Formal Static Analysis of Synchronization

Thomas Logan

November 3, 2018

1 Introduction

For this master's thesis, I have developed a formal framework for analysis of a concurrent language, along with an initial formal analysis. The language under analysis is a very simplified version of *Concurrent ML* [**?**]. The formal analysis is a recast of an informal analysis developed by Xiao and Reppy [**?**]. It categorizes communication described by the language into simple topologies. One description of topologies is static; that is, it describes topologies in terms of the finite structure of programs. Another description is dynamic; that is, it describes topologies in terms of running a program for an arbitrary number of steps. The main formal theorem states that the static analysis is sound with respect to the dynamic analysis. Two versions of the static analysis have been developed so far, one with lower precision, and one with higher precision. The higher precision analysis is closer to the work by Reppy and Xiao, but has greater complexity than the lower precision analysis. The proofs for the soundness theorem of the lower precision analysis have been mechanically verified using Isabelle [**?**], while the higher precision analysis is currently under development. Indeed, one of the motivations for placing the theory in a formal setting is to enable gradual extension of analysis and language without introducing uncaught bugs in the definitions or proofs. The definitions used in this formal theory differ significantly from that of Reppy and Xiao, in order to aid formal reasoning. Although the definitions are structurally quite different, their philosophical equivalence is hopefully apparent. Thus, recasting Reppy and Xiao's work was far more nuanced than a straightforward syntactic transliteration. In this formal theory, the dynamic semantics of Concurrent ML is defined by a small-step operational semantics. The static semantics is an instance of 0CFA [**?**], defined in terms of constraints [**?**].

2 Concurrent Language

In programing languages, concurrency is a program structuring technique that allows evaluation steps to hop back and forth between disjoint syntactic structures within a program. It is useful when conceptually distinct tasks need to overlap in time, but are easier to understand if they are written as distinct structures within the program. Concurrent languages may also allow the evaluation order between steps of expressions to be nondeterministic. If it's not necessary for tasks to be ordered in a precise way, then it may be better to allow a static or dynamic scheduler pick the most efficient execution order. A common use case for concurrent languages is for programs that interact with humans, in which a program has to process various requests while remaining responsive to subsequent user inputs, and it must continually provide the user feedback with latest information it has processed.

Concurrent ML is a particularly elegant concurrent programing language. It features threads, which are pieces of code allowed to have a wide range of evaluation orders relative to code encapsulated in other threads. Its synchronization mechanism can mandate the execution order between parts of separate threads. It is often the case that synchronization is necessary when data is shared. Thus, in *Concurrent ML*, synchronization is inherent in communication. Threads are spawned asynchronously;

that is, the parent thread need not wait for the spawned thread to terminate before it resumes evaluation. Indeed, if that were the case, then threads would be useless. Additional threads can be spawned in order to share data asynchronously, which can provide better usability or performance under some circumstances.

Threads communicate by having shared access to a common channel. A channel can be used to either send data or receive data. When a thread sends on a channel, another thread must receive on the same channel before the sending thread can continue. Likewise, when a thread receives on a channel, another thread must send on the same channel before the receiving thread can continue.

```
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
```

A given channel can have any arbitrary number of threads sending or receiving data on it over the course of the program's execution. A simple example, derived from Reppy's book *Concurrent Programing in ML* [**?**], illustrates these essential features.

The implementation of `Serv` defines a server that holds a number in its state. When a client gives the server a number `v`, the server gives back the number in its state, and updates its state with the number `v`. The next client request will get the number `v`, and so on. Essentially, a request and reply is equivalent to reading and writing a mutable cell in isolation. The function `make` makes a new server, by creating a new channel `reqCh`, and a loop `loop` which listens for requests. The loop expects the request to be composed of a number `v` and a channel `replCh`. It sends its current state's number on `replCh` and update's the loop's state with the request's number `v`, by calling the loop with a new that number. The server is created with a new thread with the initial state `0` by calling `spawn (fn () => loop 0)`. The request channel is returned as the handle to the server. The function `call` makes a request to the passed in server server `server` with a number `v` and returns a number from the server. Internally, it extracts the request channel `reqCh` from the server handle and creates a new channel `replCh`. It makes a request to the server with the number `v` and the reply channel `replCh` by calling `send (reqCh, (v, replCh))`. Then it receives the reply with the new number by calling `recv replCh`.

```
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) channel
```

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

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

end
```

Concurrent ML actually allows for events other than sending and receiving to occur during synchronization. In fact, the synchronization mechanism is decoupled from events, like sending and receiving, much in the same way that function application is decoupled from function abstraction. Sending and receiving events are represented by `sendEvt` and `recvEvt` and synchronization is represented by `sync`.

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

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

fun send (ch, v) = sync (sendEvt (ch, v))
fun recv v = sync (recvEvt v)
```

