Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 21, 2018

1 Mathematical Artifacts

```
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)

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
4     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)]);
        release lock; wait sendCond; () end)

  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
```

```
      val sendCond = condition () in
      enqueue (q, (sendCond, m));
      release lock; wait sendCond;
      () end |
    Inac => let
      val sendCond = condition () in
      conRef := Send (queue [(sendCond, m)])
      release lock; wait sendCond; () end

  fun recv (Chn (conRef, lock)) = let
    val recvCond = condition ()
    val mopRef = ref None in
    case cas (conRef, Inac, Recv (recvCond, mopRef)) of
      Inac => (* conRef already set *)
        wait recvCond; valOf (!mopRef) |
      Send q =>
        (* the current thread is the only one
        -* that updates the state from this state *)
        acquire lock;
        (let
          val (sendCond, m) = dequeue q in
          if (isEmpty q) then conRef := Inac else ();
          release lock; signal sendCond; m end) |
      Recv _ => raise NeverHappens end end
```

```
structure OneToOneChan : CHAN = struct

  datatype 'a chan_content =
    Send of condition * 'a |
    Recv of condition * 'a option ref |
    Inac

  datatype 'a chan = Chn of 'a chan_content ref

  fun channel () = Chn (ref Inac)

  fun send (Chn conRef) m = let
    val sendCond = condition () in
    case cas (conRef, Inac, Send (sendCond, m)) of
      Inac =>
        (* conRef already set to Send *)
        wait sendCond; () |
      Recv (recvCond, mopRef) =>
        (* the current thread is the only one
        -* that accesses conRef for this state *)
        mopRef := Some m; conRef := Inac;
        signal recvCond; () |
```

```
      Send _ => raise NeverHappens end end


fun recv (Chn conRef) = let
  val recvCond = condition ();
  val mopRef = ref None in
  case cas (conRef, Inac, Recv (recvCond, mopRef)) of
    Inac => (* conRef already set to Recv*)
      wait recvCond; valOf (!mopRef) |
    Send (sendCond, m) =>
      (* the current thread is the only one
      -* that accesses conRef for this state *)
      conRef := Inac; signal sendCond; m |
    Recv _ => raise NeverHappens end end

end
```

```
structure OneShotChan : CHAN = struct

datatype 'a chan_content =
  Send of condition * 'a |
  Recv of condition * 'a option ref |
  Inac

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

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

fun send (Chn (conRef, lock)) m = let
  val sendCond = condition () in
  case (conRef, Inac, Send (sendCond, m)) of
    Inac =>
      (* conRef already set to Send*)
      wait sendCond; () |
    Recv (recvCond, mopRef) =>
      mopRef := Some m; signal recvCond;
      () |
    Send _ => raise NeverHappens end end


fun recv (Chn (conRef, lock)) = let
  val recvCond = condition ()
  val mopRef = ref None in
  case (conRef, Inac, Recv (recvCond, mopRef)) of
    Inac =>
      (* conRef already set to Recv*)
      wait recvCond; valOf (!mopRef) |
    Send (sendCond, m) =>
      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
```

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 |
```

```
    App var var and

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

3 Dynamic Semantics

```
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 =
  VUnt | 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
  LetUnt :
    seq_step (Unt, env) VUnt |
  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


```

```
2
3   inductive
4     seq_step_up ::
