Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 29, 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
  static_eval_exp ::
    abstract_env * abstract_env ⇒
    exp ⇒ bool where
  Rslt :
    static_eval_exp (V, C) (Rslt 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 ⟹
```

```
      static_eval_exp (V, C) e ⟹
      static_eval_exp (V, C) (Let x (Pair x1 x2) e) |
    let_left :
      {APrim (Left xp)} ⊆ V x ⟹
      static_eval_exp (V, C) e ⟹
      static_eval_exp (V, C) (Let x (Left xp) e) |
    let_right :
      {APrim (Right xp)} ⊆ V x ⟹
      static_eval_exp (V, C) e ⟹
      static_eval_exp (V, C) (Let x (Right xp) e) |
    let_abs :
      {APrim (Abs f' x' e')} ⊆ V f' ⟹
      static_eval_exp (V, C) e' ⟹
      {APrim (Abs f' x' e')} ⊆ V x ⟹
      static_eval_exp (V, C) e ⟹
      static_eval_exp (V, C) (Let x (Abs f' x' e') e) |
    let_spawn :
      {AUnt} ⊆ V x ⟹
      static_eval_exp (V, C) ec ⟹
      static_eval_exp (V, C) e ⟹
      static_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 ⟹
      static_eval_exp (V, C) e ⟹
      static_eval_exp (V, C) (Let x (Syync xe) e) |
    let_fst :
      ∀ x1 x2.
        (APrim (Pair x1 x2)) ∈ V xp ⟶
        V x1 ⊆ V x ⟹
      static_eval_exp (V, C) e ⟹
      static_eval_exp (V, C) (Let x (Fst xp) e) |
    let_snd :
    ∀ x1 x2 .
      (APrim (Pair x1 x2) ∈ V xp ⟶
      V x2 ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_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 ∧
        static_eval_exp (V, C) el ⟹
    ∀ xr' .
```

```
      (APrim (Right xr')) ∈ V xs ⟶
        V xr' ⊆ V xr ∧ V (rslt_var er) ⊆ V x ∧
        static_eval_exp (V, C) er ⟹
        static_eval_exp (V, C) e ⟹
      static_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 ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (App f xa) e)
```

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

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

```
locale semantics_sound =
  assumes
    exp_always_not_static_bound_sound :
      env' x = Some v ⟹
      (V, C) static_eval e ⟹
```

```
    ([[] ↦ (Stt e (λ _ . None) [])], H) star concur_step (
  trpl', H') ⟹
    trpl' pi = Some (Stt e' env' k') ⟹
    {|v|} ⊆ V x and

  exp_always_exp_not_static_reachable_sound:
    ([[] ↦ (Stt e0 (λ _ . None) [])], {}) star concur_step
   (trpl', H') ⟹
    trpl' pi' = Some (Stt e' env' k') ⟹
    static_reachable e0 e'
```

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


```

```

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

inductive
  static_reachable_over_prim :: exp ⇒ prim ⇒ bool where
  SendEvt :
    static_reachable_over_prim e0 (SendEvt xC xM) |
  RecvEvt :
```

```
      static_reachable_over_prim e0 (RecvEvt xC) |
    Pair :
      static_reachable_over_prim e0 (Pair x1 x2) |
    Left :
      static_reachable_over_prim e0 (Left x) |
    Right :
      static_reachable_over_prim e0 (Right x) |
    Abs :
      static_reachable_left e0 eb ⟹
      static_reachable_over_prim e0 (Abs fp xp eb)

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

  inductive
    static_reachable_over_stack ::
      exp ⇒ cont list ⇒ bool where
    Empty :
      static_reachable_over_stack e0 [] |
    Nonempty :
      static_reachable_left e0 ek ⟹
      static_reachable_over_env e0 envk ⟹
      static_reachable_over_stack e0 k ⟹
      static_reachable_over_stack e0 ((Ctn xk ek envk) # k)

  inductive
    static_reachable_over_state ::
      exp ⇒ state ⇒ bool where
    intro :
      static_reachable_left e0 e ⟹
      static_reachable_over_env e0 env ⟹
      static_reachable_over_stack e0 k ⟹
      static_reachable_over_state e0 (Ctn e env k)
```

```
lemma static_reachable_trans :
  static_reachable ez ey ⟹
  static_reachable ey ex ⟹
  static_reachable ez ex
proof sketch
qed


lemma static_reachable_over_state_preserved :
  concur_step (trpl, ys) (trpl', ys') ⟹
  ∀ pi st.
    trpl pi = Some st ⟶
    static_reachable_over_state e0 st ⟹
  trpl' pi' = Some st' ⟹
  static_reachable_over_state e0 st'
proof sketch
qed

