

Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 15, 2018

1 Mathematical Artifacts

```
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 signature SERV = sig
12   type serv
13   val make : unit -> serv
14   val call : serv * int -> int
15 end
16
17 structure Serv : SERV = struct
18   datatype serv = S of (int * int chan) chan
19
20   fun make () = let
21     val reqChn = channel ()
22     fun loop state = let
23       val (v, replCh) = recv reqChn in
24       send (replCh, state);
25       loop v end in
26     spawn (fn () => loop 0);
27     S reqChn end
28
29   fun call (server, v) = let
30     val S reqChn = server
31     val replChn = channel () in
32     send (reqCh, (v, replCh));
33     recv replChn end end
34
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
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
```

```

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 reqChn = FanIn.channel()
7
8          fun loop state = let
9              val (v, replCh) = FanIn.recv reqChn in
10                 OneShot.send (replCh, state);
11                 loop v end in
12
13             spawn (fn () => loop 0);
14             S reqChn end
15
16     fun call (server, v) = let
17         val S reqChn = server
18         val replChn = OneShot.channel () in
19             FanIn.send (reqCh, (v, replCh));
20             OneShot.recv replChn 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
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)
23   done

```

```

1  definition True :: bool where
2      True ≡ ((λx::bool. x) = (λx. x))
3
4  definition False :: bool where
5      False ≡ (∀P. P)
6
7
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
9
10 structure ManyToManyChan : CHAN = struct
11     type message_queue = 'a option ref queue
12
13     datatype 'a chan_content =
14         Send of (condition * 'a) queue |
15         Recv of (condition * 'a option ref) queue |
16         Inac
17
18     datatype 'a chan =
19         Chn of 'a chan_content ref * mutex_lock
20
21     fun channel () = Chn (ref Inac, mutexLock ())
22
23     fun send (Chn (conRef, lock)) m =
24         acquire lock;
25         (case !conRef of
26             Recv q => let
27                 val (recvCond, mopRef) = dequeue q in
28                 mopRef := Some m;
29                 if (isEmpty q) then conRef := Inac else ();
30                 release lock; signal recvCond; () end |
31             Send q => let
32                 val sendCond = condition () in
33                 enqueue (q, (sendCond, m));
34                 release lock; wait sendCond; () end |
35             Inac => let
36                 val sendCond = condition () in
37                 conRef := Send (queue [(sendCond, m)]);
38                 release lock; wait sendCond; () end)
39
40     fun recv (Chn (conRef, lock)) =
41         acquire lock;
42         (case !conRef of
43             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
1
2     structure FanOutChan : CHAN = struct
3
4     datatype 'a chan_content =
5         Send of condition * 'a |
6         Recv of (condition * 'a option ref) queue |
7         Inac
8
9     datatype 'a chan =
10         Chn of 'a chan_content ref * mutex_lock
11
12     fun channel () = Chn (ref Inac, mutexLock ())
13
14     fun send (Chn (conRef, lock)) m = let
15         val sendCond = condition () in
16         case cas (conRef, Inac, Send (sendCond, m)) of
17             Inac => (* conRef already set *)
18                 wait sendCond; () |
19             Recv q =>
20                 (* the current thread is
21                  * the only one that updates from this state *)
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 (Chn (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      Chn of 'a chan_content ref * mutex_lock
64    10
65    11  fun channel () = Chn (ref Inac, mutexLock ())
66    12
67    13  fun send (Chn (conRef, lock)) m =
68    14      acquire lock;
69    15      case !conRef of
70    16      Recv (recvCond, mopRef) =>
71    17          mopRef := Some m; conRef := Inac;
72    18          release lock; signal recvCond;
73    19          () |
74    20      Send q => let
75    21          val sendCond = condition () in
76    22          enqueue (q, (sendCond, m));
77    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 (Chn (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 = Chn of 'a chan_content ref
10
11   fun channel () = Chn (ref Inac)
12
13   fun send (Chn 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 (Chn 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
1  structure OneShotChan : CHAN = struct
2
3  datatype 'a chan_content =
4    Send of condition * 'a |
5    Recv of condition * 'a option ref |
6    Inac
7
8  datatype 'a chan = Chn of 'a chan_content ref * mutex_lock
9
10 fun channel () = Chn (ref Inac, lock ())
11
12 fun send (Chn (conRef, lock)) m = let
13   val sendCond = condition () in
14   case (conRef, Inac, Send (sendCond, m)) of
15     Inac =>
16       (* conRef already set to Send*)
17       wait sendCond; () |
18     Recv (recvCond, mopRef) =>
19       mopRef := Some m; signal recvCond;
20     () |
21     Send _ => raise NeverHappens end end
22
23
24 fun recv (Chn (conRef, lock)) = let
25   val recvCond = condition ()
26   val mopRef = ref None in
27   case (conRef, Inac, Recv (recvCond, mopRef)) of
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         Chn of condition * condition * 'a option ref
5
6     fun channel () =
7         Chn (condition (), condition (), ref None)
8
9     fun send (Chn (sendCond, recvCond, mopRef)) m =
10         mopRef := Some m; signal recvCond;
11         wait sendCond; ()
12
13     fun recv (Chn (sendCond, recvCond, mopRef)) =
14         wait recvCond; signal sendCond;
15         valOf (!mopRef)
16
17 end
18

