A Mechanized Theory of Communication Analysis in CML

March 4, 2019

Concurrent ML

- extension of Standard ML
- concurrency and synchronization
- synchronized communication over channels: send event, receive event
- composition of events: choose event, wrap event ...

Concurrent ML

```
type thread_id
val spawn : (unit -> unit) -> thread_id

type 'a chan
val channel : unit -> 'a chan

type 'a event
val sync: 'a event -> 'a

val recvEvt: 'a chan -> 'a event
val sendEvt: 'a channel * 'a -> unit event

val send: 'a chan * 'a -> unit
fun send (ch, v) = sync (sendEvt (ch, v))

val recv: 'a chan -> 'a
fun recv ch = sync (recvEvt ch)
```

Concurrent ML

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

  fun make () =
  let
    val reqCh = channel ()
    fun loop state =
    let
      val (v, replCh) = recv reqCh
      val () = send (replCh, state)
    in
      loop v
    end
    val () = spawn (fn () => loop 0)
  in
    S reqCh
  end

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

signature SERV =
sig
  type serv
  val make : unit -> serv
  val call : serv * int -> int
end
```

Isabelle/HOL

- interactive theorem proving assistant; proof assistant
- unification and rewriting
- simply typed terms
- propositions as boolean typed terms
- higher order terms
- computable functions
- inductive data
- inductive reasoning
- tactics and composition

Isabelle/HOL

```
⊢ P1 ∨ P2 ⟶ Q
proof
  assume P1 ∨ P2:
    case P1:
      have ⊢ P1 ⟶ Q by A
      have ⊢ Q by modus ponens
    case P2:
      have ⊢ P2 ⟶ Q by B
      have ⊢ Q by modus ponens
    have P1 ⊢ Q, P2 ⊢ Q
    have ⊢ Q by disjunction elimination
  have P1 ∨ P2 ⊢ Q
  have ⊢ P1 ∨ P2 ⟶ Q
    by implication introduction
qed
```

```
⊢ P1 ∨ P2 ⟶ Q
apply (rule impI)
  P1 ∨ P2 ⊢ Q
apply (erule disjE)
  P1 ⊢ Q
∗ P2 ⊢ Q
apply (insert A)
  P1, P1 ⟶ Q ⊢ Q
∗ P2 ⊢ Q
apply (erule mp)
  P1 ⊢ P1
∗ P2 ⊢ Q
apply assumption
  P2 ⊢ Q
apply (insert B)
  P2, P2 ⟶ Q ⊢ Q
apply (erule mp)
  P2 ⊢ P2
apply assumption
done
```

Isabelle/HOL

```
datatype nat = Z | S nat

predicate lte: nat -> nat -> bool where
  eq: n .
  ⊢ lte n n
* lt: n1 n2 .
    lte n1 n2
  ⊢ lte n1 (S n2)
```

```
datatype 'a list =
  Nil
| Cons 'a ('a list)

predicate sorted:
  ('a -> 'a -> bool) -> 'a list -> bool
where
  nil: r .
  ⊢ sorted r Nil
* uni: r x .
  ⊢ sorted r (Cons x Nil)
* cons: r x y ys .
    r x y,
    sorted r (Cons y ys)
  ⊢ sorted r (Cons x (Cons y ys))
```

Isabelle/HOL