An advantageous consequence of decoupling synchronization from events, is that events can be combined with other events via event combinators, and synchronized on exactly once. One such event combinator is `choose`, which constructs a new event consisting of two constituent events, such that when sycnronized on, exactly one of the two events may take effect. There are many other useful combinators, such as the `wrap` and `guard` combinators designed by Reppy[8]. Additionally, Donnelly and Fluet extended *Concurrent ML* with the `thenEvt` combinator described in their work on transactional events [**?**]. Transactional events enable more robust structuring of programs by allowing non-isolated code to be turned into isolated code via the `thenEvt` combinator, rather than duplicating code with the addition of stronger isolation. When the event constructed by the `thenEvt` combinator is is synchronized on, either all of its constituent events and abstractions evaluate in isolation, or none evaluates.

```
val choose : 'a event * 'a event -> 'a event
val thenEvt : 'a event * ('a -> 'b event) -> 'b event
```

3 Synchronization

Synchronization of sending threads and receiving threads requires determining which threads should wait, and which threads should be dispatched. The more amount of information needed to determine this scheduling, the higher the performance penalty. A uniprocessor implementation of synchronization can have very little penalty. Since only one thread can make progress at a time, only one thread requests synchronization at a time, meaning the scheduler won't waste steps checking for threads competing for the same synchronization opportunity, before dispatching. A multiprocessor implementation, on the other hand, must consider that competing threads may exists, therefore perform additional checks. Additionally, there may be overhead in sharing data between processors due to memory hierarchy designs [].

One way to lower synchronization and communication costs is to use specialized implementations for channels that never have more than one thread ever sending or receiving on them. These specialized implementations would avoid unnecessary checks for competing threads. Concurrent ML does not feature multiple kinds of channels distinguished by their communication topologies, i.e. the number of threads that may end up sending or receiving on the channels. However, channels can be classified into various topologies simply by counting the number of threads per channel during the execution of a program. A many-to-many channel has any number of sending threads and receiving threads; a one-to-many channel has at most one sending thread and any number of receiving threads; a many-to-one channel has any number of sending threads and at most one receiving thread; a one-to-one channel has one or none of each; a one-shot channel has exactly one sending attempt.

The implementation of `Serv` is annotated to indicate the communication topologies derived from its usage. Since there are four threads that make calls to the server, the server's particular `reqCh` has four senders. Servers are created with only one thread listening for requests, so the `reqCh` of this server has just one receiver. So the server's `reqCh` is classified as many-to-one. Each use of call creates a distinct new channel `replCh` for receiving data. The function call receives on the channel once and the server sends on the channel once, so each instance of `replCh` is one-shot.

```
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) channel
```

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

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

end
```

4 Implementation

Some hypothetical implementations of specialized and generic Concurrent ML illustrate opportunities for cheaper synchronization. These implementaitons use feasible low-level thread-centric features such as wait and poll. The thread-centric approach allows us to focus on optimizations common to many implementations by decoupling the implementation of communication features from thread scheduling and management. However, a lower level view or scheduler-centric view of synchronization might offer more opportunites for optimization.

Depending on the features provided by existing language implementations, Concurrent ML could be implemented in as a library, as in the case in SML/NJ and MLton. It could also be implemented as features of a language within a compiler and runtime or interpreter. Thus, the implementations shown here can be viewed either as a library or as an intermediate representation within a compiler or interpreter presented with concrete syntax.

```
signature CHANNEL = sig
  type 'a channel
  val channel : unit -> 'a chan
  val send : 'a channel * 'a -> unit
  val recv : 'a channel -> 'a
  end
```

The benefits of specialization would be much more significant in multiprocessor implementations than in uniprocessor implementations. A uniprocessor implementation could avoid overhead caused by contention to acquire locks, by coupling the implementation of channels with scheduling and only scheduling the sending and receiving operations when no other pending operations have yet to start or have already

finished. Reppy's implementation of Concurrent ML uses SML/NJ's first class continuations to implement scheduling and communication as one with very low overhead. In contrast, a multiprocessor implementation would allow threads to run on different processors for increased parallelism, therefore it would not be able to mandate when threads attempt synchronization relative to others without losing the parallel advantage. The cost of trying to achieve parallelism is increased overhead due to contention over acquiring synchronization rights.

A channel can be in one of three states. Either some threads are trying to send through it, some threads are trying to receive from it, or no threads are trying to send or receive on it. Additionally a channel is composed of a mutex lock, so that sending and receiving operations can yield to each other when updating the channel state. When multiple threads are trying to send on a channel, the channel is associated with a queue consisting of messages to be sent, along with conditions waited on by sending threads. When multiple threads are trying to receive on a channel, the channel is associated with a queue consisting of initially empty cells that are accessible by receiving threads and conditions waited on by the receiving threads. The channel content holds the three potential states and their associated content of queues and conditions. The channel is composed of the channel content and also a mutext lock that regulates access to the channel content.

The sending operation acquires the channel's lock to ensure that it updates the channel based on any one of its latest state. If there are threads trying to receive from the channel, the sending operation dequeues an item from the state's associated queue. The item consists of a condition waited on by a receiving thread and an empty cell that can be accessed by the receiving thread. It deposits the message in the cell and signals on the condition, updates the channel state to inactive if there are no further receiving threads waiting, then releases the lock, signals on the condition and returns the unit value. If there are no threads trying to receive from the channel, the sending operation updates the channel state to that of trying to send with an additional condition and message in the associated queue. It releases the lock and waits on the enqueued condition. Once a receiving thread signals on the same condition, the sending operation returns with the unit value.

The receiving operation acquires the channel's lock to ensure that it updates the channel based on any one of its latest state. If there are threads trying to send on the channel, the receiving operation dequeues an item from the state's associated queue. The item consists of a condition waited on by a sending thread along with a message to be sent. It signals the condition and updates the channel state to inactive if there are no further sending threads waiting, then releases the lock and returns the sent message. If there are no threads trying to send on the channel, the receiving operation updates the channel state to that of trying to receive with an additional condition and empty cell in the associated queue. It releases the lock and waits on the enqueued condition. Once a sending thread signals on the same condition, the receiving operation returns with the value deposited in the cell by a sending thread.