```

2 Syntax

```

1
2     datatype var = Var string
3
4     datatype exp =
5         Let var boundexp exp |
6         Rslt var
7
8     boundexp =
9         Unt |
10        MkChn |
11        Prim prim |
12        Spwn 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   Lft var |
24   Rht var |
25   Abs var var ex

```

3 Dynamic Semantics

```

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 = Chn ctrl_path var
7
8  datatype val =
9    VUnt | VChn 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    LetUnt:
5      seq_step (Unt, env) VUnt |
6    LetPrim:
7      seq_step (Prim p, env) (VClsr p env) |
8    LetFst:
9      env xp = Some (VClsr (Pair x1 x2) envp)  $\Rightarrow$ 
10      envp x1 = Some v  $\Rightarrow$ 
11      seq_step (Fst xp, env) v |
12    LetSnd:
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
18

```

```

1
2
3  inductive seq_step_up ::
4    bind * (var  $\rightarrow$  val))  $\Rightarrow$  exp * val_env  $\Rightarrow$  bool where
5    LetCaseLft:

```

```

6      env xs = Some (VClsr (Lft xl') envl) ==>
7      envl xl' = Some vl ==>
8      seq_step_up
9      (Case xs xl el xr er, env)
10     (el, env(xl ↦ vl)) |
11  LetCaseRht:
12     env xs = Some (VClsr (Rht xr') envr) ==>
13     envr xr' = Some vr ==>
14     seq_step_up
15     (Case xs xl el xr er, env)
16     (er, env(xr ↦ vr)) |
17  LetApp:
18     env f = Some (VClsr (Abs fp xp el) envl) ==>
19     env xa = Some va ==>
20     seq_step_up
21     (App f xa, env)
22     (el, envl(
23       fp ↦ (VClsr (Abs fp xp el) envl),
24       xp ↦ va))
25
1
2
3  type_synonym cmmn_set = (ctrl_path * chan * ctrl_path) set
4
5  type_synonym trace_pool = ctrl_path → state
6
7  inductive leaf ::
8    trace_pool ⇒ ctrl_path ⇒ bool where
9    intro:
10      trpl pi ≠ None ==>
11      (∄ pi' . trpl pi' ≠ None ∧ strict_prefix pi pi') ==>
12      leaf trpl pi
13
14
1
2  inductive concur_step ::
3    trace_pool * cmmn_set ⇒
4    trace_pool * cmmn_set ⇒
5    bool where
6    Seq_Ststep_Down:
7      leaf trpl pi ==>
8      trpl pi = Some
9      (Stt (Rslt x) env
10       ((Ctn xk ek envk) # k)) ==>
11      env x = Some v ==>
12      concur_step
13      (trpl, ys)