```
⊢ sorted lte (Cons (Z) (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil))))
apply (rule cons)
  ⊢ lte Z (S Z)
* ⊢ sorted lte (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil)))
apply (rule lt)
  ⊢ lte Z Z
* ⊢ sorted lte (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil)))
apply (rule eq)
  ⊢ sorted lte (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil)))
apply (rule cons)
  ⊢ lte (S Z) (S Z)
* ⊢ sorted lte (Cons (S Z) (Cons (S (S (S Z))) Nil))
apply (rule eq)
  ⊢ sorted lte (Cons (S Z) (Cons (S (S (S Z))) Nil))
apply (rule cons)
  ⊢ lte (S Z) (S (S (S Z)))
* ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule lt)
  ⊢ lte (S Z) (S (S Z))
* ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule lt)
  ⊢ lte (S Z) (S Z)
* ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule eq)
  ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule uni)
done
```

Analysis

- communication classification: one-shot, one-to-many, many-to-one, many-to-many
- control flow analysis
- channel liveness
- algorithm vs constraints
- structural recursion vs fixpoint accumulation
- performance improvements
- safety

Synchronization

- uniprocessor; dispatch scheduling
- multiprocessor; mutex and compare-and-swap
- synchronization state
- sender and receiver thread containers
- message containers

Syntax

```
datatype name = Nm string

datatype term =
  Bind name complex term
| Rslt name

and complex =
  Unt
| MkChn
| Atom atom
| Spwn term
| Sync name
| Fst name
| Snd name
| Case name name term name term
| App name name

and atom =
  SendEvt name name
| RecvEvt name
| Pair name name
| Lft name
| Rht name
| Fun name name term
```

Dynamic Semantics

```
datatype dynamic_step =
  DSeq name
| DSpwn name
| DCll name
| DRtn name

type dynamic_path =
  dynamic_step list

datatype chan =
  Chan dynamic_path name

datatype dynamic_value =
  VUnt
| VChn chan
| VAtm atom (name -> dynamic_value option)

type environment =
  name -> dynamic_value option
```

Dynamic Semantics

```
predicate seqEval: complex -> environment -> dynamic_value -> bool where

  unit: env .
  ├─ seqEval Unt env VUnt

* atom: a env .
  ├─ seqEval (Atom a) env (VAtm a env)

* first: env n_p n_1 n_2 env_p v .
    env n_p = Some (VAtm (Pair n_1 n_2) env_p),
    env_p n_1 = Some v
  ├─ seqEval (Fst n_p) env v

* second: env n_p n_1 n_2 env_p v .
    env n_p = Some (VAtm (Pair n_1 n_2) env_p),
    env_p n_2 = Some v
  ├─ seqEval (Snd n_p) env v
```

Dynamic Semantics

```
predicate callEval: complex -> env -> term -> env -> bool where

  distincLeft: env n_s n_c env_s v n_l t_l n_r t_r .
    env n_s = Some (VAtm (Lft n_c) env_s),
    env_s n_c = Some v
  ⊢ callEval (Case n_s n_l t_l n_r t_r) env t_l (env(n_l -> v))

* distincRight: env n_s n_c env_s v n_l t_l n_r t_r .
    env n_s = Some (VAtm (Rht n_c) env_s),
    env_s n_c = Some v
  ⊢ callEval (Case n_s n_l t_l n_r t_r) env t_r (env(n_r -> v))

* application: env n_f n_f' n_p t_b env_f n_a v .
    env n_f = Some (VAtm (Fun n_f' n_p t_b) env_f),
    env n_a = Some v
  ⊢ callEval
    (App n_f n_a) env t_b
    (env_f(
      n_f' -> (VAtm (Fun n_f' n_p t_b) env_f),
      n_p -> v
    ))
```

Dynamic Semantics

```
datatype contin = Ctn name tm env

type stack = contin list

datatype state =
  Stt program env stack

type pool =
  dynamic_path -> state option

predicate leaf: pool -> dynamic_path -> bool where
  intro: pool path stt .
    pool path = Some stt,
    (∄ path' stt' .
      pool path' = Some stt',
      strictPrefix path path'
    )
  ⊢ leaf pool path

type corresp = dynamic_path * chan * dynamic_path

type communication = corresp set
```

Dynamic Semantics

```
predicate dynamicEval:
  pool -> communication -> pool -> communication -> bool
where

  return: pool path n env n_k t_k env_k stack' v comm .
    leaf pool path,
    pool path = Some (Stt (Rslt n) env ((Ctn n_k t_k env_k) # stack')),
    env n = Some v
  ├ dynamicEval
    pool comm
    (pool(
      path @ [DRtn n] ->
        (Stt t_k env_k(n_k -> v) stack')
    ))
    comm