```
structure ManyToManyChan : CHANNEL = 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 channel =
  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
```

Implementation of one-to-many channels, compared to that of many-to-many channels, requires fewer steps to synchronize and can execute more steps outside of critical regions, which reduces contention for locks. A channel is composed of a lock and one of three possible states, as is the case for many-to-many channels. However, the state of a thread trying to send need only be associated with one condition and one message.

The sending operation checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to that of trying to send. If successful, it simply waits on sendCond, the condition that a receiving thread will signal on, and then returns the unit value. If the transactional update fails and the state is that of threads trying to receive on the channel, then the sending operation acquires the lock, then dequeues an item from the associated queue where the item consists of recvCond, a condition waited on by a receiving thread, and a cell for depositing the message to that receiving thread. It deposits the message in the cell, updates the state to inactive there are no further items on the queue, then releases the lock. Then it signals on the condition and returns the unit value. The lock is acquired after the state is determined to be that of threads trying to receive, since the expectation is that the current thread is the only one that tries to update the channel from that state. If the communication topology analysis were incorrect and and there were actually multiple threads that could call the sending operation, then there might be data races. Likewise, due to the expectation of a single thread sending on the channel, the sending operation should never witness the state of threads already trying to send.

The receiving operation acquires the lock and checks the state of the channel, just as the receiving operation for many-to-many channels. If the channel is in a state where there is no already trying to send, then it behaves the same as the receiving operation of many-to-many channels. If there is already a thread trying to receive, then it updates the state to inactive and releases the lock. Then it signals on the state's associated condition, which is waited on by a sending thread, and returns the states' associated message.

```
structure FanOutChan : CHANNEL = struct

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

datatype 'a channel =
  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
```

The implementation of many-to-one channels is very similar to that of one-to-many channels.

```
  structure FanInChan : CHANNEL = struct

  datatype 'a chan_content =
    Send of (condition * 'a) queue |
    Recv of condition * 'a option ref |
    Inac
```

```
datatype 'a channel =
  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
```

a one-to-one channel can also be in one of three possible states, but there is no associated lock. Additional, none of the states are associated with queues. Instead, there is a possible state of a thread trying to send, with a condition and a message, or a possible state of a thread trying to receive with a condition and an empty cell, or a possible inactive state. The sending operation checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to that of trying to send. If successful, it simply waits on sendCond, the condition that a receiving thread will signal on, and then returns the unit value. If the transactional update fails and the state is that of a thread trying to receive on

the channel, then it deposits the message in the state's associated cell, updates the channel state to inactive, then signals on the state's associated condition and returns the unit value. If the communication analysis for the channel is correctly one-to-one, then there should be no other thread trying update the state from the state of a thread trying to receive, and no thread modifies that particular state, so no locks are necessary. Likewise, the sending operation should never witness the state of another thread already trying to send, if it is truly one-to-one.

The receiving operation checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to that of trying to receive. If successful, it simply waits on recvCond, the condition that a sending thread will signal on after it deposits a message, and then returns the deposited message. If the transactional update fails and the state is that of a thread trying to send on the channel, then it updates the channel state to inactive, then signals on the state's associated condition and returns the message associated with the sending thread. If the communication analysis for the channel is correctly one-to-one, then there should be no other thread trying update the state from the state of a thread trying to send, and no thread modifies that particular state, so no locks are necessary. Likewise, the receiving operation should never witness the state of another thread already trying to receive, if it is truly one-to-one.

```
structure OneToOneChan : CHANNEL = struct

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

  datatype 'a channel = 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
```

A one-shot channel consists of the same possible states as a one-to-one channel, but is additionally associated with a mutex lock, to account for the fact that multiple threads may try to receive on the channel, even though only at most one message is ever sent.

The sending operation is like that of one-to-one channels, except that if the state is that of a thread trying to receive, it simply deposits the message and signals on the associated condition, without updating the channel's state to inactive, which would be unnecessary, since no further attempts to send are expected.

The receiving operation checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to that of trying to receive. If successful, it simply waits on recvCond, the condition that a sending thread will signal on after it deposits a message, and then returns the deposited message. If the transactional update fails and the state is that of a thread trying to send on the channel, then it acquires the lock, signals on the state's associated condition and returns the message associated with the sending thread, without ever releasing the lock, so that competing receiving threads will know to not progress. If the state is that of a thread trying to receive on the channel, the it acquires the lock, which should block the current thread forever, if there truly is only one send ever.