```

```

14      (trpl(pi @ [LRtn xk]  $\mapsto$ 
15        (Stt ek (envk(xk  $\mapsto$  v)) k)), ys) |
16 Seq_Step:
17   leaf trpl pi  $\Rightarrow$ 
18   trpl pi = Some
19     (Stt (Let x b e) env k)  $\Rightarrow$ 
20   seq_step (b, env) v  $\Rightarrow$ 
21   concur_step
22     (trpl, ys)
23     (trpl(pi @ [LNxt x]  $\mapsto$ 
24       (Stt e (env(x  $\mapsto$  v)) k)), ys) |
25 Seq_Step_Up:
26   leaf trpl pi  $\Rightarrow$ 
27   trpl pi = Some
28     (Stt (Let x b e) env k)  $\Rightarrow$ 
29   seq_step_up (b, env) (e', env')  $\Rightarrow$ 
30   concur_step
31     (trpl, ys)
32     (trpl(pi @ [LCall x]  $\mapsto$ 
33       (Stt e' env'
34         ((Ctn x e env) # k))), ys) |
35 LetMkCh:
36   leaf trpl pi  $\Rightarrow$ 
37   trpl pi = Some (Stt (Let x MkChn e) env k)  $\Rightarrow$ 
38   concur_step
39     (trpl, ys)
40     (trpl(pi @ [LNxt x]  $\mapsto$ 
41       (Stt e (env(x  $\mapsto$  (VChn (Chn pi x)))) k)), ys) |
42 LetSpwn:
43   leaf trpl pi  $\Rightarrow$ 
44   trpl pi = Some
45     (Stt (Let x (Spwn ec) e) env k)  $\Rightarrow$ 
46   concur_step
47     (trpl, ys)
48     (trpl(
49       pi @ [LNxt x]  $\mapsto$ 
50       (St e (env(x  $\mapsto$  VUnt)) k),
51       pi @ [LSpwn x]  $\mapsto$ 
52       (St ec env []), ys) |
53 LetSync:
54   leaf trpl pis  $\Rightarrow$ 
55   trpl pis = Some
56     (Stt (Let xs (Sync xse) es) envs ks)  $\Rightarrow$ 
57   envs xse = Some
58     (VClsr (SendEvt xsc xm) envse)  $\Rightarrow$ 
59   leaf trpl pir  $\Rightarrow$ 
60   trpl pir = Some
61     (Stt (Let xr (Sync xre) er) envr kr)  $\Rightarrow$ 
62   envr xre = Some
63     (VClsr (RecvEvt xrc) envre)  $\Rightarrow$ 

```

```

64     envse xsc = Some (VChn c) ==>
65     envre xrc = Some (VChn c) ==>
66     envse xm = Some vm ==>
67     concur_step
68       (trpl, ys)
69       (trpl(
70         pis @ [LNxt xs] ↦
71         (Stt es (envs(xs ↦ VUnt)) ks),
72         pir @ [LNxt xr] ↦
73         (Stt er (envr(xr ↦ vm)) kr)),
74       ys ∪ {(pis, c, pir)})
75
76
1   inductive star :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool
2   for r where
3     refl: star r x x |
4     step: r x y ==> star r y z ==> star r x z
5

```

4 Dynamic Communication

```

1   inductive is_send_path ::
2     trace_pool ⇒ chan ⇒
3     control_path ⇒ bool where
4     intro:
5       trpl piy = Some
6         (Stt (Let xy (Sync xe) en) env k) ==>
7       env xe = Some
8         (VClSr (SendEvt xsc xm) enve) ==>
9       enve xsc = Some (VChn c) ==>
10      is_send_path trpl c piy
11
12  inductive is_recv_path ::
13    trace_pool ⇒ chan ⇒
14    control_path ⇒ bool where
15    intro:
16      trpl piy = Some
17        (Stt (Let xy (Sync xe) en) env k) ==>
18      env xe = Some
19        (VClSr (RecvEvt xrc) enve) ==>
20      enve xrc = Some (VChn c) ==>
21      is_recv_path trpl c piy
22
23
1
2   inductive every_two ::
3     ('a ⇒ bool) ⇒

```

```

4      ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$ 
5      bool where
6      intro: ( $\forall$  pi1 pi2 .
7          p x1  $\rightarrow$ 
8          p x2  $\rightarrow$ 
9          r x1 x2)  $\Rightarrow$ 
10         every_two p r
11
12     inductive ordered :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool where
13         left: prefix pi1 pi2  $\Rightarrow$  ordered pi1 pi2 |
14         right: prefix pi2 pi1  $\Rightarrow$  ordered pi1 pi2
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
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