5       bind * (var ⇀ val)) ⇒
6       exp * val_env ⇒ bool where
7     LetCaseLft :
8       env xs = Some (VClsr (Lft xl') envl) ⟹
9       envl xl' = Some vl ⟹
10      seq_step_up
11        (Case xs xl el xr er, env)
12        (el, env(xl ↦ vl)) |
13    LetCaseRht :
14      env xs = Some (VClsr (Rht xr') envr) ⟹
15      envr xr' = Some vr ⟹
16      seq_step_up
17        (Case xs xl el xr er, env)
18        (er, env(xr ↦ vr)) |
19    LetApp :
20      env f = Some (VClsr (Abs fp xp el) envl) ⟹
21      env xa = Some va  ⟹
22      seq_step_up
23        (App f xa, env)
24        (el, envl(
25          fp ↦ (VClsr (Abs fp xp el) envl),
26          xp ↦ va))
27
28
```

```
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
8     leaf ::
9       trace_pool ⇒ ctrl_path ⇒ bool where
10    intro :
11      trpl pi ≠ None ⟹
12      (∄ pi' . trpl pi' ≠ None ∧ strict_prefix pi pi') ⟹
13      leaf trpl pi
14
15
```

```
1
2   inductive
3     concur_step ::
4       trace_pool * cmmn_set ⇒
5       trace_pool * cmmn_set ⇒
6       bool where
```

```
7     Seq_Sttep_Down :
8       leaf trpl pi ⟹
9       trpl pi = Some
10        (Stt (Rslt x) env
11          ((Ctn xk ek envk) # k)) ⟹
12      env x = Some v ⟹
13      concur_step
14        (trpl, ys)
15        (trpl(pi @ [LRtn xk] ↦
16          (Stt ek (envk(xk ↦ v)) k)), ys) |
17    Seq_Step :
18      leaf trpl pi ⟹
19      trpl pi = Some
20        (Stt (Let x  b e) env k) ⟹
21      seq_step (b, env) v⟹
22      concur_step
23        (trpl, ys)
24        (trpl(pi @ [LNxt x] ↦
25          (Stt e (env(x ↦ v)) k), ys) |
26    Seq_Step_Up :
27      leaf trpl pi ⟹
28      trpl pi = Some
29        (Stt (Let x b e) env k) ⟹
30      seq_step_up (b, env) (e', env') ⟹
31      concur_step
32        (trpl, ys)
33        (trpl(pi @ [LCall x] ↦
34          (Stt e' env'
35            ((Ctn x e env) # k))), ys) |
36    LetMkCh :
37      leaf trpl pi ⟹
38      trpl pi = Some (Stt (Let x MkChn e) env k) ⟹
39      concur_step
40        (trpl, ys)
41        (trpl(pi @ [LNxt x] ↦
42          (Stt e (env(x ↦ (VChn (Chn pi x)))) k)), ys) |
43    LetSpwn :
44      leaf trpl pi ⟹
45      trpl pi = Some
46        (Stt (Let x (Spwn ec) e) env k) ⟹
47      concur_step
48        (trpl, ys)
49        (trpl(
50          pi @ [LNxt x] ↦
51              (St e (env(x ↦ VUnt)) k),
52          pi @ [LSpwn x] ↦
53              (St ec env []), ys) |
54    LetSync :
55      leaf trpl pis ⟹
56      trpl pis = Some
```

```
      (Stt (Let xs (Sync xse) es) envs ks) ⟹
    envs xse = Some
      (VClsr (SendEvt xsc xm) envse) ⟹
    leaf trpl pir  ⟹
    trpl pir = Some
      (Stt (Let xr (Sync xre) er) envr kr) ⟹
    envr xre = Some
      (VClsr (RecvEvt xrc) envre) ⟹
    envse xsc = Some (VChn c) ⟹
    envre xrc = Some (VChn c) ⟹
    envse xm = Some vm ⟹
    concur_step
      (trpl, ys)
      (trpl(
        pis @ [LNxt xs] ↦
          (Stt es (envs(xs ↦ VUnt)) 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
```

4 Dynamic Communication

```
inductive
  is_send_path ::
    trace_pool ⇒ chan ⇒
    control_path ⇒ bool where
  intro :
    trpl piy = Some
      (Stt (Let xy (Sync xe) en) env k) ⟹
    env xe = Some
      (VClsr (SendEvt xsc xm) enve) ⟹
    enve xsc = Some (VChn c) ⟹
    is_send_path trpl c piy

inductive
  is_recv_path ::
    trace_pool ⇒ chan ⇒
    control_path ⇒ bool where
  intro :
```

```
      trpl piy = Some
        (Stt (Let xy (Sync xe) en) env k) ⟹
      env xe = Some
        (VClsr (RecvEvt xrc) enve) ⟹
      enve xrc = Some (VChn c) ⟹
      is_recv_path trpl c piy
```

```
inductive
  every_two ::
    ('a ⇒ bool) ⇒
    ('a ⇒ 'a ⇒ bool) ⇒ bool where
  intro : (∀ pi1 pi2 .