```
structure OneShotChan : CHANNEL = struct

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

datatype 'a channel = 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 releases lock;
      -* blocks others forever *)
      m |
    Recv _ =>
      acquire lock;
      (* never able to acquire lock;
      -* blocked forever *)
      raise NeverHappens end end

end
```

An even more restrictive version of a channel with at most one send could be used if it's determined that the number of receiving threads is at most one. The one-shot-to-one channel is composed of a possibly empty cell, a condition for a sending thread to wait on, and a condition for a receiving thread to wait on.

The sending operation deposits the message in the cell, signals on the recvCond, waits on the sendCond, and then returns the unit value. The receiving operation waits on the recvCond, signals on the sendCond and then returns the deposited message.

```
structure OneShotToOneChan : CHANNEL = struct

  datatype 'a channel =
    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
```

The example implementations of generic synchronization and specialized synchronization suggest that cost savings of specialized implementation are significant. For example, if you know that a channel has at most one sending thread and one receiving thread, then you will lower synchronization costs by using an implementation that is specialized for one-to-one communication. To be certain that the new program with the specialized implementation behaves the same as the original program with the generic implementation, you need to be certain of three basic properties: that the specialized program behaves the same given one-to-one communication; that you have a procedure to determine the one-to-one communication topology, and that the relation between the procedure's input program and output topology upper bound is sound with respect to the semantics of the program.

Spending your energy to determine the topologies for each unique program and then verifying them for each program would be exhausting. Instead, you would probably rather have a generic procedure that can compute communication topologies for any program in a language, along with a proof that the procedure is sound with respect to the programing language.

This work discusses proofs that the communication topologies are sound with respect to the semantics. Additionally, it would be important to have proofs that the above specialized implementations are equivalent to the many-to-many implementation under the assumption of particular communication topologies.

5 Static Analysis

The right implementation choice can be determined by counting the number threads used during the execution of a program. However, some programs run forever, so it may never be possible to count all the threads. Instead, by using a static analysis it's possible to determine an estimate from the finite structure of programs. A static analysis that describes communication topologies of channels has practical benefits in at least two ways. It can highlight which channels are candidates for optimized implementations of communication; or in a language extension allowing the specification of specialized channels, it can conservatively verify their correct usage. Without a static analysis to check the usage of the special channels, one could inadvertently use a one-shot channel for a channel that has multiple senders, thus violating the intended semantics.

The utility of the static analysis additionally depends on it being precise, sound, and computable. The analysis is precise iff there exist programs about which the analysis describes information that is not directly observable. The analysis is sound iff the information it describes about a program is the same or less precise than the dynamic semantics of the program. The analysis is computable iff there exists an algorithm that determines all the values described by the analysis on any input program.

There are a large number of static analyses with a variety of practical uses. The most familiar kind of static analyses are type systems. A type system statically evaluates expressions in programs to abstract categories known as types. In a sound type

system, if an expression is well typed, then it can be dynamically evaluated (without error). Some expressions cannot be typed, indicating the possibility that the expression cannot be dynamically evaluated. Type systems can improve debugging by pointing out errors that may be infrequently executed. They can also improve execution speeds of safe languages by rendering dynamic checks unnecessary.

Other kinds of analysis are useful for describing opportunities for program optimizations. A data flow analysis describes how data flows from one point to another in the program. A data flow analysis for available expressions tracks the expressions have already been computed by every program point. Using this information, redundant code can be detected and removed to improve performance.

In the following example, an analysis for available expressions would track `(!x + 1)` and `(w - 3)` to remain available from line 5 all the way until the end of the program, meaning those two redundant expressions in line 8 can be swapped with the results of their evaluation. In contrast, `(!y + 2)` would be tracked as available from line 5 until line 7. It would be unavailable after line 7 because the number in `y` is modified in line 7, changing the how the expression would evaluate after that.

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

A data flow analysis for liveness tracks the set of stored valued that are used in the remainder of the program. Using this information, unused code can be detected and removed to improve performance. There are other uses of this analysis, as well. In this work, the liveness of channels is tracked to distinguish between distinct channels with the same name.

In the following example, an analysis for liveness of values stored in names and references would track `x` to be live from line 7 until line 5. Above line 5 there are no uses of `x`, so it would not be live. Since the name `x` at line 2 is not used, line 2 can be removed. The reference contained in `z` would be live from line 7 until line 6, where it is assigned. There is no deference of lstinlinez above line 6, so it not live from there until the beginning. Since the reference contained in `z` is not live at line 4, line 4 can be removed.