* seq: pool path n c t' env stack v .
    leaf pool path,
    pool path = Some (Stt (Bind n c t') env stack),
    seqEval c env v
  ├ dynamicEval
    pool comm
    (pool(
      path @ [DSeq n] -> (Stt t' (env(n -> v)) stack)
    ))
    comm
```

Dynamic Semantics

```
* call: pool path n c t' env stack t_c env_c comm .
    leaf pool path,
    pool path = Some (Stt (Bind n c t') env stack),
    callEval c env t_c env_c
  ├ dynamicEval
    pool comm
    (pool(
      path @ [DCll n] -> (Stt t_c env_c ((Ctn n t' env) # stack))
    )) comm

* makeChan: pool path n t' env stack .
    leaf pool path,
    pool path = Some (Stt (Bind n MkChn t') env stack)
  ├ dynamicEval pool comm
    (pool(
      path @ [DSeq n] ->
        (Stt t' (env(n -> (VChn (Chan path n)))) stack)
    )) comm

* spawn: pool path n t_c t' env stack comm .
    leaf pool path,
    pool path = Some (Stt (Bind n (Spwn t_c) t') env stack)
  ├ dynamicEval pool comm
    (pool(
      path @ [DSeq n] -> (Stt t' (env(n -> VUnt)) stack),
      path @ [DSpwn n] -> (Stt t_c env [])
    )) comm
```

Dynamic Semantics

```
* sync: pool path_s n_s n_se t_s env_s stack_s n_sc n_m
  env_se path_r n_r n_re t_r env_r stack_r n_rc env_re chan comm .
    leaf pool path_s,
    pool path_s = Some
      (Stt (Bind n_s (Sync n_se) t_s) env_s stack_s),
    env_s n_se = Some
      (VAtm (SendEvt n_sc n_m) env_se),
    leaf pool path_r,
    pool path_r = Some
      (Stt (Bind n_r (Sync n_re) t_r) env_r stack_r),
    env_r n_re = Some
      (VAtm (RecvEvt n_rc) env_re),
    env_se n_sc = Some (VChn chan),
    env_re n_rc = Some (VChn chan),
    env_se n_m = Some v_m
  ⊢ dynamicEval
    pool comm
    (pool(
      path_s @ [DSeq n_s] -> (Stt t_s (env_s(n_s -> VUnt)) stack_s),
      path_r @ [DSeq n_r] -> (Stt t_r (env_r(n_r -> v_m)) stack_r)
    ))
    (comm ∪ {(path_s, chan, path_r)})
```

Dynamic Communication

```
predicate isSendPath: pool -> chan -> dynamic_path -> bool where
  intro: pool path n n_e t' env stack n_sc n_m env_e chan .
    pool path = Some (Stt (Bind n (Sync n_e) t') env stack),
    env n_e = Some (VAtm (SendEvt n_sc n_m) env_e),
    env_e n_sc = Some (VChn chan)
  ⊢ isSendPath pool chan path

predicate isRecvPath: pool -> chan -> dynamic_path -> bool where
  intro: pool path n n_e t' env stack n_rc env_e chan .
    pool path = Some (Stt (Bind n (Sync n_e) t') env stack),
    env n_e = Some (VAtm (RecvEvt n_rc) env_e),
    env_e n_rc = Some (VChn chan)
  ⊢ isRecvPath pool chan path

predicate forEveryTwo: ('a -> bool) -> ('a -> 'a -> bool) -> bool where
  intro: p r .
    ∀ path1 path2 .
      p path1 ∧ p path2 ⟶ r path1 path2
  ⊢ forEveryTwo p r

predicate ordered: 'a list -> 'a list -> bool where
  first: path1 path2 .
    prefix path1 path2
  ⊢ ordered path1 path2
* second: path2 path1 .
    prefix path2 path1
  ⊢ ordered path1 path2
```

Dynamic Communication