      p x1 ⟶
      p x2 ⟶
      r x1 x2) ⟹
    every_two p r

inductive
  ordered ::
    'a list ⇒ 'a list ⇒ bool where
  left : prefix pi1 pi2 ⟹ ordered pi1 pi2 |
  right : prefix pi2 pi1 ⟹ ordered pi1 pi2
```

```
inductive one_shot :: trace_pool ⇒ chan ⇒ bool where
  intro :
    every_two
      (is_send_path trpl c) op= ⟹
    one_shot trpl c

inductive fan_out :: trace_pool ⇒ chan ⇒ bool where
  intro :
    every_two
      (is_send_path trpl c) ordered ⟹
    fan_out trpl c

inductive fan_in :: trace_pool ⇒ chan ⇒ bool where
  intro :
    every_two
      (is_recv_path trpl c) ordered ⟹
    fan_in trpl c

inductive one_to_one :: trace_pool ⇒ chan ⇒ bool where
  intro :
    fan_out trpl c ⟹
```

```
      fan_in trpl c ⟹
      one_to_one trpl c
```

5 Static Semantics

```

datatype abstract_value =
  AChn var |
  AUnt |
  APrim prim

type_synonym abstract_env = var ⇒ abstract_value set

fun rslt_var :: exp ⇒ var where
  rslt_var (Rslt x) = x |
  rslt_var (Let _ _ e) = (rslt_var e)
```

```

inductive
  static_eval_exp ::
    abstract_env * abstract_env ⇒
    exp ⇒ bool where
  Rslt :
    static_eval_exp (V, C) (RESULT x) |
  let_unt :
    {AUnt} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x Unt e) |
  let_chan :
    {AChn x} ⊆ V x  ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (MkChn) e) |
  let_send_evt :
    {APrim (SendEvt xc xm)} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C)
      (Let x (Prim (SendEvt xc xm)) e) |
  let_recv_evt :
    {APrim (RecvEvt xc)} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C)
      (Let x (Prim (RecvEvt xc)) e) |
  let_pair :
    {APrim (Pair x1 x2)} ⊆ V x ⟹
```

```
29        static_eval_exp (V, C) e ⟹
30        static_eval_exp (V, C) (Let x (Pair x1 x2) e) |
31      let_left :
32        {APrim (Left xp)} ⊆ V x ⟹
33        static_eval_exp (V, C) e ⟹
34        static_eval_exp (V, C) (Let x (Left xp) e) |
35      let_right :
36        {APrim (Right xp)} ⊆ V x ⟹
37        static_eval_exp (V, C) e ⟹
38        static_eval_exp (V, C) (Let x (Right xp) e) |
39      let_abs :
40        {APrim (Abs f' x' e')} ⊆ V f' ⟹
41        static_eval_exp (V, C) e' ⟹
42        {APrim (Abs f' x' e')} ⊆ V x ⟹
43        static_eval_exp (V, C) e ⟹
44        static_eval_exp (V, C) (Let x (Abs f' x' e') e) |
45      let_spawn :
46        {AUnt} ⊆ V x ⟹
47        static_eval_exp (V, C) ec ⟹
48        static_eval_exp (V, C) e ⟹
49        static_eval_exp (V, C) (Let x (Spwn ec) e) |
50      let_sync  :
51        ∀ xsc xm xc .
52          (APrim (SendEvt xsc xm)) ∈ V xe ⟶
53          AChn xc ∈ V xsc ⟶
54          {AUnt} ⊆ V x ∧ V xm ⊆ C xc ⟹
55        ∀ xrc xc .
56          (APrim (RecvEvt xrc)) ∈ V xe ⟶
57          AChn xc ∈ V xrc ⟶
58          C xc ⊆ V x ⟹
59        static_eval_exp (V, C) e ⟹
60        static_eval_exp (V, C) (Let x (Syync xe) e) |
61      let_fst :
62        ∀ x1 x2.
63          (APrim (Pair x1 x2)) ∈ V xp ⟶
64          V x1 ⊆ V x ⟹
65        static_eval_exp (V, C) e ⟹
66        static_eval_exp (V, C) (Let x (Fst xp) e) |
67      let_snd :
68      ∀ x1 x2 .
69        (APrim (Pair x1 x2) ∈ V xp ⟶
70        V x2 ⊆ V x ⟹
71      static_eval_exp (V, C) e ⟹
72      static_eval_exp (V, C) (Let x (Snd xp) e) |
73    let_case :
74      ∀ xl' .
75        (APrim (Left xl')) ∈ V xs ⟶
76          V xl' ⊆ V xl ∧ V (rslt_var el) ⊆ V x ∧
77          static_eval_exp (V, C) el ⟹
78      ∀ xr' .
```

```
79        (APrim (Right xr')) ∈ V xs ⟶
80          V xr' ⊆ V xr ∧ V (rslt_var er) ⊆ V x ∧
81          static_eval_exp (V, C) er ⟹
82          static_eval_exp (V, C) e ⟹
83        static_eval_exp (V, C) (Let x (Case xs xl el xr er) e)
       |
84    let_app :
85      ∀ f' x' e' .