```
1. let
2.   val x = 1
3.   val y = 2
4.   val z = ref (4 * 73)
5.   val x = 4 in
6.   z := 1;
7.   x * !z end
```

The information at each program point is derived from control structures in the program, which dictate how information flows between program points. Some uses of control structures are literally represented in the syntax, such as the sequencing of namings and assignments in the previous examples. Other uses of control structures may be indirectly represented through names. Function application is a control structure that allows a calling piece of code to flow into a function abstraction's code. Function abstractions can be named, which allows multiple pieces of code to all flow into into the same section of code. The name adds an additional step in to uncover control structures, and determine data flow. Additionally, in languages with higher order functions and recursion, such as those in the Lisp and ML families, it may be impossible to determine all the function abstractions that expressions resolve to. However, a control flow analysis can reveal a good approximation of the control structures and values that have been obfuscated by higher order function abstractions. Uncovering the the control structures depends on resolving expressions to values, and resolving expressions to values depends on on uncovering the control structures. The mutual dependency means that control flow analysis is a form of static semantics that describes abstract evaluations of programs. In this work, control flow analysis is used for tracking certain kinds of values, like channels and events, in addition to constructing precise data flow analysis.

6 Specification

Analyses can be described in a variety of ways. A computable algorithm that take programs as input and produces information about the behavior as output is ideal for automation. A non-algorithmic relation (without an inherent deterministic reasoner), stated in terms of programs and execution information, may be more suitable for clarity of meaning and correctness with respect to the operational semantics. However, a non-algorithmic relation can be translated into an algorithm. One rather mechanical method essentially involves specifying a reasoner associated with the relation. First, the reasoner generates a comprehensive set of data structures representing constraints from the relation's description, then the reasoner the constraints.

For a subset of Concurrent ML without event combinators, Reppy and Xiao developed an efficient algorithm that determines for each channel, all possible threads that send and receive on it. The algorithm depends on each primitive operation in the program being labeled with a program point. A sequence of program points ordered in a valid execution sequence forms a control path. Distinction between threads in a program can be inferred from whether or not their control paths diverge.

The algorithm proceeds in multiple steps that produce intermediate data structures, used for efficient lookup in the subsequent steps. It starts with a control-flow analysis [] that results in multiple mappings. One mapping is from names to abstract values that may bind to the names. Another mapping is from channel-bound names to abstract values that are sent on the respective channels. Another is from function-bound names to abstract values that are the result of respective function applications. It constructs a control-flow graph with possible paths for pattern matching and thread spawning determined directly from the primitives used in the program. Relying on

information from the mappings to abstract values, it constructs the possible paths of execution via function application and channel communication. It uses the graph for live variable analysis of channels, which limits the scope for the remaining analysis. Using the spawn and application edges of the control-flow graph, the algorithm then performs a data-flow analysis to determine a mapping from program points to all possible control paths leading into the respective program points. Using the CFA's mappings to abstract values, the algorithm determines the program points for sends and receives per channel name. Then it uses the mapping to control paths to determine all control paths that send or receive on each channel, from which it classifies channels as one-shot, one-to-one, many-to-one, one-to-many, or many-to-many.

7 Soundness

Reppy and Xiao informally prove soundness of their analysis by showing that their static analysis determines that more than one thread sends (or receives) on a channel if the execution allows more than one to send (or receive) on that channel. The proof of soundness depends on the ability to relate the execution of a program to the static analysis of a program. The static analysis describes threads in terms of control paths, since it can only describe threads in terms of statically available information. Thus, in order to describe the relationship between the threads of the static analysis and the operational semantics, the operational semantics is defined as stepping between sets of control paths paired with terms. Divergent control paths are added whenever a new thread is spawned.

The semantics and analysis must contain many details. To ensure the correctness of proofs, it is necessary to check that there are no subtle errors in either the definitions or proofs. Proofs in general require many subtle manipulations of symbols. The difference between a false statement and a true statement can often be difficult to spot, since the two may be very similar lexically. However, a mechanical proof checker, such as that of Isabelle, has no difficulty discerning between valid and invalid derivations. Mechanical checking of proofs can notify users of errors in the proofs or definitions far better and faster than manual checking. This work has greatly benefited from Isabelle's proof checker in order to correctly define the language semantics, control flow analysis, communication analysis, and other helpful definitions. For instance, some bugs in the definintions were found trying to prove soundness. The proof checker would not accept the proof unless I provided facts that should be false, indicating that the definitions did not state my intentions. After correcting the errors in the definitions, the proof was completed such that the proof checker was satisfied.

8 Formal Theory

The definitions and theorems of this work were constructed in the formal language of Isabelle/HOL, to enable mechanical verification of the correctness of the proofs. However, the formal logic used for presentation in this paper has been somewhat modified from Isabelle's syntax. To aid the development of formal proofs, the anal-

yses are stated using non-algorithmic specifications. The static relations are defined with syntax-directed structures, providing strong evidence of computability. That is, a relation about a program could be described by the same relation on a subprogram, but certainly not by the original program, or a larger program. This work does not contain formal proofs that the specialized implementations are behaviorally equivalent to a generic implementation, but the example implementations should provide good for that.

In order to relate the static analyses to the operational semantics, I borrowed Reppy and Xiao's strategy of stepping between sets of control paths and their associated terms.

It is neccessary to prove additional soundness theorems about the static semantics en route to proving soundness of communication classification. Restricting the grammar to a form that requires every abstraction and application to be bound to a name allows the operational semantics to maintain static program information necessary for proofs of soundness []. The semantics are defined as an environment based operational semantics, rather than a substitution based operational semantics. By avoiding simplification of terms in the operational semantics, it is possible to relate the abstract values of the static semantics to the values produced by the dynamic semantics, which in turn is relied on to prove soundness of the static semantics.

The restricted ANF grammar is impractical for a programer to write, yet it is still practical for a language under automated analysis since there is a straightforward procedure to transform more flexible and user-friendly grammars into the restricted form as demonstrated by Flanagan et al []. Additionally, the restricted grammar melds nicely with the control path semantics. Instead of relying on additional meta-syntax for program points of primitive operations, the analysis can simply use the required names of the restricted grammar to identify locations in the program. Control paths are simply sequences of let bound names.

9 Syntax

The syntax used in this formal theory contains a very small subset of *Concurrent ML*'s features. The features include recursive function abstraction with application, left and right construction with pattern matching, pair construction with first and second projections, sending and receiving event construction with synchronization, channel creation, thread spawning, and the unit literal. The syntax is defined in a way to make it possible to relate the dynamic semantics of programs to the syntax programs. The syntax is in a very restrictive administrative normal form (ANF), in which every expression is bound to a name. Furthermore, expression only accept names in place of eagerly evaluated inputs.