```
predicate oneToMany: tm -> chan -> bool where
  intro: t0 chan .
    star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
    forEveryTwo (isSendPath pool chan) ordered
  ├ oneToMany pool chan

predicate manyToOne: tm -> chan -> bool where
  intro: t0 chan .
    star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
    forEveryTwo (isRecvPath pool chan) ordered
  ├ manyToOne t0 chan

predicate oneToOne: tm -> chan -> bool where
  intro: t0 chan .
    star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
    forEveryTwo (isSendPath pool chan) ordered,
    forEveryTwo (isRecvPath pool chan) ordered
  ├ oneToOne t0 chan
```

Dynamic Communication

```
predicate oneShot: tm -> chan -> bool where
  intro: t0 chan .
    star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
    forEveryTwo (isSendPath pool chan) (op =)
  ⊢ oneShot t0 chan

predicate oneSync: tm -> chan -> bool where
  intro: t0 chan .
    star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
    forEveryTwo (isSendPath pool chan) (op =),
    forEveryTwo (isRecvPath pool chan) ordered
  ⊢ oneSync t0  chan
```

Static Semantics

```
datatype static_value =
  SUnt
| SChn name
| SAtm atom

type static_value_map =
  name -> static_value set

fun resultName: term -> name where
  n .
  ├─ resultName (Rslt n) = n
* n c t' .
  ├─ resultName (Bind n c t') = (resultName t)
```

Static Semantics

```
predicate staticEval:
  static_value_map -> static_value_map -> term -> bool
where

  result: staticEnv staticComm n .
  ⊢ staticEval staticEnv staticComm (Rslt n)

* unit: staticEnv n staticComm t' .
    SUnt ∈ staticEnv n,
    staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n Unt t')

* makeChan: n staticEnv staticComm t' .
    (SChn n) ∈ staticEnv n,
    staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n MkChn t')

* sendEvt: n_c n_m staticEnv n staticComm t' .
    (SAtm (SendEvt n_c n_m)) ∈ staticEnv n,
    staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (SendEvt n_c n_m)) t')