86        (APrim (Abs f' x' e') ∈ V f ⟶
87        V xa ⊆ V x' ∧
88        V (rslt_var e') ⊆ V x ⟹
89      static_eval_exp (V, C) e ⟹
90      static_eval_exp (V, C) (Let x (App f xa) e)
91
92
```

```
1
2
3   fun abstract :: val ⇒ abstract_value where
4     abstract VUnt = AUnt |
5     abstract VChn (Chn pi x) = AChn x |
6     abstract VClsr p env = APrim p
7
8
```

```
1
2   inductive
3     static_eval_val ::
4       abstract_env * abstract_env ⇒ val ⇒ bool and
5     static_eval_env ::
6       abstract_env * abstract_env ⇒ val_env ⇒ bool where
7     Unt :
8       static_eval_val (V, C) VUnt |
9     Chan :
10      static_eval_val (V, C) VChn c |
11    SendEvt :
12      static_eval_env (V, C) env ⟹
13      static_eval_val (V, C) (VClsr (SendEvt _ _) env) |
14    RecvEvt :
15      static_eval_env (V, C) env ⟹
16      static_eval_val (V, C) (VClsr (RecvEvt _) env) |
17    Left :
18      static_eval_env (V, C) env ⟹
19      static_eval_val (V, C) (VClsr (Left _) env) |
20    Right :
21      static_eval_env (V, C) env ⟹
22      static_eval_val (V, C) (VClsr (Right _) env) |
23    Abs :
24      {(APrim (Abs f x e)} ⊆ V f ⟹
25      static_eval_exp (V, C) e ⟹
```

```
      static_eval_env (V, C) env ⟹
      static_eval_val (V, C) (VClsr (Abs f x e) env) |
    Pair :
      static_eval_env (V, C) env ⟹
      static_eval_val (V, C) (VClsr (Pair _ _) env) |
    intro :
      ∀ x v .
        env x = Some v ⟶
        {abstract v} ⊆ V x ∧ static_eval_val (V, C) v ⟹
      static_eval_env (V, C) env
```

```
inductive static_eval_stack ::
  abstract_env * abstract_env ⇒
  abstract_value set ⇒ cont list ⇒ bool where
  Empty :
    static_eval_stack (V, C) valset [] |
  Nonempty :
    valset ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_env (V, C) env ⟹
    static_eval_stack (V, C) (V (rslt_var e)) k ⟹
    static_eval_stack (V, C) valset ((Ctn x e env) # k)


inductive static_eval_state ::
  abstract_env * abstract_env ⇒
  state ⇒ bool where
  intro :
    static_eval_exp (V, C) e ⟹
    static_eval_env (V, C) env ⟹
    static_eval_stack (V, C) (V (rslt_var e)) k ⟹
    static_eval_state (V, C) (Stt e env k)

inductive static_eval_pool ::
  abstract_env * abstract_env ⇒
  trace_pool ⇒ bool where
  intro :
    ∀ pi st .
      trpl pi = Some st ⟶
      static_eval_state (V, C) st ⟹
    static_eval_pool (V, C) trpl
```

```
theorem static_eval_preserved_under_concur_step : "
  static_eval_pool (V, C) trpl ⟹
```

```
    concur_step (trpl, ys) (trpl', ys') ⟹
    static_stati_eval_pool (V, C) trpl'"
  proof sketch
  qed

  theorem static_eval_preserved_under_concur_step_star : "
    static_eval_pool (V, C) trpl ⟹
    star concur_step (trpl, ys) (trpl', ys')  ⟹
    static_concur_step (V, C) trpl'"
  proof sketch
  qed
```

```

  theorem trace_pool_snapshot_value_not_bound_sound : "
    env x = Some v ⟹
    trpl pi = Some (Stt e env k) ⟹
    static_eval_pool (V, C) trpl ⟹
    {abstract v} ⊆ V x "
  proof sketch
  qed
```

```

  theorem trace_pool_always_value_not_bound_sound : "
    env' x = Some v ⟹
    static_eval_pool (V, C) trpl ⟹
    star concur_step (trpl, ys) (trpl', ys') ⟹
    trpl' pi = Some (Stt e' env' k') ⟹
    {abstract v} ⊆ V x"
  proof sketch
  qed

  theorem exp_always_value_not_bound_sound : "
    env' x = Some v ⟹
    static_eval_exp (V, C) e ⟹
    star concur_step
      ([[] ↦ (Stt e (λ _ . None) [])], ys)
      (trpl', ys') ⟹
    trpl' pi = Some (Stt e' env' k') ⟹
    {abstract v} ⊆ V x"
  proof sketch
  qed


```

```
  inductive is_super_exp :: exp ⇒ exp ⇒ bool where
    Refl :
      is_super_exp e e |
    let_Spawn_Child
```

```
5       is_super_exp ec e ⟹
6       is_super_exp (Let x (Spwn ec) en) e |
7     let_Case_Left :