```
datatype name = Nm string

datatype
  program =
    Let name expression program |
    Rslt name and
```

```
expression =
  Unt | MkChn |
  Prim primitive |
  Spwn program |
  Sync name |
  Fst name | Snd name |
  Case name name program name program |
  App name name and

primitive =
  SendEvt name name | RecvEvt name |
  Pair name name |
  Lft name | Rght name |
  Abs name name exp
```

10 Dynamic Semantics

The dynamic semantics is described by how programs evaluate to values. A history of execution is represented a list of binding names, or resulting names, along with modes of control, such as sequencing, spawning, calling, or returning. Channels have no literal representation, but each time a channel is created, it is uniquely identified by the history of the execution up until that point. Primitive expressions are not simplified. Instead, primitives are evaluated to closures consisting of the primitive syntax, along with an environment that maps its constituent names to their values.

```
datatype dynamic_point =
  DNxt name | DSpwn name | DCll name | DRtn name

type dynamic_path = dynamic_point list

datatype channel =
  Chan dynamic_path name

datatype dynamic_value =
  VUnt | VChn channel | VPrm primitive (name -> dynamic_value option)

type environment =
  name -> dynamic_value option
```

The evaluation of some expressions results in sequencing, meaning there is no coordination with other threads, and there is no backtracking. These expressions are the unit literal, primitives, pairs, and first and second projections. The evaluation depends only on the syntax and an environment for looking up the values of names within the syntax. Additionally, all these expressions evaluate to values in a single step.

```
predicate seq_eval of expression -> environment -> dynamic_value -> bool:
  only

  (∀ env .
      seq_eval Unt env VUnt),

  (∀ p env .
      seq_eval (Prim p) env (VPrm p env)),

  (∀ env x_p x1 x2 env_p v.
    if
      env x_p = Some (VPrm (Pair x1 x2) env_p),
      env_p x1 = Some v
    then
      seq_eval (Fst x_p) env v),

  (∀ env x_p x1 x2 env_p v .
    if
      env x_p = Some (VPrm (Pair x1 x2) env_p),
      env_p x2 = Some v
    then
      seq_eval (Snd x_p) env v)
```

The evaluation of expressions for application, and matching result in calling, which leads to backtracking after the called programs are evaluated. The evaluation depends on the syntax and an environment for looking up the values of names within the syntax. The expressions are evaluated to a called program, and a new environment that will later be used in evaluation of the called program. For matching, either the left or the right program is called, and the environment is updated with the corresponding name mapping to the value extracted from the pattern. For application, the program inside of an applied abstraction is called, and the environment is updated with the abstraction's parameter mapped to the application's argument. The environment is also updated with the recursive name mapped to the same applied abstraction.

```
predicate call_eval of expression -> env -> program -> env -> bool:
  only

  (∀ env x_s x_c env_s v x_l e_l x_r e_r .
    if
      env x_s = Some (VPrm (Lft x_c) env_s),
      env_s x_c = Some v
    then
      call_eval (Case x_s x_l e_l x_r e_r) env e_l (env(x_l -> v))),

  (∀ env x_s x_c env_s v x_l e_l x_r e_r .
    if
      env x_s = Some (VPrm (Rght x_c) env_s);
      env_s x_c = Some v
    then
```

```
    call_eval (Case x_s x_l e_l x_r e_r) env e_r (env(x_r -> v))),

(∀ env f f_p x_p e_b env_l x_a v .
  if
    env f = Some (VPrm (Abs f_p x_p e_b) env_l);
    env_l x_a = Some v
  then
    call_eval (App f x_a) env e_b (
      env_l(f_p -> (VPrm (Abs f_p x_p e_b) env_l), x_p -> v)))
```

Backtracking is made possible by recording the programs that have been sidestepped and returned to them later. In addition to the syntactic program, the environment for resolving the program's names, and an unresolved name are also recorded together to form a continuation. The initial state of execution consists of a program, an empty environment, and an empty stack of continuations. With each sequential step, the program is reduced to a subprogram, and the environment is updated with the name bound to the value of the expression. Each time a subprogram is sidestepped to evaluate a dependent expression, a continuation is formed around it and pushed onto a stack of continuations. A continuation is popped of the stack when a state's program is reduces to a result program. A pool of states keeps track of all the states that have been reached through the evaluation of an initial program. Each state is indexed by the dynamic path taken to reach it. A pool's leaf path indicates a state that has yet to be evaluated. Additionally, The conversation between threads is also recorded as a set of correspondences consisting of the path to the sending state, the path to the receiving state, and the channel used for communication.

