      OneShot.send (replCh, state);
      loop v
    end
  in
    spawn (fn () => loop 0);
    S reqCh
  end

  fun call (server, v) = let
    val S reqCh = server
    val replCh = OneShot.channel ()
  in
    FanIn.send (reqCh, (v, replCh));
    OneShot.recv replCh
  end
end
```

```
   let
   val w = 4
   val x = ref 1
   val y = ref 2
   val z = (!x + 1) + (!y + 2) + (w - 3)
   val w = 1
   in
   y := 0;
   (!y + 2) - (!x + 1) * (w - 3)
   end

```

```

  let
  val x = 1
  val y = 2
  val z = ref (4 * 73)
  val x = 4
in
  z := 1;
  x * !z
end

```

```

    let
  val f = fn x => x 1
  val g = fn y => y + 2
  val h = fn z => z + 3
in
  (f g) + (f h)
end


```

```

```

```

        datatype 'a list = Nil | Cons 'a "'a list"

inductive sorted :: "('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool" where
  Nil : "sorted P Nil" |
  Single : "sorted P (Cons x Nil)" |
  Cons : "P x y ⟹ sorted P (Cons y ys) ⟹ sorted P (Cons x (Cons y ys))"
```

```
        datatype nat = Z | S nat

inductive lte :: "nat ⇒ nat ⇒ bool" where
```

```
  Eq : "lte n n" |
  Lt : "lte n1 n2 ⟹ lte n1 (S n2)"

theorem \"
  sorted lte (Cons (Z) (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil))))\"
 apply (rule Cons)
  apply (rule Lt)
  apply (rule Eq)
 apply (rule Cons)
  apply (rule Eq)
 apply (rule Cons)
  apply (rule Lt)
  apply (rule Lt)
  apply (rule Eq)
 apply (rule Single)
done
```

```


        definition True :: bool where
  "True ≡ ((λx::bool. x) = (λx. x))"

definition False :: bool where
"False ≡ (∀P. P)"
```

```

signature CHAN = sig
  type 'a chan
  val channel: unit -> 'a chan
  val send: 'a chan * 'a -> unit
  val recv: 'a chan -> 'a
end
```

```

        structure ManyToManyChan : CHAN = struct

  type message_queue = 'a option ref queue

  datatype 'a chan_content =
    Send of (condition * 'a) queue |
    Recv of (condition * 'a option ref) queue |
    Inactive

  datatype 'a chan = Ch of 'a chan_content ref * mutex_lock

  fun channel () = Ch (ref Inactive, mutexLock ())

  fun send (Ch (contentRef, lock)) m =
    acquire lock;
    (case !contentRef of
      Recv q =>
        let
          val (recvCond, mopRef) = dequeue q
        in
          mopRef := Some m;
          if (isEmpty q) then contentRef := Inactive else ();
          release lock;
          signal recvCond;
          ()
        end |
      Send q =>
        let
          val sendCond = condition ()
        in
          enqueue (q, (sendCond, m));
          release lock;
          wait sendCond;
          ()
        end |
      Inactive =>
        let
          val sendCond = condition ()
        in
          contentRef := Send (queue [(sendCond, m)]);
          release lock;
          wait sendCond;
          ()
        end)

  fun recv (Ch (contentRef, lock)) =
    acquire lock;
    (case !contentRef of
      Send q =>
        let
          val (sendCond, m) = dequeue q
        in
          if (isEmpty q) then contentRef := Inactive else ();
          release lock;
          signal sendCond;
          m
        end |
      Recv q =>
        let
          val recvCond = condition ()
          val mopRef = ref None
        in
          enqueue (q, (recvCond, mopRef));
```

```
          release lock;
          wait recvCond;
          valOf (!mopRef) |
        end
      Inactive =>
        let
          val recvCond = condition ()
          val mopRef = ref None
        in
          contentRef := Recv (queue [(recvCond, mopRef)]);
          release lock;
          wait recvCond;
          valOf (!mopRef)
        end)

end
```