8       is_super_exp el e ⟹
9       is_super_exp (Let x (case xs xl el xr er) en) e |
10    let_Case_Right :
11      is_super_exp er e ⟹
12      is_super_exp (Let x (case xs xl el xr er) en) e |
13    let_Abs_Body : "
14      is_super_exp eb e ⟹
15      is_super_exp (Let x (Abs f xp eb) en) e |
16    Let :
17      is_super_exp en e ⟹
18      is_super_exp (Let x b en) e
19
20  inductive
21    is_super_exp_left :: exp ⇒ exp ⇒ bool where
22    Refl :
23      is_super_exp_left e0 e0 |
24    let_Spawn_Child :
25      is_super_exp_left e0 (Let x (Spwn ec) en)⟹
26      is_super_exp_left e0 ec |
27    let_Case_Left :
28      is_super_exp_left e0 (Let x (case xs xl el xr er) en)
    ⟹
29      is_super_exp_left e0 el |
30    let_Case_Right :
31      is_super_exp_left e0 (Let x (case xs xl el xr er) en)
    ⟹
32      is_super_exp_left e0 er |
33    let_Abs_Body :
34      is_super_exp_left e0 (Let x (Abs f xp eb) en) ⟹
35      is_super_exp_left e0 eb |
36    Let :
37      is_super_exp_left e0 (Let x b en) ⟹
38      is_super_exp_left e0 en
39
40  inductive
41    is_super_exp_over_prim :: exp ⇒ prim ⇒ bool where
42    SendEvt :
43      is_super_exp_over_prim e0 (SendEvt xC xM) |
44    RecvEvt :
45      is_super_exp_over_prim e0 (RecvEvt xC) |
46    Pair :
47      is_super_exp_over_prim e0 (Pair x1 x2) |
48    Left :
49      is_super_exp_over_prim e0 (Left x) |
50    Right :
51      is_super_exp_over_prim e0 (Right x) |
52    Abs :
```

```
    is_super_exp_left e0 eb ⟹
    is_super_exp_over_prim e0 (Abs fp xp eb)

inductive
  is_super_exp_over_env :: exp ⇒ env ⇒ bool and
  is_super_exp_over_val :: exp ⇒ val ⇒ bool where
  VUnt :
    is_super_exp_over_val e0 VUnt |
  VChn :
    is_super_exp_over_val e0 (VChn c) |
  VClsr : "
    is_super_exp_over_prim e0 p ⟹
    is_super_exp_over_env e0 env' ⟹
    is_super_exp_over_val e0 (VClsr p env') |
  intro : "
    ∀ x v .
      env x = Some v ⟶
      is_super_exp_over_val e0 v ⟹
    is_super_exp_over_env e0 env

inductive
  is_super_exp_over_stack ::
    exp ⇒ cont list ⇒ bool where
  Empty :
    is_super_exp_over_stack e0 [] |
  Nonempty :
    is_super_exp_left e0 ek ⟹
    is_super_exp_over_env e0 envk ⟹
    is_super_exp_over_stack e0 k ⟹
    is_super_exp_over_stack e0 ((Ctn xk ek envk) # k)

inductive
  is_super_exp_over_state ::
    exp ⇒ state ⇒ bool where
  intro :
    is_super_exp_left e0 e ⟹
    is_super_exp_over_env e0 env ⟹
    is_super_exp_over_stack e0 k ⟹
    is_super_exp_over_state e0 (Ctn e env k)
```

```
lemma is_super_exp_trans : "
  is_super_exp ez ey ⟹
  is_super_exp ey ex ⟹
  is_super_exp ez ex"
proof sketch
qed
```

```
lemma is_super_exp_over_state_preserved : "
  concur_step (trpl, ys) (trpl', ys') ⟹
  ∀ pi st.
    trpl pi = Some st ⟶
    is_super_exp_over_state e0 st ⟹
  trpl' pi' = Some st' ⟹
  is_super_exp_over_state e0 st'"
proof sketch
qed
```

```
lemma state_always_exp_not_reachable_sound : "
  star concur_step (trpl0, ys0) (trpl', ys') ⟹
  trpl0 = [[] ↦ (Stt e0 (λ _ . None) [])] ⟹
  trpl' pi' = Some st' ⟹
  is_super_exp_over_state e0 st' "
proof sketch
qed

lemma exp_always_exp_not_reachable_sound : "
  star concur_step
  ([[] ↦ (Stt e0 (λ _ . None) [])], {})
    (trpl', ys') ⟹
  trpl' pi' = Some (Stt e' env' k')) ⟹
  is_super_exp e0 e' "
proof sketch
qed
```

6 Static Communication

```
datatype node_label = NLet var | NRslt var

fun top_node_label :: exp ⇒ node_label where
  top_node_label (Let x b e) = NLet x |
  top_node_label (Rslt y) = NRslt y

type_synonym node_set = node_label set

type_synonym node_map = node_label ⇒ var set

inductive
  static_static_send_node_label ::
    abstract_env ⇒ exp ⇒
    var ⇒ node_label ⇒ bool where
  intro:
    {AChn xC} ⊆ V xSC ⟹
```

```