```

```

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


```

```
  interpretation semantics_sound
  proof sketch
  qed

```

6 Static Communication

```
locale communication_sound =
  fixes
    static_one_shot :: abstract_env ⇒ exp ⇒ var ⇒ bool and
    static_fan_out :: abstract_env ⇒ exp ⇒ var ⇒ bool and
    static_fan_in :: abstract_env ⇒ exp ⇒ var ⇒ bool and
    static_one_to_one :: abstract_env ⇒ exp ⇒ var ⇒ bool

  assumes
    static_one_shot_sound:
      static_one_shot V e xC ⟹
      static_eval (V, C) e ⟹
      star concur_step ([[] ↦ (Stt e (λ _ . None) [])], {})
   (trpl', H') ⟹
```

```
    one_shot trpl' (Ch pi xC) and

  static_fan_out_sound:
    static_fan_out V e xC ⟹
    (V, C) static_eval e ⟹
    star concur_step ([[] ↦ (Stt e (λ _ . None) [])], {}) (trpl', H') ⟹
    fan_out trpl' (Ch pi xC) and

  static_fan_in_sound:
    static_fan_in V e xC ⟹
    (V, C) static_eval e ⟹
    star concur_step ([[] ↦ (Stt e (λ _ . None) [])], {}) (trpl', H') ⟹
    fan_in trpl' (Ch pi xC) and

  static_one_to_one_sound: "
    static_one_to_one V e xC ⟹
    (V, C) static_eval e ⟹
    star concur_step ([[] ↦ (Stt e (λ _ . None) [])], {}) (trpl', H') ⟹
    one_to_one trpl' (Ch pi xC)"

```

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

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

```
    {APrim (RecvEvt xRC)} ⊆ V xE ⟹
    static_reachable e (Let x (Sync xE) e') ⟹
    static_static_recv_node_label V e xC (NLet x)


```

7 Static Communication part A

```

datatype edge_label = ENxt | ESpwn | ECall | ERtn

type_synonym flow_label = node_label * edge_label * node_label

type_synonym flow_set = flow_label set

type_synonym step_label = node_label * edge_label

type_synonym abstract_path = step_label list

```

```
inductive
  static_traversable ::
    abstract_env ⇒
    (node_label * edge_label * node_label) set ⇒
    exp ⇒ bool  where
  Rslt:
    static_traversable V F (Rslt x) |
  let_Unit:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e  ⟹
    static_traversable V F (Let x Unt e) |
  let_Chan:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x MkChn e) |
  let_SendEvt:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x = (SendEvt xc xm) e) |
  let_RecvEvt:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x = (RecvEvt xc) in e) |
  let_Pair:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Pair x1 x2) e) |
  let_Left:
```

```
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Lft xp) e) |
  let_Right:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Rht xp) e)" |
  let_Abs:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F eb ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Abs f xp eb) e) |
  let_Spawn:
    {
      (NLet x, ENxt, top_node_label e),
      (NLet x, ESpwn, top_node_label ec)}
      ⊆ F ⟹
    static_traversable V F ec ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Spwn ec)  e) |
  let_Sync:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Sync xSE) e) |
  let_Fst:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Fst xp) e) |
  let_Snd:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Snd xp) e) |
  let_Case:
    {
      (NLet x, ECall, top_node_label el),
      (NLet x, ECall, top_node_label er),
      (NRslt (rslt_var el), ERtn, top_node_label e),
      (NRslt (rslt_var er), ERtn, top_node_label e)}
      ⊆ F ⟹
    static_traversable V F el ⟹
    static_traversable V F er ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Case xs xl el xr er) e)
  |
  let_App:
    ∀ fp xp eb .
      APrim (Abs fp xp eb) ∈ V f ⟶
      {
        (NLet x, ECall, top_node_label eb),
        (NRslt (rslt_var eb), ERtn, top_node_label e)}
```

```
      ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (App f xa) e)