* recvEvt: n_c staticEnv n staticComm t' .
    (SAtm (RecvEvt n_c)) ∈ staticEnv n,
    staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (RecvEvt n_c)) t')
```

Static Semantics

* pair: n_1 n_2 staticEnv n staticComm t' .
  (SAtm (Pair n_1 n_2)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Pair n_1 n_2)) t')

* left: n_a staticEnv n staticComm t' .
  (SAtm(Lft n_a)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Lft n_a)) t')

* right: n_a staticEnv n staticComm t' .
  (SAtm(Rht n_a)) ∈ staticEnv n,
  staticEval staticEnv staticComm t
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Rht n_a)) t')

* function: n_f n_t t_b staticEnv staticComm n t' .
  (SAtm (Fun n_f n_t t_b)) ∈ staticEnv n_f,
  staticEval staticEnv staticComm t_b,
  (SAtm (Fun n_f n_t t_b)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Fun n_f n_t t_b)) t')

* spawn: n_f n_t t_b staticEnv staticComm n t' .
  SUnt ∈ staticEnv n,
  staticEval staticEnv staticComm t_c,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Spwn t_c) t')

Static Semantics

* sync: staticEnv n_e n staticComm t'.

$$\forall\ n_{sc}\ n_m\ n_c\ .\ (\mathsf{SAtm}\ (\mathsf{SendEvt}\ n_{sc}\ n_m)) \in \mathsf{staticEnv}\ n_e \rightarrow \mathsf{SChn}\ n_c \in \mathsf{staticEnv}\ n_{sc} \rightarrow \mathsf{SUnt} \in \mathsf{staticEnv}\ n \wedge \mathsf{staticEnv}\ n_m \subseteq \mathsf{staticComm}\ n_c,$$

$$\forall\ n_{rc}\ n_c\ .\ (\mathsf{SAtm}\ (\mathsf{RecvEvt}\ n_{rc})) \in \mathsf{staticEnv}\ n_e \rightarrow \mathsf{SChn}\ n_c \in \mathsf{staticEnv}\ n_{rc} \rightarrow \mathsf{staticComm}\ n_c \subseteq \mathsf{staticEnv}\ n,\ \mathsf{staticEval}\ \mathsf{staticEnv}\ \mathsf{staticComm}\ t'$$

$$\vdash \mathsf{staticEval}\ \mathsf{staticEnv}\ \mathsf{staticComm}\ (\mathsf{Bind}\ n\ (\mathsf{Sync}\ n_e)\ t')$$

* first: staticEnv n_t n staticComm t' .

$$\forall\ n_1\ n_2\ .\ (\mathsf{SAtm}\ (\mathsf{Pair}\ n_1\ n_2)) \in \mathsf{staticEnv}\ n_t \rightarrow \mathsf{staticEnv}\ n_1 \subseteq \mathsf{staticEnv}\ n,\ \mathsf{staticEval}\ \mathsf{staticEnv}\ \mathsf{staticComm}\ t'$$

$$\vdash \mathsf{staticEval}\ \mathsf{staticEnv}\ \mathsf{staticComm}\ (\mathsf{Bind}\ n\ (\mathsf{Fst}\ n_t)\ t')$$

* second: staticEnv n_t n staticComm t' .

$$\forall\ n_1\ n_2\ .\ (\mathsf{SAtm}\ (\mathsf{Pair}\ n_1\ n_2)) \in \mathsf{staticEnv}\ n_t \rightarrow \mathsf{staticEnv}\ n_2 \subseteq \mathsf{staticEnv}\ n,\ \mathsf{staticEval}\ \mathsf{staticEnv}\ \mathsf{staticComm}\ t'$$

$$\vdash \mathsf{staticEval}\ \mathsf{staticEnv}\ \mathsf{staticComm}\ (\mathsf{Bind}\ n\ (\mathsf{Snd}\ n_t)\ t')$$

Static Semantics

```
* distinction: staticEnv n_s n_l t_l n staticComm n_r t_r t' .
    ∀ n_c .
      (SAtm (Lft n_c)) ∈ staticEnv n_s
    ⟶ staticEnv n_c ⊆ staticEnv n_l,
    staticEnv (resultName t_l) ⊆ staticEnv n,
    staticEval staticEnv staticComm t_l,
    ∀ n_c .
      (SAtm (Rht n_c)) ∈ staticEnv n_s
    ⟶ staticEnv n_c ⊆ staticEnv n_r,
    staticEval staticEnv staticComm t_r,
    staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Case n_s n_l t_l n_r t_r) t')

* application: staticEnv n_f n_a n staticComm t' .
    ∀ n_f' n_t t_b .
      (SAtm (Fun n_f' n_t t_b)) ∈ staticEnv n_f
    ⟶ staticEnv n_a ⊆ staticEnv n_t,
    staticEnv (resultName t_b) ⊆ staticEnv n),
    staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (App n_f n_a) t')
```

Static Semantics

```
bind u1 = unt
bind r1 = rht u1
bind l1 = lft r1
bind l2 = lft l1

bind mksr = fun _ x2 =>
(
  bind k1 = mkChn
  bind srv = fun srv' x3 =>
  (
    bind e1 = recvEvt k1
    bind p1 = sync e1
    bind v1 = fst p1
    bind k2 = snd p1
    bind e2 = sendEvt k2 x3
    bind z5 = sync e2
    bind z6 = app srv' v1
    rslt z6
  )
  bind z7 = spawn
  (
    bind z8 = app srv r1
    rslt z8
  )
  rslt k1
)
```