17      {APrim (SendEvt xSC xM)} ⊆ V xE ⟹
18      is_super_exp e (Let x (Sync xE) e') ⟹
19      static_static_send_node_label V e xC (NLet x)
20
21  inductive
22    static_static_recv_node_label ::
23      abstract_env ⇒ exp ⇒
24      var ⇒ node_label ⇒ bool where
25    intro:
26      {AChn xC} ⊆ V xRC ⟹
27      {APrim (RecvEvt xRC)} ⊆ V xE ⟹
28      is_super_exp e (Let x (Sync xE) e') ⟹
29      static_static_recv_node_label V e xC (NLet x)
30
31
```

7 Static Communication part A

```
1
2   datatype edge_label = ENext | ESpawn | ECall | ERtn
3
4   type_synonym flow_label = node_label * edge_label *
      node_label
5
6   type_synonym flow_set = flow_label set
7
8   type_synonym step_label = node_label * edge_label
9
10  type_synonym abstract_path = step_label list
11
```

```
1   inductive
2     static_traversable ::
3       abstract_env ⇒
4       (node_label * edge_label * node_label) set ⇒
5       exp ⇒ bool  where
6     Rslt:
7       static_traversable V F (Rslt x) |
8     let_Unit:
9       {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
10      static_traversable V F e  ⟹
11      static_traversable V F (Let x Unt e) |
12    let_Chan:
13      {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
14      static_traversable V F e ⟹
15      static_traversable V F (Let x MkChn e) |
16    let_Send_Evt:
17      {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
18      static_traversable V F e ⟹
```

```
19        static_traversable V F (Let x = (SendEvt xc xm) e) |
20      let_Recv_Evt:
21        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
22        static_traversable V F e ⟹
23        static_traversable V F (Let x = (RecvEvt xc) in e) |
24      let_Pair:
25        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
26        static_traversable V F e ⟹
27        static_traversable V F (Let x (Pair x1 x2) e) |
28      let_Left:
29        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
30        static_traversable V F e ⟹
31        static_traversable V F (Let x (Lft xp) e) |
32      let_Right:
33        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
34        static_traversable V F e ⟹
35        static_traversable V F (Let x (Rht xp) e)" |
36      let_Abs:
37        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
38        static_traversable V F eb ⟹
39        static_traversable V F e ⟹
40        static_traversable V F (Let x (Abs f xp eb) e) |
41      let_Spawn:
42        {
43          (NLet x, ENext, top_node_label e),
44          (NLet x, ESpawn, top_node_label ec)}
45          ⊆ F ⟹
46        static_traversable V F ec ⟹
47        static_traversable V F e ⟹
48        static_traversable V F (Let x (Spwn ec)  e) |
49      let_Sync:
50        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
51        static_traversable V F e ⟹
52        static_traversable V F (Let x (Sync xSE) e) |
53      let_Fst:
54        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
55        static_traversable V F e ⟹
56        static_traversable V F (Let x (Fst xp) e) |
57      let_Snd:
58        {(NLet x, ENext, top_node_label e)} ⊆ F ⟹
59        static_traversable V F e ⟹
60        static_traversable V F (Let x (Snd xp) e) |
61      let_Case:
62        {
63          (NLet x, ECall, top_node_label el),
64          (NLet x, ECall, top_node_label er),
65          (NRslt (rslt_var el), ERtn, top_node_label e),
66          (NRslt (rslt_var er), ERtn, top_node_label e)}
67          ⊆ F ⟹
68        static_traversable V F el ⟹
```

```
69      static_traversable V F er ⟹
70      static_traversable V F e ⟹
71      static_traversable V F (Let x (Case xs xl el xr er) e)
      |
72    let_App:
73      ∀ fp xp eb .