```

```

1
2     inductive one_shot :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
3         intro:
4             every_two
5                 (is_send_path trpl c) op=  $\Rightarrow$ 
6                 one_shot trpl c
7
8     inductive fan_out :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
9         intro:
10             every_two
11                 (is_send_path trpl c) ordered  $\Rightarrow$ 
12                 fan_out trpl c
13
14     inductive fan_in :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
15         intro:
16             every_two
17                 (is_recv_path trpl c) ordered  $\Rightarrow$ 
18                 fan_in trpl c
19
20     inductive one_to_one :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
21         intro:
22             fan_out trpl c  $\Rightarrow$ 
23             fan_in trpl c  $\Rightarrow$ 
24             one_to_one trpl c
25
26

```

5 Static Semantics

```

1
2     datatype abstract_value =
3         AChn var |
4         AUnt |
5         APrim prim
6

```

```

7  type_synonym abstract_env = var  $\Rightarrow$  abstract_value set
8
9  fun rslt_var :: exp  $\Rightarrow$  var where
10   rslt_var (Rslt x) = x |
11   rslt_var (Let _ _ e) = (rslt_var e)
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41

```

```

1
2
3  inductive may_be_eval_exp ::
4    abstract_env * abstract_env  $\Rightarrow$ 
5    exp  $\Rightarrow$  bool where
6    Result:
7      may_be_eval_exp (V, C) (RESULT x) |
8    Let_Unt:
9      {AUnt}  $\subseteq$  V x  $\Rightarrow$ 
10     may_be_eval_exp (V, C) e  $\Rightarrow$ 
11     may_be_eval_exp (V, C) (Let x Unt e) |
12    Let_Chn:
13      {AChn x}  $\subseteq$  V x  $\Rightarrow$ 
14     may_be_eval_exp (V, C) e  $\Rightarrow$ 
15     may_be_eval_exp (V, C) (Let x (MkChn) e) |
16    Let_SendEvt :
17      {APrim (SendEvt xc xm)}  $\subseteq$  V x  $\Rightarrow$ 
18     may_be_eval_exp (V, C) e  $\Rightarrow$ 
19     may_be_eval_exp (V, C) (Let x (Prim (SendEvt xc xm)) e
20    ) |
21    Let_RecvEvt :
22      {APrim (RecvEvt xc)}  $\subseteq$  V x  $\Rightarrow$ 
23     may_be_eval_exp (V, C) e  $\Rightarrow$ 
24     may_be_eval_exp (V, C) (Let x (Prim (RecvEvt xc)) e) |
25    Let_Pair :
26      {APrim (Pair x1 x2)}  $\subseteq$  V x  $\Rightarrow$ 
27     may_be_eval_exp (V, C) e  $\Rightarrow$ 
28     may_be_eval_exp (V, C) (Let x (Pair x1 x2) e) |
29    Let_Left :
30      {APrim (Left xp)}  $\subseteq$  V x  $\Rightarrow$ 
31     may_be_eval_exp (V, C) e  $\Rightarrow$ 
32     may_be_eval_exp (V, C) (Let x (Left xp) e) |
33    Let_Right:
34      {APrim (Right xp)}  $\subseteq$  V x  $\Rightarrow$ 
35     may_be_eval_exp (V, C) e  $\Rightarrow$ 
36     may_be_eval_exp (V, C) (Let x (Right xp) e) |
37    Let_Abs :
38      {APrim (Abs f' x' e')}  $\subseteq$  V f'  $\Rightarrow$ 
39     may_be_eval_exp (V, C) e'  $\Rightarrow$ 
40     {APrim (Abs f' x' e')}  $\subseteq$  V x  $\Rightarrow$ 
41     may_be_eval_exp (V, C) e  $\Rightarrow$ 
42     may_be_eval_exp (V, C) (Let x (Abs f' x' e') e) |