```
bind rqst = fun _ x4 =>
(
  bind k3 = fst x4
  bind v2 = snd x4
  bind k4 = mkChn
  bind p2 = pair v2 k4
  bind e3 = sendEvt k3 p2
  bind z9 = sync e3
  bind e4 = recvEvt k4
  bind v3 = sync e4
  rslt v3
)

bind srvr = mksr u1
bind z10 = spawn
(
  bind p3 = pair srvr l1
  bind z11 = app rqst p3
  rslt z11
)
bind p4 = pair srvr l2
bind z12 = app rqst p4
rslt z12
```

Static Semantics

```
val staticEnv: name -> static_value set =
(
  u1 -> {unt},
  r1 -> {rht u1},
  l1 -> {lft r1},
  l2 -> {lft l1},
  mksr -> {fun  _ x2 => ...},
  x2 -> {unt},
  k1 -> {chn k1},
  srv -> {fun srv' x3 => ...},
  srv' -> {fun srv' x3 => ...},
  x3 -> {rht u1, lft r1, lft l1},
  e1 -> {recvEvt k1},
  p1 -> {pair v2 k4},
  v1 -> {lft r1, lft l1},
  k2 -> {chn k4},
  e2 -> {sendEvt k2 x3},
  z5 -> {unt},
  z7 -> {unt},
  u5 -> {unt},
  rqst -> {fun _ x4 => ...},
  x4 -> {pair srvr l1, pair srvr l2},
  k3 -> {chn k1},
  v2 -> {lft r1, lft l1},
  k4 -> {chn k4},
  p2 -> {pair v2 k4},
  e3 -> {sendEvt k3 p2},
  z9 -> {unt},
  e4 -> {recvEvt k4},
  v3 -> {rht u1, lft r1, lft r2},
  srvr -> {chn k1},
  z10 -> {unt},
  p3 -> {pair srvr l1},
  z11 -> {rht u1, lft r2},
  p4 -> {pair srvr l2},
  z12 -> {rht u1, lft l1}
)

val staticComm: name -> static_value set =
(
  k1 -> {pair v2 k4},
  k4 -> {rht u1, lft l1, lft l2}
)
```

Static Semantics

```
predicate staticReachable: term -> term -> bool where

  refl: t .
  ├ staticReachable t t

* spawn: t_c t_z n t' .
    staticReachable t_c t_z
  ├ staticReachable (Bind n (Spwn t_c) t') t_z

* distincLeft: t_l t_z n n_s n_l n_r t_r t' .
    staticReachable t_l t_z
  ├ staticReachable (Bind n (Case n_s n_l t_l n_r t_r) t') t_z

* distincRight: t_r t_z n n_s n_l t_l n_r t' .
    staticReachable t_r t_z
  ├ staticReachable (Bind n (Case n_s n_l t_l n_r t_r) t') t_z

* function: t_b t_z n n_f n_t t_b t' .
    staticReachable t_b t_z
  ├ staticReachable (Bind n (Atom (Fun n_f n_t t_b)) t') t_z

* seq: t' t_z n c .
    staticReachable t' t_z
  ├ staticReachable (Bind n c t') t_z
```

Static Communication

```
datatype tm_id =
  IdBind name
| IdRslt name

fun termId: term -> tm_id where
  n c t' .
  ⊢ termId (Bind n c t') = IdBind n
* n .
  ⊢ termId (Rslt n) = IdRslt n

type tm_id_map = tm_id -> name set

predicate staticSendId: static_value_map -> term -> name -> tm_id -> bool where
  intro: t0 n ne t' nsc nm staticEnv nc .
    staticReachable t0 (Bind n (Sync ne) t'),
    (SAtm (SendEvt nsc nm)) ⊆ staticEnv ne,
    (SChn nc) ∈ staticEnv nsc
  ⊢ staticSendId staticEnv t0 nc (IdBind n)

predicate staticRecvId: static_value_map -> term -> name -> tm_id -> bool where
  intro: t0 n ne t' nrc staticEnv nc .
    staticReachable t0 (Bind n (Sync ne) t'),
    (SAtm (RecvEvt nrc)) ∈ staticEnv ne,
    (SChn nc) ∈ staticEnv nrc
  ⊢ staticRecvId staticEnv t0 nc (IdBind n)
```

Static Communication