```
datatype continuation =
  Ctn name program env

type stack = continuation list

datatype state =
  Stt program env stack

type pool =
  dynamic_path -> state option

predicate leaf of pool -> dynamic_path -> bool:
  only

  (∀ pool path stt .
    if
      pool path = Some stt,
      (∄ path' stt' .
        pool path' = Some stt',
        strict_prefix path path')
    then
      leaf pool path)
```

```
type conversation =
  (dynamic_path * channel * dynamic_path) set
```

The evaluation of a program may involve evaluation of multiple threads concurrently and also communication between threads. Since pools contain multiple states and paths, they can accommodate multiple threads as well. A single evaluation step depends on one pool and evaluates to a new pool based on one or more states in that pool. The initial pool for a program contains just one state indexed by an empty path. The state contains the program, an empty environment, and an empty stack. The pool will grow strictly larger with each evaluation step, maintaining a full history. Each step adds new states and paths extended from previous ones, and each point in the path indicates the mode of transition to take to reach the state. Only states indexed by leaf paths are used to evaluate to the next pool.

A sequencing evaluation step of a program picks a leaf state and relies on sequential evaluation of its top expression. It updates the state's environment with the value of the expression, leaves the stack unchanged, and reduces the program to the next subprogram. A calling evaluation step relies on the calling evaluation of a state's top expression. The binding name, subprogram, environment are pushed onto the stack, and the new state gets its program and environment from the evaluation of the expression.

For the evaluation a leaf path pointing to a result program, a continuation is popped of the stack, the new state's program is taken from the continuation, and the new state's environment is taken from the continuation and modified with the result value.

In the case of channel creation, the evaluation updates the state's environment with the value of a channel consisting of the path leading to its creation; it leaves the stack unchanged and reduces the program to the next subprogram.

In the case of spawning, the evaluation updates the state's environment with the unit value, leaves the stack unchanged, and reduces the program to the next subprogram. Additionally, it generates a another state consisting of the spawning expression's child program, the same environment unchanged, and an empty stack.

the evaluation is updated with two new paths extending the leaf path. For one, the leaf path is extended with a sequential program point whose state has the next expression and the environment updated with the unit value bound to the let binding name, and the original continuation stack. For the other, the leaf path extended with an program point indicating a spawning transition. Its state has the spawned expression, the original environment, and an empty continuation stack.

In the case where two leaf paths in the pool correspond to synchronization on the same channel, and one synchronizes on a send event and the other synchronizes on a receive event, then evaluation updates the pool with two new paths and corresponding states. It updates the send event's state with its subprogram, the environment updates with the unit value, and the stack unchanged. It updates the receive event's state with its subprogram, the environment updates with the sent value, and the stack unchanged.

Additionally, the conversation is updated with the sending and receiving paths,

and the channel that the synchronization used for communication.

```
predicate dynamic_eval of
  pool -> conversation -> pool -> conversation -> bool:
  only

  (∀ pool path x env x_k e_k env_k stack' v convo .
    if
      leaf pool path,
      pool path = Some (Stt
        (Rslt x) env ((Ctn x_k e_k env_k) # stack')),
      env x = Some v
    then
      dynamic_eval
        pool convo
        (pool(path @ [DRtn x] -> (Stt e_k env_k(x_k -> v) stack')))
        convo),

  (∀ pool path x b e' env stack v .
    if
      leaf pool path,
      pool path = Some (Stt (Let x b e') env stack),
      seq_eval b env v
    then
      dynamic_eval
        pool convo
        (pool(path @ [DNxt x] -> (Stt e (env(x -> v)) stack)))
        convo),

  (∀ pool path x b e' env stack e_u env_u convo .
    if
      leaf pool path,
      pool path = Some (Stt (Let x b e') env stack),
      call_eval b env e_u env_u
    then
      dynamic_eval
        pool convo
        (pool(path @ [DCll x] ->
          (Stt e_u env_u ((Ctn x e' env) # stack))
        convo),

  (∀ pool path x e' env stack .
    if
      leaf pool path,
      pool path = Some (Stt (Let x MkChn e') env stack)
    then
      dynamic_eval
        pool convo
        (pool(path @ [DNxt x] ->
          (Stt e' (env(x -> (VChn (Chan path x)))) stack)))
```

```
      convo),

(∀ pool path x e_c e' env stack convo .
  if
    leaf pool path,
    pool path = Some (Stt (Let x (Spwn e_c) e') env stack)
  then
    dynamic_eval
    pool convo
    (pool(
        path @ [DNxt x] -> (Stt e' (env(x -> VUnt)) stack),
        path @ [DSpwn x] -> (Stt e_c env [])))
    convo),

