Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 15, 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 reqChn = channel ()
    fun loop state = let
      val (v, replCh) = recv reqChn in
      send (replCh, state);
      loop v end in
    spawn (fn () => loop 0);
    S reqChn end

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

```

```
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 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
```

```
    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 |
    Inac

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

  fun channel () = Chn (ref Inac, mutexLock ())

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

```
30              release lock; wait sendCond; () end)
31
32        fun recv (Chn (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
```

```
 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
```

```
 1    structure FanInChan : CHAN = struct
 2
 3    datatype 'a chan_content =
 4      Send of (condition * 'a) queue |
 5      Recv of condition * 'a option ref |
 6      Inac
 7
 8    datatype 'a chan =
 9      Chn of 'a chan_content ref * mutex_lock
10
11    fun channel () = Chn (ref Inac, mutexLock ())
12
13    fun send (Chn (conRef, lock)) m =
14      acquire lock;
15      case !conRef of
16      Recv (recvCond, mopRef) =>
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 (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 relases 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
```

```
1
2     datatype var = Var string
3
4     datatype exp =
5       Let var boundexp exp |
6       Rslt var
7
8     boundexp =
9       Unit |
10      MkChn |
11      Prim prim |
12      Spwn exp |
13      Sync var |
14      Fst var |
15      Snd var |
```

```
    Case var var exp var exp |
    App var var and

prim =
    SendEvt var var |
    RecvEvt var |
    Pair var var |
    Lft var |
    Rht var |
    Abs var var ex
```

```
datatype ctrl_label =
    LNxt var | LSpwn var | LCall var | LRtn var

type_synonym ctrl_path = (ctrl_label list)

datatype chan = Chn ctrl_path var

datatype val =
    VUnit | VChn chan | VClsr prim (var ⇀ val)

datatype ctn = Ctn var exp (var ⇀ val)

datatype state = Stt exp (var ⇀ val) (ctn list)
```

```

inductive seq_step ::
    bind * (var ⇀ val)) ⇒ val ⇒ bool where
    LetUnit:
        seq_step (Unit, env) VUnit |
    LetPrim:
        seq_step (Prim p, env) (VClsr p env) |
    LetFst:
        env xp = Some (VClsr (Pair x1 x2) envp) ⟹
        envp x1 = Some v ⟹
        seq_step (Fst xp, env) v |
    LetSnd:
        env xp = Some (VClsr (Pair x1 x2) envp) ⟹
        envp x2 = Some v ⟹
        seq_step (Snd xp, env) v

inductive seq_step_up ::
    bind * (var ⇀ val)) ⇒ exp * val_env ⇒ bool where
    LetCaseLft:
        env xs = Some (VClsr (Lft xl') envl) ⟹
        envl xl' = Some vl ⟹
```

```
      seq_step_up
        (Case xs xl el xr er, env)
        (el, env(xl ↦ vl)) |
    LetCaseRht:
      env xs = Some (VClsr (Rht xr') envr) ⟹
      envr xr' = Some vr ⟹
      seq_step_up
        (Case xs xl el xr er, env)
        (er, env(xr ↦ vr)) |
    LetApp:
      env f = Some (VClsr (Abs fp xp el) envl) ⟹
      env xa = Some va  ⟹
      seq_step_up
        (App f xa, env)
        (el, envl(
          fp ↦ (VClsr (Abs fp xp el) envl),
          xp ↦ va))


type_synonym cmmn_set = (ctrl_path * chan * ctrl_path)
set

type_synonym trace_pool = ctrl_path ⇀ state

inductive leaf ::
  trace_pool ⇒ ctrl_path ⇒ bool where
  Intro:
    trpl pi ≠ None ⟹
    (∄ pi' . trpl pi' ≠ None ∧ strict_prefix pi pi') ⟹
    leaf trpl pi

inductive concur_step ::
  trace_pool * cmmn_set ⇒
  trace_pool * cmmn_set ⇒
  bool where
  Seq_Sttep_Down:
    leaf trpl pi ⟹
    trpl pi = Some
      (Stt (Rslt x) env
        ((Ctn xk ek envk) # k)) ⟹
    env x = Some v ⟹
    concur_step
      (trpl, ys)
      (trpl(pi @ [LRtn xk] ↦
        (Stt ek (envk(xk ↦ v)) k)), ys) |
  Seq_Step:
    leaf trpl pi ⟹
    trpl pi = Some
      (Stt (Let x  b e) env k) ⟹
    seq_step (b, env) v⟹
```

```
71          concur_step
72            (trpl, ys)
73            (trpl(pi @ [LNxt x] ↦
74              (Stt e (env(x ↦ v)) k), ys) |
75        Seq_Step_Up:
76          leaf trpl pi ⟹
77          trpl pi = Some
78            (Stt (Let x b e) env k) ⟹
79          seq_step_up (b, env) (e', env') ⟹
80          concur_step
81            (trpl, ys)
82            (trpl(pi @ [LCall x] ↦
83              (Stt e' env'
84                ((Ctn x e env) # k))), ys) |
85        LetMkCh:
86          leaf trpl pi ⟹
87          trpl pi = Some (Stt (Let x MkChn e) env k) ⟹
88          concur_step
89            (trpl, ys)
90            (trpl(pi @ [LNxt x] ↦
91              (Stt e (env(x ↦ (VChn (Chn pi x)))) k)), ys) |
92        LetSpwn:
93          leaf trpl pi ⟹
94          trpl pi = Some
95            (Stt (Let x (Spwn ec) e) env k) ⟹
96          concur_step
97            (trpl, ys)
98            (trpl(
99              pi @ [LNxt x] ↦
100                 (St e (env(x ↦ VUnit)) k),
101             pi @ [LSpwn x] ↦
102                 (St ec env []), ys) |
103       LetSync:
104         leaf trpl pis ⟹
105         trpl pis = Some
106           (Stt (Let xs (Sync xse) es) envs ks) ⟹
107         envs xse = Some
108           (VClsr (SendEvt xsc xm) envse) ⟹
109         leaf trpl pir  ⟹
110         trpl pir = Some
111           (Stt (Let xr (Sync xre) er) envr kr) ⟹
112         envr xre = Some
113           (VClsr (RecvEvt xrc) envre) ⟹
114         envse xsc = Some (VChn c) ⟹
115         envre xrc = Some (VChn c) ⟹
116         envse xm = Some vm ⟹
117         concur_step
118           (trpl, ys)
119           (trpl(
120             pis @ [LNxt xs] ↦
```

```
              (Stt es (envs(xs ↦ VUnit)) ks),
            pir @ [LNxt xr] ↦
              (Stt er (envr(xr ↦ vm)) kr)),
            ys ∪ {(pis, c, pir)})
```

```
inductive star :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool
  for r where
  refl: star r x x |
  step: r x y ⟹ star r y z ⟹ star r x z
```