

Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 14, 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
9
10
11
12 signature SERV = sig
13   type serv
14   val make : unit -> serv
15   val call : serv * int -> int
16 end
17
18 structure Serv : SERV = struct
19   datatype serv = S of (int * int chan) chan
20
21   fun make () = let
22     val reqCh = channel ()
23     fun loop state = let
24       val (v, replCh) = recv reqCh in
25       send (replCh, state);
26       loop v end in
27     spawn (fn () => loop 0);
28     S reqCh end
29
30   fun call (server, v) = let
31     val S reqCh = server
32     val replCh = channel () in
33     send (reqCh, (v, replCh));
34     recv replCh end end
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

11   val thenEvt: 'a event * ('a -> 'b event) -> 'b event
12
13
1   val server = Serv.make ()
2   val _ = spawn (fn () => Serv.call (server, 35))
3   val _ = spawn (fn () =>
4       Serv.call (server, 12);
5       Serv.call (server, 13))
6   val _ = spawn (fn () => Serv.call (server, 81))
7   val _ = spawn (fn () => Serv.call (server, 44))
8
1   structure Serv : SERV = struct
2     datatype serv = S of (int * int chan) chan
3
4     fun make () = let
5
6       val reqCh = FanIn.channel()
7
8       fun loop state = let
9         val (v, replCh) = FanIn.recv reqCh in
10        OneShot.send (replCh, state);
11        loop v end in
12
13      spawn (fn () => loop 0);
14      S reqCh end
15
16    fun call (server, v) = let
17      val S reqCh = server
18      val replCh = OneShot.channel () in
19      FanIn.send (reqCh, (v, replCh));
20      OneShot.recv replCh end
21
22    end
23
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 in
7     y := 0;
8     (!y + 2) - (!x + 1) * (w - 3) end
9
1   let
2     val x = 1
3     val y = 2

```

```

4      val z = ref (4 * 73)
5      val x = 4 in
6      z := 1;
7      x * !z end
8
1
2      let
3          val f = fn x => x 1
4          val g = fn y => y + 2
5          val h = fn z => z + 3 in
6          (f g) + (f h) end
7
8
1
2      datatype 'a list = Nil | Cons 'a ('a list)
3
4      inductive sorted ::
5          ('a => 'a => bool) =>
6          'a list => bool where
7          Nil : sorted P Nil |
8          Single : sorted P (Cons x Nil) |
9          Cons :
10             P x y =>
11             sorted P (Cons y ys) =>
12             sorted P (Cons x (Cons y ys))
13
1
2      datatype nat = Z | S nat
3
4      inductive lte :: nat => nat => bool where
5          Eq : lte n n |
6          Lt : lte n1 n2 => lte n1 (S n2)
7
8      theorem "
9          sorted lte
10             (Cons (Z) (Cons (S Z)
11                 (Cons (S Z) (Cons
12                     (S (S (S Z))) Nil))))"
13      apply (rule Cons)
14      apply (rule Lt)
15      apply (rule Eq)
16      apply (rule Cons)
17      apply (rule Eq)
18      apply (rule Cons)
19      apply (rule Lt)
20      apply (rule Lt)
21      apply (rule Eq)
22      apply (rule Single)

```

```

22     done
23

```

```

1
2  definition True :: bool where
3      True  $\equiv ((\lambda x :: \text{bool}. x) = (\lambda x. x))$ 
4
5  definition False :: bool where
6      False  $\equiv (\forall P. P)$ 
7
8

```

```

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
8

```

```

1
2  structure ManyToManyChan : CHAN = struct
3      type message_queue = 'a option ref queue
4
5      datatype 'a chan_content =
6          Send of (condition * 'a) queue |
7          Recv of (condition * 'a option ref) queue |
8          Inac
9
10     datatype 'a chan =
11         Ch of 'a chan_content ref * mutex_lock
12
13     fun channel () = Ch (ref Inac, mutexLock ())
14
15     fun send (Ch (conRef, lock)) m =
16         acquire lock;
17         (case !conRef of
18             Recv q => let
19                 val (recvCond, mopRef) = dequeue q in
20                 mopRef := Some m;
21                 if (isEmpty q) then conRef := Inac else ();
22                 release lock; signal recvCond; () end |
23             Send q => let
24                 val sendCond = condition () in
25                 enqueue (q, (sendCond, m));
26                 release lock; wait sendCond; () end |
27             Inac => let
28                 val sendCond = condition () in
29                 conRef := Send (queue [(sendCond, m)]);

```

