

# Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 8, 2018

# 1 Mathematical Artifacts

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

```
1  type thread_id
2  val spawn: (unit -> unit) -> thread_id
3
4  type 'a chan
5  val channel : unit -> 'a chan
6  val recv : 'a chan -> 'a
7  val send : ('a chan * 'a) -> unit
8
```

```
1
2  signature SERV = sig
3    type serv
4    val make : unit -> serv
5    val call : serv * int -> int
6  end
7
8  structure Serv : SERV = struct
9    datatype serv = S of (int * int chan) chan
10
11    fun make () = let
12      val reqCh = channel ()
13      fun loop state = let
14        val (v, replCh) = recv reqCh
15      in
16        send (replCh, state);
17        loop v
18      end
19    in
20      spawn (fn () => loop 0);
21      S reqCh
22    end
23
24    fun call (server, v) = let
25      val S reqCh = server
26      val replCh = channel ()
27    in
28      send (reqCh, (v, replCh));
29      recv replCh
30    end
31  end
32
```

```
1
2  type 'a event
3  val sync : 'a event -> 'a
4  val recvEvt : 'a chan -> 'a event
```

```

5 val sendEvt : 'a chan * 'a -> unit event
6 val choose: 'a event * 'a event -> 'a event
7
8 fun send (ch, v) = sync (sendEvt (ch, v))
9 fun recv v = sync (recvEvt v)
10
11 val thenEvt: 'a event * ('a -> 'b event) -> 'b event
12
13

```

```

1
2
3     val server = Serv.make ()
4 val _ = spawn (fn () => Serv.call (server, 35))
5 val _ = spawn (fn () =>
6     Serv.call (server, 12);
7     Serv.call (server, 13)
8 )
9 val _ = spawn (fn () => Serv.call (server, 81))
10 val _ = spawn (fn () => Serv.call (server, 44))
11

```

```

1
2
3
4     structure Serv :> SERV = struct
5 datatype serv = S of (int * int chan) chan
6
7 fun make () = let
8     val reqCh = FanIn.channel()
9     fun loop state = let
10         val (v, replCh) = FanIn.recv reqCh
11     in
12         OneShot.send (replCh, state);
13         loop v
14     end
15 in
16     spawn (fn () => loop 0);
17     S reqCh
18 end
19
20 fun call (server, v) = let
21     val S reqCh = server
22     val replCh = OneShot.channel ()
23 in
24     FanIn.send (reqCh, (v, replCh));
25     OneShot.recv replCh
26 end
27 end
28

```

```

1  let
2    val w = 4
3    val x = ref 1
4    val y = ref 2
5    val z = (!x + 1) + (!y + 2) + (w - 3)
6    val w = 1
7  in
8    y := 0;
9    (!y + 2) - (!x + 1) * (w - 3)
10 end
11

```

```

1
2  let
3    val x = 1
4    val y = 2
5    val z = ref (4 * 73)
6    val x = 4
7  in
8    z := 1;
9    x * !z
10 end
11

```

```

1
2  let
3    val f = fn x => x 1
4    val g = fn y => y + 2
5    val h = fn z => z + 3
6  in
7    (f g) + (f h)
8  end
9
10

```

```

1

```

```

1
2      datatype 'a list = Nil | Cons 'a "'a list"
3
4  inductive sorted :: "('a => 'a => bool) => 'a list => bool"
5    where
6    Nil : "sorted P Nil" |
7    Single : "sorted P (Cons x Nil)" |
8    Cons : "P x y ==> sorted P (Cons y ys) ==> sorted P (Cons x
9      (Cons y ys))"

```

```

1      datatype nat = Z | S nat
2
3  inductive lte :: "nat => nat => bool" where

```

```

4   Eq : "lte n n" |
5   Lt : "lte n1 n2  $\implies$  lte n1 (S n2)"
6
7 theorem \
8   sorted lte (Cons (Z) (Cons (S Z) (Cons (S Z) (Cons (S (S (
9     S Z))) Nil))))\
10  apply (rule Cons)
11  apply (rule Lt)
12  apply (rule Eq)
13  apply (rule Cons)
14  apply (rule Eq)
15  apply (rule Cons)
16  apply (rule Lt)
17  apply (rule Lt)
18  apply (rule Eq)
19 done

```

```

1
2
3   definition True :: bool where
4     "True  $\equiv$  (( $\lambda x :: \text{bool}.$  x) = ( $\lambda x.$  x))"
5
6 definition False :: bool where
7   "False  $\equiv$  ( $\forall P.$  P)"

```

```

1
2 signature CHAN = sig
3   type 'a chan
4   val channel: unit -> 'a chan
5   val send: 'a chan * 'a -> unit
6   val recv: 'a chan -> 'a
7 end

```

```

1
2   structure ManyToManyChan : CHAN = struct
3
4   type message_queue = 'a option ref queue
5
6   datatype 'a chan_content =
7     Send of (condition * 'a) queue |
8     Recv of (condition * 'a option ref) queue |
9     Inactive
10
11  datatype 'a chan = Ch of 'a chan_content ref * mutex_lock
12
13  fun channel () = Ch (ref Inactive, mutexLock ())
14
15  fun send (Ch (contentRef, lock)) m =
16    acquire lock;

```

```

17 (case !contentRef of
18   Recv q =>
19     let
20       val (recvCond, mopRef) = dequeue q
21     in
22       mopRef := Some m;
23       if (isEmpty q) then contentRef := Inactive else ()
24   ;
25     release lock;
26     signal recvCond;
27     ()
28   end |
29   Send q =>
30     let
31       val sendCond = condition ()
32     in
33       enqueue (q, (sendCond, m));
34       release lock;
35       wait sendCond;
36       ()
37     end |
38   Inactive =>
39     let
40       val sendCond = condition ()
41     in
42       contentRef := Send (queue [(sendCond, m)]);
43       release lock;
44       wait sendCond;
45       ()
46     end)
47 fun recv (Ch (contentRef, lock)) =
48   acquire lock;
49   (case !contentRef of
50     Send q =>
51       let
52         val (sendCond, m) = dequeue q
53       in
54         if (isEmpty q) then contentRef := Inactive else ()
55       ;
56         release lock;
57         signal sendCond;
58         m
59       end |
60     Recv q =>
61       let
62         val recvCond = condition ()
63         val mopRef = ref None
64       in
65         enqueue (q, (recvCond, mopRef));

```

```
65         release lock;
66         wait recvCond;
67         valOf (!mopRef) |
68     end
69 Inactive =>
70     let
71         val recvCond = condition ()
72         val mopRef = ref None
73     in
74         contentRef := Recv (queue [(recvCond, mopRef)]);
75         release lock;
76         wait recvCond;
77         valOf (!mopRef)
78     end)
79
80 end
```