74        APrim (Abs fp xp eb) ∈ V f ⟶
75        {
76          (NLet x, ECall, top_node_label eb),
77          (NRslt (rslt_var eb), ERtn, top_node_label e)}
78          ⊆ F ⟹
79        static_traversable V F e ⟹
80        static_traversable V F (Let x (App f xa) e)
81
82
```

```
1   inductive static_traceable ::
2     abstract_env ⇒
3       flow_set ⇒ node_label ⇒
4       (node_label ⇒ bool) ⇒
5       abstract_path ⇒ bool where
6   Empty:
7     isEnd start ⟹
8     static_traceable V F start isEnd [] |
9   Edge:
10    isEnd end ⟹
11    {(start, edge, end)} ⊆ F ⟹
12    static_traceable V F start isEnd [(start, edge)] |
13  Step:
14    static_traceable V F middle isEnd ((middle, edge') #
      path) ⟹
15    {(start, edge, middle)} ⊆ F ⟹
16    static_traceable V F start isEnd ((start, edge) # (
      middle, edge') # path)
17
```

```
1   inductive static_inclusive ::
2     abstract_path ⇒ abstract_path ⇒ bool where
3   Prefix1:
4     prefix pi1 pi2 ⟹
5     static_inclusive pi1 pi2 |
6   Prefix2:
7     prefix pi2 pi1 ⟹
8     static_inclusive pi1 pi2 |
9   Spawn1:
10    static_inclusive (pi @ (NLet x, ESpawn) # pi1) (pi @ (
      NLet x, ENext) # pi2) |
11  Spawn2:
12    static_inclusive (pi @ (NLet x, ENext) # pi1) (pi @ (
      NLet x, ESpawn) # pi2)
```

```
13


 1  inductive
 2    singular ::
 3      abstract_path ⇒
 4      abstract_path ⇒ bool where
 5  equal:
 6    pi1 = pi2 ⟹
 7    singular pi1 pi2 |
 8  exclusive:
 9    /(static_inclusive pi1 pi2) ⟹
10    singular pi1 pi2
11
12  inductive
13    noncompetitive ::
14      abstract_path ⇒ abstract_path ⇒ bool where
15    ordered:
16      ordered pi1 pi2 ⟹
17      noncompetitive pi1 pi2 |
18    exclusive:
19      /(not_inclusive pi1 pi2) ⟹
20      noncompetitive pi1 pi2
21


 1  inductive
 2    static_one_shot ::
 3      abstract_env ⇒ exp ⇒
 4      var ⇒ bool where
 5    Sync:
 6      every_two
 7        (static_traceable V F (top_node_label e)
 8          (static_static_send_node_label V e xC))
 9        singular ⟹
10      static_traversable V F e ⟹
11      static_one_shot V e xC
12
13    inductive
14      static_one_to_one ::
15        abstract_env ⇒ exp ⇒
16        var ⇒ bool where
17      Sync:
18        every_two
19          (static_traceable V F
20            (top_node_label e) (
      static_static_send_node_label V e xC))
21          noncompetitive ⟹
22        every_two
23          (static_traceable V F
24            (top_node_label e) (
      static_static_recv_node_label V e xC))
```

```
25          noncompetitive ⟹
26        static_traversable V F e ⟹
27        static_one_to_one V e xC
28
29  inductive
30    static_fan_out ::
31      abstract_env ⇒ exp ⇒
32      var ⇒ bool where
33    Sync:
34      every_two
35        (static_traceable V F
36          (top_node_label e) (static_static_send_node_label
    V e xC))
37        noncompetitive ⟹
38      static_traversable V F e ⟹
39      static_fan_out V e xC
40
41  inductive
42    static_fan_in ::
43      abstract_env ⇒ exp ⇒
44      var ⇒ bool where
45    Sync:
46      every_two (static_traceable V F (top_node_label e) (
    static_static_recv_node_label V e xC)) noncompetitive ⟹
47      static_traversable V F e ⟹
48      static_fan_in V e xC
49
```

```
1   inductive paths_correspond ::
2     control_path ⇒ abstract_path ⇒ bool where
3     Empty:
4       paths_correspond [] [] |
5     Next:
6       paths_correspond pi path ⟹
7       paths_correspond (pi @ [LNext x]) (path @ [(NLet x,
    ENext)]) |
8     Spawn:
9       paths_correspond pi path ⟹
10      paths_correspond (pi @ [LSpawn x]) (path @ [(NLet x,
    ESpawn)]) |
11    Call:
12      paths_correspond pi path ⟹
13      paths_correspond (pi @ [LCall x]) (path @ [(NLet x,
    ECall)])  |
14    Rtn:
15      paths_correspond pi path ⟹
16      paths_correspond (pi @ [LRtn x]) (path @ [(NRslt x,
    ERtn)])
17
```

```
lemma not_static_inclusive_sound: "
  star concur_step
    ([[] ↦ (Stt e (λ _ . None) [])], {})
    (trpl', ys') ⟹
  trpl' pi1 ≠ None ⟹
  trpl' pi2 ≠ None ⟹
  paths_correspond pi1 path1 ⟹
  paths_correspond pi2 path2 ⟹
  static_inclusive path1 path2"

lemma equality_abstract_to_concrete: "
  path1 = path2 ⟹
  paths_correspond pi1 path1 ⟹
  paths_correspond pi2 path2 ⟹
  pi1 = pi2"

lemma
 abstract_paths_equal_or_exclusive_implies_dynamic_paths_equal
 : "
  pathSync = pathSynca ∨ /(static_inclusive pathSynca
 pathSync) ⟹
  paths_correspond pi1 pathSync ⟹
  paths_correspond pi2 pathSynca ⟹

  star concur_step ([[] ↦ (Stt e (λ _ . None) [])], {}) (
 trpl', ys') ⟹
  trpl' pi1 ≠ None ⟹
  trpl' pi2 ≠ None ⟹
  pi1 = pi2"
```

```
inductive
  static_traversable_env ::
    abstract_env ⇒ flow_set ⇒ env ⇒ bool and
  static_traversable_val ::
    abstract_env ⇒ flow_set ⇒ val ⇒ bool where
  Intro:
    ∀ x v .