```

30         release lock; wait sendCond; () end)
31
32 fun recv (Ch (conRef, lock)) =
33     acquire lock;
34     (case !conRef of
35         Send q => let
36             val (sendCond, m) = dequeue q in
37             if (isEmpty q) then
38                 conRef := Inac
39             else
40                 ();
41             release lock; signal sendCond; m end |
42         Recv q => let
43             val recvCond = condition ()
44             val mopRef = ref None in
45             enqueue (q, (recvCond, mopRef));
46             release lock; wait recvCond;
47             valOf (!mopRef) end |
48         Inac => let
49             val recvCond = condition ()
50             val mopRef = ref None in
51             conRef := Recv (queue [(recvCond, mopRef)]);
52             release lock; wait recvCond;
53             valOf (!mopRef) end)
54
55 end
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

22         acquire lock;
23         (let
24             val (recvCond, mopRef) = dequeue q in
25             mopRef := Some m;
26             if (isEmpty q) then conRef := Inac else ();
27             release lock; signal (recvCond);
28             () end) |
29         Send _ => raise NeverHappens end
30
31     fun recv (Ch (conRef, lock)) =
32         acquire lock;
33         (case !conRef of
34             Inac => let
35                 val recvCond = condition ()
36                 val mopRef = ref None in
37                 conRef := Recv (queue [(recvCond, mopRef)]);
38                 release lock; wait recvCond;
39                 valOf (!mopRef) end |
40             Recv q => let
41                 val recvCond = condition ()
42                 val mopRef = ref None in
43                 enqueue (q, (recvCond, mopRef));
44                 release lock; wait recvCond;
45                 valOf (!mopRef) end |
46             Send (sendCond, m) =>
47                 conRef := Inac;
48                 release lock;
49                 signal sendCond;
50                 m end)
51
52     end
53
54
55 1     structure FanInChan : CHAN = struct
56 2
57 3     datatype 'a chan_content =
58 4         Send of (condition * 'a) queue |
59 5         Recv of condition * 'a option ref |
60 6         Inac
61 7
62 8     datatype 'a chan =
63 9         Ch of 'a chan_content ref * mutex_lock
64 10
65 11     fun channel () = Ch (ref Inac, mutexLock ())
66 12
67 13     fun send (Ch (conRef, lock)) m =
68 14         acquire lock;
69 15         case !conRef of
70 16             Recv (recvCond, mopRef) =>
71 17                 mopRef := Some m; conRef := Inac;

```

```

18         release lock; signal recvCond;
19         () |
20     Send q => let
21         val sendCond = condition () in
22         enqueue (q, (sendCond, m));
23         release lock; wait sendCond;
24         () end |
25     Inac => let
26         val sendCond = condition () in
27         conRef := Send (queue [(sendCond, m)])
28         release lock; wait sendCond; () end
29
30 fun recv (Ch (conRef, lock)) = let
31     val recvCond = condition ()
32     val mopRef = ref None in
33     case cas (conRef, Inac, Recv (recvCond, mopRef)) of
34         Inac => (* conRef already set *)
35             wait recvCond; valOf (!mopRef) |
36         Send q =>
37             (* the current thread is the only one
38              * that updates the state from this state *)
39             acquire lock;
40             (let
41                 val (sendCond, m) = dequeue q in
42                 if (isEmpty q) then conRef := Inac else ();
43                 release lock; signal sendCond; m end) |
44         Recv _ => raise NeverHappens end end
45
46

```

```

1
2 structure OneToOneChan : CHAN = struct
3
4     datatype 'a chan_content =
5         Send of condition * 'a |
6         Recv of condition * 'a option ref |
7         Inac
8
9     datatype 'a chan = Ch of 'a chan_content ref
10
11 fun channel () = Ch (ref Inac)
12
13 fun send (Ch conRef) m = let
14     val sendCond = condition () in
15     case cas (conRef, Inac, Send (sendCond, m)) of
16         Inac =>
17             (* conRef already set to Send *)
18             wait sendCond; () |
19         Recv (recvCond, mopRef) =>
20             (* the current thread is the only one

```