```

```
inductive static_traceable ::
  abstract_env ⇒
    flow_set ⇒ node_label ⇒
    (node_label ⇒ bool) ⇒
    abstract_path ⇒ bool where
Empty:
  isEnd start ⟹
  static_traceable V F start isEnd [] |
Edge:
  isEnd end ⟹
  {(start, edge, end)} ⊆ F ⟹
  static_traceable V F start isEnd [(start, edge)] |
Step:
  static_traceable V F middle isEnd ((middle, edge') # path) ⟹
  {(start, edge, middle)} ⊆ F ⟹
  static_traceable V F start isEnd ((start, edge) # (middle, edge') # path)

```

```
inductive static_inclusive ::
  abstract_path ⇒ abstract_path ⇒ bool where
Prefix1:
  prefix pi1 pi2 ⟹
  static_inclusive pi1 pi2 |
Prefix2:
  prefix pi2 pi1 ⟹
  static_inclusive pi1 pi2 |
Spawn1:
  static_inclusive (pi @ (NLet x, ESpwn) # pi1) (pi @ (NLet x, ENxt) # pi2) |
Spawn2:
  static_inclusive (pi @ (NLet x, ENxt) # pi1) (pi @ (NLet x, ESpwn) # pi2)

```

```
inductive
  singular ::
    abstract_path ⇒
    abstract_path ⇒ bool where
equal:
  pi1 = pi2 ⟹
  singular pi1 pi2 |
exclusive:
```

```
    /(static_inclusive pi1 pi2) ⟹
    singular pi1 pi2

inductive
  noncompetitive ::
    abstract_path ⇒ abstract_path ⇒ bool where
  ordered:
    ordered pi1 pi2 ⟹
    noncompetitive pi1 pi2 |
  exclusive:
    /(not_inclusive pi1 pi2) ⟹
    noncompetitive pi1 pi2

```

```
inductive
  static_one_shot ::
    abstract_env ⇒ exp ⇒
    var ⇒ bool where
  Sync:
    every_two
      (static_traceable V F (top_node_label e)
        (static_static_send_node_label V e xC))
      singular ⟹
    static_traversable V F e ⟹
    static_one_shot V e xC

  inductive
    static_one_to_one ::
      abstract_env ⇒ exp ⇒
      var ⇒ bool where
    Sync:
      every_two
        (static_traceable V F
          (top_node_label e) (
 static_static_send_node_label V e xC))
        noncompetitive ⟹
      every_two
        (static_traceable V F
          (top_node_label e) (
 static_static_recv_node_label V e xC))
        noncompetitive ⟹
      static_traversable V F e ⟹
      static_one_to_one V e xC

inductive
  static_fan_out ::
    abstract_env ⇒ exp ⇒
    var ⇒ bool where
  Sync:
    every_two
```

```
      (static_traceable V F
        (top_node_label e) (static_static_send_node_label V e xC))
      noncompetitive ⟹
    static_traversable V F e ⟹
    static_fan_out V e xC

inductive
  static_fan_in ::
    abstract_env ⇒ exp ⇒
    var ⇒ bool where
  Sync:
    every_two (static_traceable V F (top_node_label e) (static_static_recv_node_label V e xC)) noncompetitive ⟹
    static_traversable V F e ⟹
    static_fan_in V e xC
```

```
locale communication_sound_A =
  Static_Communication.communication_sound static_one_shot static_fan_out static_fan_in static_one_to_one
```

```
inductive paths_correspond ::
  control_path ⇒ abstract_path ⇒ bool where
  Empty:
    paths_correspond [] [] |
  Next:
    paths_correspond pi path ⟹
    paths_correspond (pi @ [LNext x]) (path @ [(NLet x, ENxt)]) |
  Spawn:
    paths_correspond pi path ⟹
    paths_correspond (pi @ [LSpawn x]) (path @ [(NLet x, ESpwn)]) |
  Call:
    paths_correspond pi path ⟹
    paths_correspond (pi @ [LCall x]) (path @ [(NLet x, ECall)])  |
  Rtn:
    paths_correspond pi path ⟹
    paths_correspond (pi @ [LRtn x]) (path @ [(NRslt x, ERtn)])
```

```
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"