      env x = Some v ⟶
      {rslt_var v} ⊆ V x ∧ static_traversable_val V F v
 ⟹
    static_traversable_env V F env |

  Unit:
    static_traversable_val V F VUnit |
  Chan:
    static_traversable_val V F (VChan c) |
  Send_Evt:
```

```
      static_traversable_env V F env ⟹
      static_traversable_val V F (VClsr (Send_Evt _ _) env)
  |
   Recv_Evt:
      static_traversable_env V F env ⟹
      static_traversable_val V F (VClsr (Recv_Evt _) env) |
   Left:
      static_traversable_env V F env ⟹
      static_traversable_val V F (VClsr (Left _) env) |
   Right:
      static_traversable_env V F env ⟹
      static_traversable_val V F (VClsr (Right _) env) |
   Abs:
      static_traversable V F e ⟹
      static_traversable_env V F  env ⟹
      static_traversable_val V F (VClsr (Abs f x e) env) |
Pair:
   static_traversable_env V F env ⟹
   static_traversable_val V F (VClsr (Pair _ _) env)



inductive static_traversable_stack ::
   abstract_env ⇒ flow_set ⇒ cont list ⇒ bool where
   Empty:
      static_traversable_stack V F [] |
   Nonempty:
      static_traversable V F e ⟹
      static_traversable_env V F env ⟹
      static_traversable_stack V F k ⟹
      static_traversable_stack V F ((Ctn x e env) # k))

inductive
   static_traversable_pool ::
      abstract_env ⇒ flow_set ⇒
      trace_pool ⇒ bool  where
   Intro:
     ∀ pi e env k .
        E pi = Some (Stt e env k) ⟶
        static_traversable V F e ∧
        static_traversable_env V F env ∧
        static_traversable_stack V F k ⟹
      static_traversable_pool V F E



lemma static_traversable_pool_preserved_star: "
   static_traversable_pool V F
```

```
      ([[] ↦ (Stt e (λ _ . None) [])]) ⟹
    static_eval (V, C) e ⟹
    trpl' pi = Some (Stt (Let x b en) envk) ⟹
    star concur_step
      ([[] ↦ (Stt e (λ _ . None) [])], {})
      (trpl', ys') ⟹
    isEnd (NLet x) ⟹
    static_traversable_pool V F trpl' "
  proof sketch
  qed

lemma static_traversable_pool_implies_static_traceable: "
  trpl' pi = Some (Stt (Let x b in en) env k) ⟹
  concur_step
    ([[] ↦ (Stt e (λ _ . None) [])], {})
    (trpl', ys') ⟹
  static_eval (V, C) e ⟹
  static_traversable_pool V F trpl' ⟹
  isEnd (NLet x) ⟹
  ∃ path .
    paths_correspond pi path ∧
    static_traceable V F (top_node_label e) isEnd path "
  proof sketch
  qed
```

```
  lemma path_not_traceable_sound: "
  trpl' pi = Some (Stt (Let x b en) env k) ⟹
  star concur_step
    ([[] ↦ (Stt e (λ _ . None) [])], {})
    (trpl', ys') ⟹
  static_eval (V, C) e ⟹
  static_traversable V F e ⟹
  isEnd (NLet x) ⟹
  ∃ path .
    paths_correspond pi path ∧
    static_traceable V F (top_node_label e) isEnd path
  proof sketch
  qed
```

```
  theorem one_shot_sound: "
    static_one_shot V e xC ⟹
    static_eval (V, C) e ⟹
    concur_step
      ([[] ↦ (Stt e (λ _ . None) [])], {})
      (trpl', ys') ⟹
    one_shot trpl' (Ch pi xC) "
```

```
proof sketch
qed


theorem fan_out_sound: "
  static_fan_out V e xC ⟹
  static_eval (V, C) e ⟹
  star concur_step
    ([[] ↦ (Stt e (λ _ . None) [])], {})
    (trpl', ys') ⟹
  fan_out trpl' (Ch pi xC) "
proof sketch
qed

theorem fan_in_sound: "
  static_fan_in V e xC ⟹
  static_eval (V, C) e ⟹
  star concur_step
    ([[] ↦ (Stt e (λ _ . None) [])], {})
    (trpl', ys') ⟹
  fan_in trpl' (Ch pi xC) "
proof sketch
qed

theorem one_to_one_sound: "
  static_one_to_one V e xC ⟹
  static_eval (V, C) e ⟹
  star concur_step
    ([[] ↦ (Stt e (λ _ . None) [])], {})
    (trpl', ys') ⟹
  one_to_one trpl' (Ch pi xC) "
proof sketch
qed
```

8 Static Communication part B

```
```