```



```

42   Let_Spawn:
43     {AUnt}  $\subseteq V$  x  $\implies$ 
44     may_be_eval_exp (V, C) ec  $\implies$ 
45     may_be_eval_exp (V, C) e  $\implies$ 
46     may_be_eval_exp (V, C) (Let x (Spwn ec) e) |
47   Let_Sync :
48      $\forall$  xsc xm xc .
49     (APrim (SendEvt xsc xm))  $\in V$  xe  $\longrightarrow$ 
50     AChn xc  $\in V$  xsc  $\longrightarrow$ 
51     {AUnt}  $\subseteq V$  x  $\wedge V$  xm  $\subseteq C$  xc  $\implies$ 
52      $\forall$  xrc xc .
53     (APrim (RecvEvt xrc))  $\in V$  xe  $\longrightarrow$ 
54     AChn xc  $\in V$  xrc  $\longrightarrow$ 
55     C xc  $\subseteq V$  x  $\implies$ 
56     may_be_eval_exp (V, C) e  $\implies$ 
57     may_be_eval_exp (V, C) (Let x (Syync xe) e) |
58   Let_Fst:
59      $\forall$  x1 x2.
60     (APrim (Pair x1 x2))  $\in V$  xp  $\longrightarrow$ 
61     V x1  $\subseteq V$  x  $\implies$ 
62     may_be_eval_exp (V, C) e  $\implies$ 
63     may_be_eval_exp (V, C) (Let x (Fst xp) e) |
64   Let_Snd:
65      $\forall$  x1 x2 .
66     (APrim (Pair x1 x2))  $\in V$  xp  $\longrightarrow$ 
67     V x2  $\subseteq V$  x  $\implies$ 
68     may_be_eval_exp (V, C) e  $\implies$ 
69     may_be_eval_exp (V, C) (Let x (Snd xp) e) |
70   Let_Case:
71      $\forall$  x1' .
72     (APrim (Left x1'))  $\in V$  xs  $\longrightarrow$ 
73     V x1'  $\subseteq V$  x1  $\wedge V$  (rslt_var e1)  $\subseteq V$  x  $\wedge$ 
74     may_be_eval_exp (V, C) e1  $\implies$ 
75      $\forall$  xr' .
76     (APrim (Right xr'))  $\in V$  xs  $\longrightarrow$ 
77     V xr'  $\subseteq V$  xr  $\wedge V$  (rslt_var er)  $\subseteq V$  x  $\wedge$ 
78     may_be_eval_exp (V, C) er  $\implies$ 
79     may_be_eval_exp (V, C) e  $\implies$ 
80     may_be_eval_exp (V, C) (Let x (Case xs x1 e1 xr er) e)
81   |
82   Let_App:
83      $\forall$  f' x' e' .
84     (APrim (Abs f' x' e'))  $\in V$  f  $\longrightarrow$ 
85     V xa  $\subseteq V$  x'  $\wedge$ 
86     V (rslt_var e')  $\subseteq V$  x  $\implies$ 
87     may_be_eval_exp (V, C) e  $\implies$ 
88     may_be_eval_exp (V, C) (Let x (App f xa) e)
89

```

```

1
2
3 fun abstract :: val  $\Rightarrow$  abstract_value where
4   abstract VUnt = AUnt |
5   abstract VChn (Chn pi x) = AChn x |
6   abstract VClsr p env = APrim p
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39