```

```
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 (VChn c) |
  SendEvt:
    static_traversable_env V F env ⟹
    static_traversable_val V F (VClsr (SendEvt _ _) env) |
  RecvEvt:
    static_traversable_env V F env ⟹
    static_traversable_val V F (VClsr (RecvEvt _) 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 not_static_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
```

```
interpretation communication_sound_A
  proof -
```

8 Static Communication part B

```
datatype edge_label = ENxt | ESpwn | ESend var | ECall |
    ERtn var

type_synonym flow_label = (node_label * edge_label *
    node_label)

type_synonym flow_set = flow_label set

type_synonym step_label = (node_label * edge_label)

type_synonym abstract_path = step_label list
```

```
inductive static_traversable :: abstract_env ⇒ flow_set
  ⇒ (var ⇒ node_label ⇒ bool) ⇒ exp ⇒ bool  where
  result:
      static_traversable V F static_recv_site (Rslt x) |
  let_Unit:
      {(NLet x , ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x Unt e
  ) |
  let_Chan:
```

```
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x MkChn e) |
let_SendEvt:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Prim (SendEvt xc xm)) e)" |
let_RecvEvt:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Prim (RecvEvt xc)) e) |
let_Pair:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Prim (Pair x1 x2)) e) |
let_Left:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Prim (Lft xp)) e) |
let_Right:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Prim (Rght xp)) e) |
let_Abs:
  {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
  static_traversable V F static_recv_site eb ⟹
  static_traversable V F static_recv_site e ⟹
  static_traversable V F static_recv_site (Let x (Prim (Abs f xp eb)) e) |
let_Spawn:
  {(NLet x, ENxt, top_node_label e),
    (NLet x, ESpawn, top_node_label ec)} ⊆ F ⟹
  static_traversable V F static_recv_site ec ⟹
  static_traversable V F static_recv_site e ⟹
  static_traversable V F static_recv_site (Let x (Spwn ec) e) |
let_Sync:
  {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
  (∀ xSC xM xC y .
    {^SendEvt xSC xM} ⊆ V xSE ⟶
    {^Chan xC} ⊆ V xSC ⟶
    static_recv_site xC (NLet y) ⟶
    {(NLet x, ESend xSE, NLet y)} ⊆ F) ⟹
  static_traversable V F static_recv_site e ⟹
```

```
    static_traversable V F static_recv_site (Let x (Sync xSE) e) |
  let_Fst:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Fst xp) e) |
  let_Snd:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Snd xp) e) |
  let_Case:
      {
        (NLet x, ECall, top_node_label el),
        (NLet x, ECall, top_node_label er),
        (NRslt (rslt_var el), ERtn x, top_node_label e),
        (NRslt (rslt_var er), ERtn x, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site el ⟹
      static_traversable V F static_recv_site er ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Case xs xl el xr er) e) |
  let_App:
      (∀ f' xp eb . ^Abs f' xp eb ∈ V f ⟶
        {(NLet x, ECall, top_node_label eb),
          (NRslt (rslt_var eb), ERtn x, top_node_label e)} ⊆ F) ⟹
      static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (App f xa) e)