```

21      /* that accesses conRef for this state */
22      mopRef := Some m; conRef := Inac;
23      signal recvCond; () |
24      Send _ => raise NeverHappens end end
25
26
27 fun recv (Ch conRef) = let
28   val recvCond = condition ();
29   val mopRef = ref None in
30   case cas (conRef, Inac, Recv (recvCond, mopRef)) of
31     Inac => (* conRef already set to Recv*)
32     wait recvCond; valOf (!mopRef) |
33     Send (sendCond, m) =>
34     (* the current thread is the only one
35      /* that accesses conRef for this state */
36      conRef := Inac; signal sendCond; m |
37     Recv _ => raise NeverHappens end end
38
39 end
40
41
42 1 structure OneShotChan : CHAN = struct
43 2
44 3 datatype 'a chan_content =
45 4   Send of condition * 'a |
46 5   Recv of condition * 'a option ref |
47 6   Inac
48 7
49 8 datatype 'a chan = Ch of 'a chan_content ref *
50 mutex_lock
51 9
52 10 fun channel () = Ch (ref Inac, lock ())
53 11
54 12 fun send (Ch (conRef, lock)) m = let
55 13   val sendCond = condition () in
56 14   case (conRef, Inac, Send (sendCond, m)) of
57 15     Inac =>
58 16     (* conRef already set to Send*)
59 17     wait sendCond; () |
60 18     Recv (recvCond, mopRef) =>
61 19     mopRef := Some m; signal recvCond;
62 20     () |
63 21     Send _ => raise NeverHappens end end
64 22
65 23
66 24 fun recv (Ch (conRef, lock)) = let
67 25   val recvCond = condition ()
68 26   val mopRef = ref None in
69 27   case (conRef, Inac, Recv (recvCond, mopRef)) of
70 28     Inac =>

```

```

29      (* conRef already set to Recv*)
30      wait recvCond; valOf (!mopRef) |
31      Send (sendCond, m) =>
32          acquire lock; signal sendCond;
33          (* never releases lock;
34          -* blocks others forever *)
35          m |
36      Recv _ =>
37          acquire lock;
38          (* never able to acquire lock;
39          -* blocked forever *)
40          raise NeverHappens end end
41
42 end
43
1  structure OneShotToOneChan : CHAN = struct
2
3      datatype 'a chan =
4          Ch of condition * condition * 'a option ref
5
6      fun channel () =
7          Ch (condition (), condition (), ref None)
8
9      fun send (Ch (sendCond, recvCond, mopRef)) m =
10          mopRef := Some m; signal recvCond;
11          wait sendCond; ()
12
13      fun recv (Ch (sendCond, recvCond, mopRef)) =
14          wait recvCond; signal sendCond;
15          valOf (!mopRef)
16
17      end
18
1
2      datatype var = Var string
3
4      datatype exp =
5          Let var boundexp exp |
6          Rslt var
7
8      boundexp =
9          Unit |
10         Chan |
11         Prim prim |
12         Spawn exp |
13         Sync var |
14         Fst var |
15         Snd var |

```

```

16     Case var var exp var exp |
17     App var var and
18
19   prim =
20     SendEvt var var |
21     RecvEvt var |
22     Pair var var |
23     Left var |
24     Right var |
25     Abs var var ex
26
27
1   datatype ctrl_label =
2     LNxt var | LSpwn var | LCall var | LRtn var
3
4   type_synonym ctrl_path = (ctrl_label list)
5
6   datatype chan = Ch ctrl_path var
7
8   datatype val =
9     VUnit | VChan chan | VClsr prim (var  $\rightarrow$  val)
10
11   datatype ctn = Ctn var exp (var  $\rightarrow$  val)
12
13   datatype state = Stt exp (var  $\rightarrow$  val) (ctn list)
14
15
1
2   inductive seq_step ::
3     bind * (var  $\rightarrow$  val))  $\Rightarrow$  val  $\Rightarrow$  bool where
4     Let_Unit:
5       seq_step (Unit, env) VUnit |
6     Let_Prim:
7       seq_step (Prim p, env) (VClsr p env) |
8     Let_Fst:
9       env xp = Some (VClsr (Pair x1 x2) envp)  $\Rightarrow$ 
10       envp x1 = Some v  $\Rightarrow$ 
11       seq_step (FST xp, env) v |
12     Let_Snd:
13       env xp = Some (VClsr (Pair x1 x2) envp)  $\Rightarrow$ 
14       envp x2 = Some v  $\Rightarrow$ 
15       seq_step (SND xp, env) v
16
17   inductive seq_step_up ::
18     bind * (var  $\rightarrow$  val))  $\Rightarrow$  exp * val_env  $\Rightarrow$  bool where
19     Let_Case_Left:
20       env xs = Some (VClsr (Left x1') envl)  $\Rightarrow$ 
21       envl x1' = Some vl  $\Rightarrow$ 

```