```
datatype mode =
  MSeq
| MSpwn
| MCll
| MRtn

type flow = tm_id * mode * tm_id

type graph = flow set

type static_step = tm_id * mode

type static_path = static_step list
```

Static Communication

```
predicate staticFlowsAccept:
  static_value_map -> graph -> term -> bool
where

  result: staticEnv graph n .
  ├ staticFlowsAccept staticEnv graph (Rslt n)

* unit: n t' graph staticEnv  .
    (IdBind n , MSeq, termId t') ∈ graph,
    staticFlowsAccept staticEnv graph t'
  ├ staticFlowsAccept staticEnv graph (Bind n Unt t')

* makeChan: n t' graph staticEnv  .
    (IdBind n , MSeq, termId t') ∈ graph,
    staticFlowsAccept staticEnv graph t'
  ├ staticFlowsAccept staticEnv graph (Bind n MkChn t')

* sendEvt: n t' graph staticEnv  n_c n_m .
    (IdBind n , MSeq, termId t') ∈ graph,
    staticFlowsAccept staticEnv graph t'
  ├ staticFlowsAccept
    staticEnv graph
    (Bind n (Atom (SendEvt n_c n_m)) t')

* recvEvt: n t' graph staticEnv n_c .
    (IdBind n , MSeq, termId t') ∈ graph,
    staticFlowsAccept staticEnv graph t'
  ├ staticFlowsAccept staticEnv graph (Bind n (Atom (RecvEvt n_c)) t')
```

Static Communication

* pair: n t' graph staticEnv n_1 n_2 .
  (IdBind n , MSeq, termId t') $\in$ graph,
  staticFlowsAccept staticEnv graph t'
  $\vdash$ staticFlowsAccept staticEnv graph (Bind n (Atom (Pair n_1 n_2)) t')

* left: n t' graph staticEnv n_s .
  (IdBind n , MSeq, termId t') $\in$ graph,
  staticFlowsAccept staticEnv graph t'
  $\vdash$ staticFlowsAccept staticEnv graph (Bind n (Atom (Lft n_s)) t')

* right: n t' graph staticEnv n_s .
  (IdBind n , MSeq, termId t') $\in$ graph,
  staticFlowsAccept staticEnv graph t'
  $\vdash$ staticFlowsAccept staticEnv graph (Bind n (Atom (Rht n_s)) t')

* function: n t' graph staticEnv t_b n_f n_t .
  (IdBind n , MSeq, termId t') $\in$ graph,
  staticFlowsAccept staticEnv graph t',
  staticFlowsAccept staticEnv graph t_b
  $\vdash$ staticFlowsAccept staticEnv graph (Bind n (Atom (Fun n_f n_t t_b)) t')

Static Communication

```
* spawn: n t' t_c graph staticEnv .
    {
      (IdBind n, MSeq, termId t'),
      (IdBind n, MSpwn, termId t_c)
    } ⊆ graph,
    staticFlowsAccept staticEnv graph t_c,
    staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Spwn t_c) t')

* sync: n t' graph staticEnv n_se .
    (IdBind n, MSeq, termId t') ∈ graph,
    staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Sync n_se) t')

* first: n t' graph staticEnv n_t .
    (IdBind n, MSeq, termId t') ∈ graph,
    staticFlowsAccept staticEnv graph t',
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Fst n_t) t')

* second: n t' graph staticEnv n_t .
    (IdBind n, MSeq, termId t') ∈ graph,
    staticFlowsAccept staticEnv graph t',
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Snd n_t) t')
```

Static Communication

