Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 15, 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
```

```
val server = Serv.make ()
val _ = spawn (fn () => Serv.call (server, 35))
val _ = spawn (fn () =>
  Serv.call (server, 12);
  Serv.call (server, 13))
val _ = spawn (fn () => Serv.call (server, 81))
val _ = spawn (fn () => Serv.call (server, 44))
```

```
structure Serv : SERV = struct
  datatype serv = S of (int * int chan) chan

  fun make () = let

    val reqChn = FanIn.channel()

    fun loop state = let
      val (v, replCh) = FanIn.recv reqChn in
      OneShot.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 = OneShot.channel () in
    FanIn.send (reqCh, (v, replCh));
    OneShot.recv replChn end

  end
```

```
let
  val w = 4
  val x = ref 1
  val y = ref 2
  val z = (!x + 1) + (!y + 2) + (w - 3)
  val w = 1 in
  y := 0;
  (!y + 2) - (!x + 1) * (w - 3) end
```

```
let
  val x = 1
  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)]);
          release lock; wait sendCond; () end)

    fun recv (Chn (conRef, lock)) =
```

```
      acquire lock;
      (case !conRef of
        Send q => let
          val (sendCond, m) = dequeue q in
          if (isEmpty q) then
            conRef := Inac
          else
            ();
          release lock; signal sendCond; m end |
        Recv q => let
          val recvCond = condition ()
          val mopRef = ref None in
          enqueue (q, (recvCond, mopRef));
          release lock; wait recvCond;
          valOf (!mopRef) end |
        Inac => let
          val recvCond = condition ()
          val mopRef = ref None in
          conRef := Recv (queue [(recvCond, mopRef)]);
          release lock; wait recvCond;
          valOf (!mopRef) end)

    end
```

```
    structure FanOutChan : CHAN = struct

    datatype 'a chan_content =
      Send of condition * 'a |
      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 = let
      val sendCond = condition () in
      case cas (conRef, Inac, Send (sendCond, m)) of
        Inac => (* conRef already set *)
          wait sendCond; () |
        Recv q =>
        (* the current thread is
          * the only one that updates from this state *)
          acquire lock;
          (let
            val (recvCond, mopRef) = dequeue q in
```

```
          mopRef := Some m;
          if (isEmpty q) then conRef := Inac else ();
          release lock; signal (recvCond);
          () end) |
      Send _ => raise NeverHappens end

  fun recv (Chn (conRef, lock)) =
    acquire lock;
    (case !conRef of
      Inac => let
        val recvCond = condition ()
        val mopRef = ref None in
        conRef := Recv (queue [(recvCond, mopRef)]);
        release lock; wait recvCond;
        valOf (!mopRef) end |
      Recv q => let
        val recvCond = condition ()
        val mopRef = ref None in
        enqueue (q, (recvCond, mopRef));
        release lock; wait recvCond;
        valOf (!mopRef) end |
      Send (sendCond, m) =>
        conRef := Inac;
        release lock;
        signal sendCond;
        m end)

  end

```

```
structure FanInChan : CHAN = struct

datatype 'a chan_content =
  Send of (condition * 'a) queue |
  Recv of condition * 'a option ref |
  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 (recvCond, mopRef) =>
    mopRef := Some m; conRef := Inac;
    release lock; signal recvCond;
    () |
  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;
```

```
        (* never relases lock;
        -* blocks others forever *)
        m |
      Recv _ =>
        acquire lock;
        (* never able to acquire lock;
        -* blocked forever *)
        raise NeverHappens end end

  end

```

```
structure OneShotToOneChan : CHAN = struct

  datatype 'a chan =
    Chn of condition * condition * 'a option ref

  fun channel () =
    Chn (condition (), condition (), ref None)

  fun send (Chn (sendCond, recvCond, mopRef)) m =
    mopRef := Some m; signal recvCond;
    wait sendCond; ()

  fun recv (Chn (sendCond, recvCond, mopRef)) =
    wait recvCond; signal sendCond;
    valOf (!mopRef)