```

22     seq_step_up
23       (Case xs xl el xr er, env)
24       (el, env(xl ↦ vl)) |
25   Let_Case_Right:
26     env xs = Some (VClsr (Right xr') envr) ⇒
27     envr xr' = Some vr ⇒
28     seq_step_up
29       (Case xs xl el xr er, env)
30       (er, env(xr ↦ vr)) |
31   Let_App:
32     env f = Some (VClsr (Abs fp xp el) envl) ⇒
33     env xa = Some va ⇒
34     seq_step_up
35       (App f xa, env)
36       (el, envl(
37         fp ↦ (VClsr (Abs fp xp el) envl),
38         xp ↦ va))
39
40
41   type_synonym cmmn_set = (ctrl_path * chan * ctrl_path)
42   set
43
44   type_synonym trace_pool = ctrl_path → state
45
46   inductive leaf ::
47     trace_pool ⇒ ctrl_path ⇒ bool where
48     Intro:
49       trpl pi ≠ None ⇒
50       (∄ pi' . trpl pi' ≠ None ∧ strict_prefix pi pi') ⇒
51       leaf trpl pi
52
53   inductive concur_step ::
54     trace_pool * cmmn_set ⇒
55     trace_pool * cmmn_set ⇒
56     bool where
57     Seq_Ststep_Down:
58       leaf trpl pi ⇒
59       trpl pi = Some (
60         Stt (Rslt x) env ((Ctn xk ek envk) # k)) ⇒
61       env x = Some v ⇒
62       concur_step
63         (trpl, ys)
64         (trpl(pi;;(LRtn xk) ↦
65           (Stt ek (envk(xk ↦ v)) k)), ys) |
66     Seq_Step:
67       leaf trpl pi ⇒
68       trpl pi = Some (Stt (Let x b e) env k) ⇒
69       seq_step (b, env) v ⇒
70       concur_step
71         (trpl, ys)

```

```

71      (trpl(pi;;(LNext x)  $\mapsto$  (Stt e (env(x  $\mapsto$  v)) k), ys)
72      |
73      Seq_Step_Up:
74      leaf trpl pi  $\Rightarrow$ 
75      trpl pi = Some (Stt (Let x b e) env k)  $\Rightarrow$ 
76      seq_step_up (b, env) (e', env')  $\Rightarrow$ 
77      concur_step
78      (trpl, ys)
79      (trpl(pi;;(LCall x)  $\mapsto$ 
80      (Stt e' env' ((Ctn x e env) # k))), ys) |
81      Let_Chain:
82      leaf trpl pi  $\Rightarrow$ 
83      trpl pi = Some (Stt (Let x Chan e) env k)  $\Rightarrow$ 
84      concur_step
85      (trpl, ys)
86      (trpl(pi;;(LNext x)  $\mapsto$ 
87      (Stt e (env(x  $\mapsto$  (VChan (Ch pi x)))) k))), ys) |
88      Let_Spawn:
89      leaf trpl pi  $\Rightarrow$ 
90      trpl pi = Some (Stt (Let x (Spawn ec) e) env k)  $\Rightarrow$ 
91      concur_step
92      (trpl, ys)
93      (trpl(
94      pi;;(LNext x)  $\mapsto$  (St e (env(x  $\mapsto$  VUnit)) k),
95      pi;;(LSpawn x)  $\mapsto$  (St ec env []), ys) |
96      Let_Sync:
97      leaf trpl pis  $\Rightarrow$ 
98      trpl pis = Some (Stt (Let xs (Sync xse) es) envs ks)
99       $\Rightarrow$ 
100      envs xse = Some (VClsr (Send_Evt xsc xm) envse)  $\Rightarrow$ 
101      leaf trpl pir  $\Rightarrow$ 
102      trpl pir = Some (Stt (Let xr (Sync xre) er) envr kr)
103       $\Rightarrow$ 
104      envr xre = Some (VClsr (Recv_Evt xrc) envre)  $\Rightarrow$ 
105      envse xsc = Some (VChan c)  $\Rightarrow$ 
106      envre xrc = Some (VChan c)  $\Rightarrow$ 
107      envse xm = Some vm  $\Rightarrow$ 
108      concur_step
109      (trpl, ys)
110      (trpl(
111      pis;;(LNext xs)  $\mapsto$  (Stt es (envs(xs  $\mapsto$  VUnit)) ks
112      ),
113      pir;;(LNext xr)  $\mapsto$  (Stt er (envr(xr  $\mapsto$  vm)) kr)),
114      ys  $\cup$  {(pis, c, pir)})

```