```

```
inductive
  static_built_on_chan :: "abstract_env ⇒ node_map ⇒ var ⇒ var ⇒ bool"
where
  Chan:
      AChn xc ∈ V x ⟹
      static_built_on_chan V Ln xc x |
  SendEvt:
      APrim (SendEvt xsc xm) ∈ V x ⟹
      static_built_on_chan V Ln xc xsc ∨ static_built_on_chan V Ln xc xm ⟹
      static_built_on_chan V Ln xc x |
  RecvEvt:
    APrim (RecvEvt xrc) ∈ V x ⟹
    static_built_on_chan V Ln xc xrc ⟹
```

```
  static_built_on_chan V Ln xc x |
Pair:
    APrim (Pair x1 x2) ∈ V x ⟹
    static_built_on_chan V Ln xc x1 ∨ static_built_on_chan
  V Ln xc x2 ⟹
    static_built_on_chan V Ln xc x |
Left:
    APrim (Left xa) ∈ V x ⟹
    static_built_on_chan V Ln xc xa ⟹
  static_built_on_chan V Ln xc x |
Right:
    APrim (Right xa) ∈ V x ⟹
    static_built_on_chan V Ln xc xa ⟹
    static_built_on_chan V Ln xc x |
Abs:
  APrim (Abs f xp eb) ∈ V x ⟹
  /Set.is_empty (Ln (nodeLabel eb) - {xp}) ⟹
  static_built_on_chan V Ln xc x

```

```
fun chan_set ::
  abstract_env ⇒ node_map ⇒ var ⇒ var ⇒ var set" where
  chan_set V Ln xc x = (if (static_built_on_chan V Ln xc x
 ) then {x} else {})