```
* distinction: n t_l t_r t' graph staticEnv n_s .
    {
      (IdBind n, MCll, termId t_l),
      (IdBind n, MCll, termId t_r),
      (IdRslt (resultName t_l), MRtn, termId t'),
      (IdRslt (resultName t_r), MRtn, termId t')
    } ⊆ graph,
    staticFlowsAccept staticEnv graph t_l,
    staticFlowsAccept staticEnv graph t_r,
    staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Case n_s n_l t_l n_r t_r) t')

* application: n t' graph staticEnv n_f n_a .
    ∀ n_f' n_t t_b .
      (SAtm (Fun n_f' n_t t_b)) ∈ staticEnv n_f
    ⟶
      {
        (IdBind n, MCll, termId t_b),
        (IdRslt (resultName t_b), MRtn, termId t')
      } ⊆ graph,
    staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (App n_f n_a) t')
```

Static Communication

```
predicate staticTraceable:
  flow set -> tm_id -> (tm_id -> bool) -> static_path -> bool
where

  empty: start graph isEnd .
    isEnd start
  ├─ staticTraceable graph start isEnd []

* snoc: graph star middle path isEnd end mode .
    staticTraceable graph start (λ l . l = middle) path,
    isEnd end,
    (middle, mode, end) ∈ graph
  ├─ staticTraceable graph start isEnd (path @ [(middle, mode)])
```

Static Communication

```
predicate staticInclusive: static_path -> static_path -> bool where
  first: path1 path2 .
    prefix path1 path2
  ├ staticInclusive path1 path2
* second path2 path1 .
    prefix path2 path1
  ├ staticInclusive path1 path2
* spawnFirst: path n path1 path2 .
  ├ staticInclusive
    (path @ [(IdBind n, MSpwn)] @ path1)
    (path @ [(IdBind n, MSeq)] @ path2)
* spawnSecond: path n path1 path2 .
  ├ staticInclusive
    (path @ [(IdBind n, MSeq)] @ path1])
    (path @ [(IdBind n, MSpwn)] @ path2)
```

Static Communication

```
predicate uncompetitive: static_path -> static_path -> bool where
  ordered: path1 path2 .
    ordered path1 path2
  ├─ uncompetitive path1 path2
* notInclus: path1 path2 .
    ¬ (staticInclusive path1 path2)
  ├─ uncompetitive path1 path2

predicate singular: static_path -> static_path -> bool where
  refl: path .
  ├─ singular path path
* notInclus: path1 path2 .
    ¬ (staticInclusive path1 path2)
  ├─ singular path1 path2
```

Static Communication

```
predicate staticOneToMany: term -> name -> bool where
  intro: staticEnv staticComm t graph n_c .
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    forEveryTwo (staticTraceable graph (termId t)
      (staticSendId staticEnv t n_c)) uncompetitive
  ⊢ staticOneToMany t n_c

predicate staticManyToOne: term -> name -> bool where
  intro: staticEnv statcComm t graph n_c .
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    forEveryTwo (staticTraceable graph (termId t)
      (staticRecvId staticEnv t n_c)) uncompetitive
  ⊢ staticManyToOne t n_c
```

Static Communication

```
predicate staticOneShot: term -> name -> bool where
  intro: staticEnv staticComm t graph n_c .
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    forEveryTwo (staticTraceable graph (termId t)
      (staticSendId staticEnv t n_c)) singular
  ⊢ staticOneShot t n_c

predicate staticOneSync: term -> name -> bool where
  intro: staticEnv staticComm t graph n_c .
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    forEveryTwo (staticTraceable graph (termId t) (staticSendId staticEnv t n_c))
     singular,
    forEveryTwo (staticTraceable graph (termId t) (staticRecvId staticEnv t n_c))
     uncompetitive
  ⊢ staticOneSync t n_c
```