```

```

1
2
3
4 inductive
5   may_be_eval_val ::
6     abstract_env * abstract_env  $\Rightarrow$  val  $\Rightarrow$  bool and
7   may_be_eval_env ::
8     abstract_env * abstract_env  $\Rightarrow$  val_env  $\Rightarrow$  bool where
9   Unt:
10     may_be_eval_val (V, C) VUnt |
11   Chan:
12     may_be_eval_val (V, C) VChn c |
13   SendEvt:
14     may_be_eval_env (V, C) env  $\Rightarrow$ 
15     may_be_eval_val (V, C) (VClsr (SendEvt _ _) env) |
16   RecvEvt:
17     may_be_eval_env (V, C) env  $\Rightarrow$ 
18     may_be_eval_val (V, C) (VClsr (RecvEvt _) env) |
19   Left:
20     may_be_eval_env (V, C) env  $\Rightarrow$ 
21     may_be_eval_val (V, C) (VClsr (Left _) env) |
22   Right:
23     may_be_eval_env (V, C) env  $\Rightarrow$ 
24     may_be_eval_val (V, C) (VClsr (Right _) env) |
25   Abs:
26     {(APrim (Abs f x e))}  $\subseteq$  V f  $\Rightarrow$ 
27     may_be_eval_exp (V, C) e  $\Rightarrow$ 
28     may_be_eval_env (V, C) env  $\Rightarrow$ 
29     may_be_eval_val (V, C) (VClsr (Abs f x e) env) |
30   Pair:
31     may_be_eval_env (V, C) env  $\Rightarrow$ 
32     may_be_eval_val (V, C) (VClsr (Pair _ _) env) |
33   intro:
34      $\forall$  x v .
35     env x = Some v  $\longrightarrow$ 
36     {abstract v}  $\subseteq$  V x  $\wedge$  may_be_eval_val (V, C) v  $\Rightarrow$ 
37     may_be_eval_env (V, C) env
38
39

```

1

```

2   inductive may_be_eval_stack ::
3     abstract_env * abstract_env ⇒
4     abstract_value set ⇒ cont list ⇒ bool where
5     Empty:
6       may_be_eval_stack (V, C) valset [] |
7     Nonempty:
8       valset ⊆ V x ⇒
9       may_be_eval_exp (V, C) e ⇒
10      may_be_eval_env (V, C) env ⇒
11      may_be_eval_stack (V, C) (V (rslt_var e)) k ⇒
12      may_be_eval_stack (V, C) valset ((Ctn x e env) # k)
13
14
15  inductive may_be_eval_state ::
16    abstract_env * abstract_env ⇒
17    state ⇒ bool where
18    intro:
19      may_be_eval_exp (V, C) e ⇒
20      may_be_eval_env (V, C) env ⇒
21      may_be_eval_stack (V, C) (V (rslt_var e)) k ⇒
22      may_be_eval_state (V, C) (Stt e env k)
23
24  inductive may_be_eval_pool ::
25    abstract_env * abstract_env ⇒
26    trace_pool ⇒ bool where
27    intro:
28      ∀ pi st .
29      trpl pi = Some st →
30      may_be_eval_state (V, C) st ⇒
31      may_be_eval_pool (V, C) trpl
32
33
34
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

```

```

1
2   theorem may_be_eval_preserved_under_concur_step : "
3     may_be_eval_pool (V, C) trpl ⇒
4     concur_step (trpl, ys) (trpl', ys') ⇒
5     may_be_stati_eval_pool (V, C) trpl'"
6 proof outline
7 qed
8
9   theorem may_be_eval_preserved_under_concur_step_star : "
10    may_be_eval_pool (V, C) trpl ⇒
11    star concur_step (trpl, ys) (trpl', ys') ⇒
12    may_be_concur_step (V, C) trpl'"
13 proof outline
14 qed
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
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

```

```

2  theorem trace_pool_snapshot_not_value_sound : "
3    env x = Some v  $\implies$ 
4    trpl pi = Some (Stt e env k)  $\implies$ 
5    may_be_eval_pool (V, C) trpl  $\implies$ 
6    {abstract v}  $\subseteq$  V x "
7  proof outline
8  qed
9
1
10
11 theorem trace_pool_always_not_value_sound : "
12   env' x = Some v  $\implies$ 
13   may_be_eval_pool (V, C) trpl  $\implies$ 
14   star concur_step (trpl, ys) (trpl', ys')  $\implies$ 
15   trpl' pi = Some (Stt e' env' k')  $\implies$ 
16   {abstract v}  $\subseteq$  V x"
17 proof outline
18 qed
19
20 theorem exp_always_not_value_sound : "
21   env' x = Some v  $\implies$ 
22   may_be_eval_exp (V, C) e  $\implies$ 
23   star concur_step
24     ([[]  $\mapsto$  (Stt e ( $\lambda$  _ . None) [])], ys)
25     (trpl', ys')  $\implies$ 
26   trpl' pi = Some (Stt e' env' k')  $\implies$ 
27   {abstract v}  $\subseteq$  V x
28 proof outline
29 qed
30
31
32

```