```

```
inductive static_live_chan ::
  abstract_env ⇒ node_map ⇒ node_map ⇒ var ⇒ exp ⇒ bool
  where
  Result:
    chan_set V Ln xc y = Ln (NRslt y) ⟹
    static_live_chan V Ln Lx xc (Rslt y) |
Let_Unit:
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    Lx (NLet x) = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x Unt e) |
Let_Chan:
  static_live_chan V Ln Lx xc e ⟹
  Ln (top_node_label e) = Lx (NLet x) ⟹
  (Lx (NLet x) - {x}) = Ln (NLet x) ⟹
  static_live_chan V Ln Lx xc (Let x MkChn e) |
Let_SendEvt:
  static_live_chan V Ln Lx xc e ⟹
  Ln (top_node_label e) = Lx (NLet x) ⟹
  (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xsc ∪
    chan_set V Ln xc xm = Ln (NLet x) ⟹
  static_live_chan V Ln Lx xc (Let x (Prim (SendEvt xsc xm
 )) e) |
Let_RecvEvt:
```

```
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xrc = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x (Prim (RecvEvt xrc)) e) |
  Let_Pair:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪  chan_set V Ln xc x1 ∪
        chan_set V Ln xc x2 = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Pair x1 x2)) e) |
  Let_Left:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Lft xa)) e) |
  Let_Right:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Rght xa)) e) |
  Let_Abs:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      static_live_chan V Ln Lx xc eb ⟹
      (Lx (NLet x) - {x}) ∪
        (Ln (top_node_label eb) - {xp}) = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Abs f xp eb)) e) |
  Let_Spawn:
      static_live_chan V Ln Lx xc e ⟹
      static_live_chan V Ln Lx xc ec ⟹
      Ln (top_node_label e) ∪ Ln (top_node_label ec) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Spwn ec) e) |
  Let_Sync:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xe = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Sync xe) e) |
  Let_Fst:
      static_live_chan V Ln Lx xc e ⟹
```

```
    Ln (top_node_label e) = Lx (NLet x) ⟹
    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x (Fst xa) e) |
Let_Snd:
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x (Snd xa) e) |
Let_Case:
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    static_live_chan V Ln Lx xc el ⟹
    static_live_chan V Ln Lx xc er ⟹
    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xs ∪
    (Ln (top_node_label el) - {xl}) ∪
    (Ln (top_node_label er) - {xr}) = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x (Case xs xl el xr er) e) |
Let_App:
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    (Lx (NLet x) - {x}) ∪
    chan_set V Ln xc f ∪
    chan_set V Ln xc xa = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x (App f xa) e)
```

```
inductive static_traceable ::
  flow_set ⇒ node_label
  ⇒ abstract_path ⇒ bool" where
  Empty:
    static_traceable F end [] |
  Edge:
    (start, edge, end) ∈ F ⟹
    static_traceable F end [(start, edge)] |
  Step:
    static_traceable F end ((middle, edge') # post) ⟹
    (start, edge, middle) ∈ F ⟹
    path = [(start, edge), (middle, edge')] @ post ⟹
    static_traceable F end path
```

```
inductive static_live_traversable :: "flow_set ⇒ node_map
  ⇒ node_map ⇒ flow_label ⇒ bool"  where
  Next: "
    (l, ENxt, l') ∈ F ⟹
    /Set.is_empty (Lx l) ⟹
    /Set.is_empty (Ln l') ⟹
    static_live_traversable F Ln Lx (l, ENxt, l')
  " |
  Spawn: "
    (l, ESpwn, l') ∈ F ⟹
    /Set.is_empty (Lx l) ⟹
    /Set.is_empty (Ln l') ⟹
    static_live_traversable F Ln Lx (l, ESpwn, l')
  " |
  Call_Live_Outer: "
    (l, ECall, l') ∈ F ⟹
    /Set.is_empty (Lx l) ⟹
    static_live_traversable F Ln Lx (l, ECall, l')
  " |
  Call_Live_Inner: "
    (l, ECall, l') ∈ F ⟹
    /Set.is_empty (Ln l') ⟹
    static_live_traversable F Ln Lx (l, ECall, l')
  " |
  Return: "
    (l, ERtn x, l') ∈ F ⟹
    /Set.is_empty (Ln l') ⟹
    static_live_traversable F Ln Lx (l, ERtn x, l')
  " |
  Send: "
    ((NLet xSend), ESend xE, (NLet xRecv)) ∈ F ⟹
    {xE} ⊆ (Ln (NLet xSend)) ⟹
    static_live_traversable F Ln Lx ((NLet xSend), ESend xE,
     (NLet xRecv))
    "
```

```
inductive static_live_traceable :: "abstract_env ⇒ flow_set
   ⇒ node_map ⇒ node_map ⇒ node_label ⇒ (node_label ⇒