  end

```

2 Syntax

```

  datatype var = Var string

  datatype exp =
    Let var boundexp exp |
    Rslt var

  boundexp =
    Unt |
    MkChn |
    Prim prim |
    Spwn exp |
    Sync var |
    Fst var |
    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
```

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
```

```
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⟹
  concur_step
    (trpl, ys)
    (trpl(pi @ [LNxt x] ↦
      (Stt e (env(x ↦ v)) k), ys) |
Seq_Step_Up :
  leaf trpl pi ⟹
  trpl pi = Some
    (Stt (Let x b e) env k) ⟹
  seq_step_up (b, env) (e', env') ⟹
  concur_step
    (trpl, ys)
    (trpl(pi @ [LCall x] ↦
      (Stt e' env'
        ((Ctn x e env) # k))), ys) |
LetMkCh :
  leaf trpl pi ⟹
  trpl pi = Some (Stt (Let x MkChn e) env k) ⟹
  concur_step
    (trpl, ys)
    (trpl(pi @ [LNxt x] ↦
      (Stt e (env(x ↦ (VChn (Chn pi x)))) k)), ys) |
LetSpwn :
  leaf trpl pi ⟹
  trpl pi = Some
    (Stt (Let x (Spwn ec) e) env k) ⟹
  concur_step
    (trpl, ys)
    (trpl(
      pi @ [LNxt x] ↦
          (St e (env(x ↦ VUnt)) k),
      pi @ [LSpwn x] ↦
          (St ec env []), ys) |
LetSync :
  leaf trpl pis ⟹
  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
  may_be_eval_exp ::
    abstract_env * abstract_env ⇒
    exp ⇒ bool where
  Result :
    may_be_eval_exp (V, C) (RESULT x) |
  let_unt :
    {AUnt} ⊆ V x ⟹
    may_be_eval_exp (V, C) e ⟹
    may_be_eval_exp (V, C) (Let x Unt e) |
  let_chan :
    {AChn x} ⊆ V x  ⟹
    may_be_eval_exp (V, C) e ⟹
    may_be_eval_exp (V, C) (Let x (MkChn) e) |
  let_send_evt :
    {APrim (SendEvt xc xm)} ⊆ V x ⟹
    may_be_eval_exp (V, C) e ⟹
    may_be_eval_exp (V, C)
      (Let x (Prim (SendEvt xc xm)) e) |
  let_recv_evt :
    {APrim (RecvEvt xc)} ⊆ V x ⟹
    may_be_eval_exp (V, C) e ⟹
    may_be_eval_exp (V, C)
      (Let x (Prim (RecvEvt xc)) e) |
  let_pair :
    {APrim (Pair x1 x2)} ⊆ V x ⟹
```

```
      may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Pair x1 x2) e) |
    let_left :
      {APrim (Left xp)} ⊆ V x ⟹
      may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Left xp) e) |
    let_right :
      {APrim (Right xp)} ⊆ V x ⟹
      may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Right xp) e) |
    let_abs :
      {APrim (Abs f' x' e')} ⊆ V f' ⟹
      may_be_eval_exp (V, C) e' ⟹
      {APrim (Abs f' x' e')} ⊆ V x ⟹
      may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Abs f' x' e') e) |
    let_spawn :
      {AUnt} ⊆ V x ⟹
      may_be_eval_exp (V, C) ec ⟹
      may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Spwn ec) e) |
    let_sync  :
      ∀ xsc xm xc .
        (APrim (SendEvt xsc xm)) ∈ V xe ⟶
        AChn xc ∈ V xsc ⟶
        {AUnt} ⊆ V x ∧ V xm ⊆ C xc ⟹
      ∀ xrc xc .
        (APrim (RecvEvt xrc)) ∈ V xe ⟶
        AChn xc ∈ V xrc ⟶
        C xc ⊆ V x ⟹
      may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Syync xe) e) |
    let_fst :
      ∀ x1 x2.
        (APrim (Pair x1 x2)) ∈ V xp ⟶
        V x1 ⊆ V x ⟹
      may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Fst xp) e) |
    let_snd :
    ∀ x1 x2 .
      (APrim (Pair x1 x2) ∈ V xp ⟶
      V x2 ⊆ V x ⟹
    may_be_eval_exp (V, C) e ⟹
    may_be_eval_exp (V, C) (Let x (Snd xp) e) |
  let_case :
    ∀ xl' .
      (APrim (Left xl')) ∈ V xs ⟶
        V xl' ⊆ V xl ∧ V (rslt_var el) ⊆ V x ∧
        may_be_eval_exp (V, C) el ⟹
    ∀ xr' .
```

```
      (APrim (Right xr')) ∈ V xs ⟶
        V xr' ⊆ V xr ∧ V (rslt_var er) ⊆ V x ∧
        may_be_eval_exp (V, C) er ⟹
        may_be_eval_exp (V, C) e ⟹
      may_be_eval_exp (V, C) (Let x (Case xs xl el xr er) e)
    |
  let_app :
    ∀ f' x' e' .
      (APrim (Abs f' x' e') ∈ V f ⟶
      V xa ⊆ V x' ∧
      V (rslt_var e') ⊆ V x ⟹
    may_be_eval_exp (V, C) e ⟹
    may_be_eval_exp (V, C) (Let x (App f xa) e)
```

```
fun abstract :: val ⇒ abstract_value where
  abstract VUnt = AUnt |
  abstract VChn (Chn pi x) = AChn x |
  abstract VClsr p env = APrim p
```

```
inductive
  may_be_eval_val ::
    abstract_env * abstract_env ⇒ val ⇒ bool and
  may_be_eval_env ::
    abstract_env * abstract_env ⇒ val_env ⇒ bool where
  Unt :
    may_be_eval_val (V, C) VUnt |
  Chan :
    may_be_eval_val (V, C) VChn c |
  SendEvt :
    may_be_eval_env (V, C) env ⟹
    may_be_eval_val (V, C) (VClsr (SendEvt _ _) env) |
  RecvEvt :
    may_be_eval_env (V, C) env ⟹
    may_be_eval_val (V, C) (VClsr (RecvEvt _) env) |
  Left :
    may_be_eval_env (V, C) env ⟹
    may_be_eval_val (V, C) (VClsr (Left _) env) |
  Right :
    may_be_eval_env (V, C) env ⟹
    may_be_eval_val (V, C) (VClsr (Right _) env) |
  Abs :
    {(APrim (Abs f x e)} ⊆ V f ⟹
    may_be_eval_exp (V, C) e ⟹
```