   bool) ⇒ abstract_path ⇒ bool" where
  Empty:
    isEnd start ⟹
    static_live_traceable V F Ln Lx start isEnd [] |
  Edge:
    isEnd end ⟹
    static_live_traversable F Ln Lx (start, edge, end) ⟹
    static_live_traceable V F Ln Lx start isEnd [(start,
```

```
  edge)] |
Step:
  static_live_traceable V F Ln Lx middle isEnd ((middle, edge') # path) ⟹
  static_live_traversable F Ln Lx (start, edge, middle) ⟹
  static_live_traceable V F Ln Lx start isEnd ((start, edge) # (middle, edge') # path) |
Pre_Return:
  static_live_traceable V F Ln Lx (NRslt y) isEnd ((NRslt y, ERtn x) # post) ⟹
  static_traceable  F (NRslt y) pre ⟹
  /static_balanced (pre @ [(NRslt y, ERtn x)]) ⟹
  /Set.is_empty (Lx (NLet x)) ⟹
  path = pre @ (NRslt y, ERtn x) # post ⟹
  static_live_traceable V F Ln Lx start isEnd path
```

```
inductive static_inclusive ::
  abstract_path ⇒ abstract_path ⇒ bool where
Prefix1:
  prefix pi1 pi2 ⟹
  pi1 static_inclusive pi2 |
Prefix2:
  prefix pi2 pi1 ⟹
  pi1 static_inclusive pi2 |
Spawn1:
  static_inclusive (pi @ (NLet x, ESpwn) # pi1) (pi @ (NLet x, ENxt) # pi2) |
Spawn2:
  static_inclusive (pi @ (NLet x, ENxt) # pi1 static_inclusive) (pi @ (NLet x, ESpwn) # pi2) |
Send1:
  static_inclusive (pi @ (NLet x, ESend xE) # pi1) (pi @ (NLet x, ENxt) # pi2) |
Send2:
  static_inclusive (pi @ (NLet x, ENxt) # pi1) (pi @ (NLet x, ESend xE) # pi2)
```

```
inductive singular ::
  abstract_path ⇒ abstract_path ⇒ bool where
  equal:
    pi1 = pi2 ⟹
```

```
      singular pi1 pi2 |
    exclusive:
      /(pi1 static_inclusive pi2) ⟹
      singular pi1 pi2

    inductive noncompetitive ::
      abstract_path ⇒ abstract_path ⇒ bool" where
  ordered:
    ordered pi1 pi2 ⟹
    noncompetitive pi1 pi2 |
  exclusive:
    /(pi1 static_inclusive pi2) ⟹
    noncompetitive pi1 pi2

inductive static_one_shot :: abstract_env ⇒ exp ⇒ var ⇒
    bool where
  Sync:
    every_two (static_live_traceable V F Ln Lx (NLet xC) (
    static_send_node_label V e xC)) singular ⟹
    static_live_chan V Ln Lx xC e ⟹
    static_traversable V F (static_recv_node_label V e) e ⟹
    static_one_shot V e xC

inductive static_one_to_one :: abstract_env ⇒ exp ⇒ var ⇒
    bool where
  Sync:
    every_two (static_live_traceable V F Ln Lx (NLet xC) (
    static_send_node_label V e xC)) noncompetitive ⟹
    every_two (static_live_traceable V F Ln Lx (NLet xC) (
    static_recv_node_label V e xC)) noncompetitive ⟹
    static_live_chan V Ln Lx xC e ⟹
    static_traversable V F (static_recv_node_label V e) e ⟹
    static_one_to_one V e xC

inductive static_fan_out :: abstract_env ⇒ exp ⇒ var ⇒
    bool where
  Sync:
    every_two (static_live_traceable V F Ln Lx (NLet xC) (
    static_send_node_label V e xC)) noncompetitive ⟹
    static_live_chan V Ln Lx xC e ⟹
    static_traversable V F (static_recv_node_label V e) e ⟹
    static_fan_out V e xC

inductive static_fan_in :: abstract_env ⇒ exp ⇒ var ⇒ bool
     where
  Sync:
    every_two (static_live_traceable V F Ln Lx (NLet xC) (
    static_recv_node_label V e xC)) noncompetitive ⟹
    static_live_chan V Ln Lx xC e ⟹
    static_traversable V F (static_recv_node_label V e) e ⟹
```

```
    static_fan_in V e xC
```

```
locale communication_sound_B =
  Static_Communication.communication_sound static_one_shot
    static_fan_out static_fan_in static_one_to_one
```

```
interpretation communication_sound_B
proof sketch
qed
```

```
  let lp = fun lp x =>
    let z1 = case x of
      L y => let z2 = lp y in z2 |
      R () => let z3 = () in z3
      in ()
    in

  let mksr = fun _ x =>
    let ch1 = mkChan ()  in
    let z4 = (lp (L (L (R ())))) in
    let srv = fun srv x  =>
      let p = sync (recv_evt ch1) in
      let v1 = fst p in
      let ch2 = snd p in
      let z5 = sync (send_evt ch2 x) in
      let z6 = srv v1 in ()
      in
    let z7 = spawn (
      let z8 = srv (R ()) in ()) in
```

```
  ch1 in

let rqst = fun _ pair =>
  let ch3 = fst pair in
  let v2 = snd pair in
  let ch4 = chan () in
  let z9 = sync (send_evt ch3 (v2, ch4)) in
  let v3 = sync (recv_evt ch4) in
  v3 in

let srvr = mksr () in
let z10 = spawn (
  let z11 = rqst (srvr, R ()) in ())
  in
let z12 = rqst (srvr, L (R ())) in
()
```