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

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


inductive may_be_eval_state ::
  abstract_env * abstract_env ⇒
  state ⇒ bool where
  intro :
    may_be_eval_exp (V, C) e ⟹
    may_be_eval_env (V, C) env ⟹
    may_be_eval_stack (V, C) (V (rslt_var e)) k ⟹
    may_be_eval_state (V, C) (Stt e env k)

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

```
theorem may_be_eval_preserved_under_concur_step : "
  may_be_eval_pool (V, C) trpl ⟹
```

```
  concur_step (trpl, ys) (trpl', ys') ⟹
  may_be_stati_eval_pool (V, C) trpl'"
proof outline
qed

theorem may_be_eval_preserved_under_concur_step_star : "
  may_be_eval_pool (V, C) trpl ⟹
  star concur_step (trpl, ys) (trpl', ys')  ⟹
  may_be_concur_step (V, C) trpl'"
proof outline
qed
```

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

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

theorem exp_always_value_not_bound_sound : "
  env' x = Some v ⟹
  may_be_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 outline
qed
```

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

```
    is_super_exp ec e ⟹
    is_super_exp (Let x (Spwn ec) en) e |
  let_Case_Left :
    is_super_exp el e ⟹
    is_super_exp (Let x (case xs xl el xr er) en) e |
  let_Case_Right :
    is_super_exp er e ⟹
    is_super_exp (Let x (case xs xl el xr er) en) e |
  let_Abs_Body : "
    is_super_exp eb e ⟹
    is_super_exp (Let x (Abs f xp eb) en) e |
  Let :
    is_super_exp en e ⟹
    is_super_exp (Let x b en) e

inductive
  is_super_exp_left :: exp ⇒ exp ⇒ bool where
  Refl :
    is_super_exp_left e0 e0 |
  let_Spawn_Child :
    is_super_exp_left e0 (Let x (Spwn ec) en)⟹
    is_super_exp_left e0 ec |
  let_Case_Left :
    is_super_exp_left e0 (Let x (case xs xl el xr er) en) ⟹
    is_super_exp_left e0 el |
  let_Case_Right :
    is_super_exp_left e0 (Let x (case xs xl el xr er) en) ⟹
    is_super_exp_left e0 er |
  let_Abs_Body :
    is_super_exp_left e0 (Let x (Abs f xp eb) en) ⟹
    is_super_exp_left e0 eb |
  Let :
    is_super_exp_left e0 (Let x b en) ⟹
    is_super_exp_left e0 en

inductive
  is_super_exp_over_prim :: exp ⇒ prim ⇒ bool where
  SendEvt :
    is_super_exp_over_prim e0 (SendEvt xC xM) |
  RecvEvt :
    is_super_exp_over_prim e0 (RecvEvt xC) |
  Pair :
    is_super_exp_over_prim e0 (Pair x1 x2) |
  Left :
    is_super_exp_over_prim e0 (Left x) |
  Right :
    is_super_exp_over_prim e0 (Right x) |
  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 outline
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 outline
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 outline
qed

lemma exp_always_exp_not_reachable_sound : "
  star concur_step
  ([[] ↦ (Stt e0 (λ _ . None) [])], {})
    (trpl', H') ⟹
  trpl' pi' = Some (Stt e' env' k')) ⟹
  is_super_exp e0 e' "
proof outline
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
  may_be_static_send_node_label ::
    abstract_env ⇒ exp ⇒
    var ⇒ node_label ⇒ bool where
  intro:
    {AChn xC} ⊆ V xSC ⟹
```

```
  {APrim (SendEvt xSC xM)} ⊆ V xE ⟹
  is_super_exp e (Let x (Sync xE) e') ⟹
  may_be_static_send_node_label V e xC (NLet x)

inductive
  may_be_static_recv_node_label ::
    abstract_env ⇒ exp ⇒
    var ⇒ node_label ⇒ bool where
  intro:
    {AChn xC} ⊆ V xRC ⟹
    {APrim (RecvEvt xRC)} ⊆ V xE ⟹
    is_super_exp e (Let x (Sync xE) e') ⟹
    may_be_static_recv_node_label V e xC (NLet x)
```