(∀ pool path_s path x_s x_se e_s env_s stack_s x_sc x_m env_se
  path_r x_r x_re e_r env_r stack_r x_rc env_re c convo .
  if
    leaf pool path_s,
    pool path_s = Some (Stt
      (Let x_s (Sync x_se) e_s) env_s stack_s),
    env_s x_se = Some (VPrm
      (SendEvt x_sc x_m) env_se),
    leaf pool path_r,
    pool path_r = Some (Stt
      (Let x_r (Sync x_re) e_r) env_r stack_r),
    env_r x_re = Some (VPrm
      (RecvEvt x_rc) env_re),
    env_se x_sc = Some (VChn c),
    env_re x_rc = Some (VChn c),
    env_se x_m = Some v_m
  then
    dynamic_eval
      pool convo
      (pool(
          path_s @ [DNxt x_s] ->
            (Stt e_s (env_s(xs -> VUnt)) stack_s),
          path_r @ [DNxt x_r] ->
            (Stt e_r (env_r(x_r -> v_m))stack_r)))
      (convo ∪ {(path_s, c, path_r)}))
```

11 Dynamic Communication

The dynamic one shot classification is described by pool where there is only one dynamic path that synchronizes and sends on a given channel. Whether or not two attempts to synchronize on a channel are competitive can be determined by looking at the paths of the pool. If two paths are ordered, that is, one is the prefix of the other or vice versa, then necessarily occur in sequence, so the shorter path synchronizes

before the longer path. Two paths may be competitive only if they are unordered. The dynamic many-to-one classification means that there is no competition on the receiving end of a channel; any two paths that synchronize to receive on a channel are ordered. The dynamic one-to-many classification means that there is no competition on the sending end of a channel; any two paths that synchronize to send on a channel are ordered. The dynamic one-to-one classification means that there is no competition on either the receiving or the sending ends of a channel; any two paths that synchronize on a channel are necessarily ordered for either end of the channel.

```
predicate is_send_path of
  pool -> channel -> dynamic_path -> bool:
  only

  (∀ pool path x x_e e' env stack x_sc x_m env_e c.
    if
      pool path = Some (Stt (Let x (Sync x_e) e') env stack),
      env x_e = Some (VPrm (SendEvt x_sc x_m) env_e),
      env_e x_sc = Some (VChn c)
    then
      is_send_path pool c path)

predicate is_recv_path of pool -> channel -> dynamic_path -> bool:
  only
  (∀ poo path x x_e e' env stack x_rc env_e c .
    then
      pool path = Some (Stt (Let x (Sync x_e) e') env stack),
      env x_e = Some (VPrm (RecvEvt x_rc) env_e),
      env_e x_rc = Some (VChn c)
    then
      is_recv_path pool c path)

predicate every_two of ('a -> bool) -> ('a -> 'a -> bool) -> bool:
  only
  (∀ p r .
    if
      (∀ path1 path2 .
        if p path1, p path2 then r path1 path2)
    then
      every_two p r)

predicate ordered of 'a list -> 'a list -> bool:
  only
  (∀ path1 path2 . if prefix path1 path2 then
    ordered path1 path2),
  (∀ path2 path1 . if prefix path2 path1 then
    ordered path1 path2)

predicate one_shot of pool -> channel -> bool:
  only
  (∀ pool c .
```

```
    if
      every_two (is_send_path pool c) (op =)
    then
      one_shot pool c)

predicate one_to_many of pool -> channel -> bool:
  only
  (∀ pool c .
    if
      every_two (is_send_path pool c) ordered
    then
      one_to_many pool c)

predicate many_to_one of pool -> channel -> bool:
  only
  (∀ pool c .
    if
      every_two (is_recv_path pool c) ordered
    then
      many_to_one pool c)

predicate one_to_one of pool -> channel -> bool:
  only
  (∀ pool c.
    if
      one_to_many pool c,
      many_to_one pool c
    then
      one_to_one pool c)
```

12 Static Semantics

The static semantics is described by an estimation of intermediate static values and subprograms that might result from running a program. The static semantics helps to deduce static information about channels and events, which is crucial for statically deducing information about synchronization on channels and communication topologies. The static values consist of the static unit value, static channels, and static primitive values. The static unit value is no less precise than the dynamic unit value, but static channels and static primitive values are imprecise versions of their dynamic counterparts. The static channel is identified only by the name it binds to at creation time, rather than the full path that leads up to its creation. A static primitive value is simply a primitive expression without an environment for looking up its named arguments. The static environment contains the internal evaluation results by associating names to static values. In order to find the return value of a program or subprogram, it is useful to fetch the name embedded within its result subprogram.

```
datatype static_value =
```

```
  SChn name | SUnt | SPrm primitive

type static_value_map = name -> static_value set

fun result_name of program -> name:
  (∀ x .
    result_name (Rslt x) = x),
  (∀ x b e' .
    result_name (Let x b e') = (result_name e))
```

The static evaluation is a control flow analysis described by a relation from an input program to two outputs, each a static environment. The first output contains names associated with the evaluations of expressions that bind to those names in the program. The second output contains names associated with values that might be sent over channels identified by those name.

The definition of static evaluation is syntax-directed, meaning it should be possible to prove any correct evaluation simply by unraveling the program in to smaller and smaller programs, until a theorem or axiom is induced. The definition also appears to allow deterministic reasoning. With both determinism and termination, it would be possible to compute the static evaluation outputs from the input program. This certainly appears likely, but it has not been formally proven in this work.

```
predicate static_eval of
  static_value_map -> static_value_map -> program -> bool:
  only
  (∀ env_a convo_a x .
    static_eval env_a convo_a (Rslt x)),

  (∀ env_a x convo_a e' .
    if
      SUnt ∈ env_a x,
      static_eval env_a convo_a e'
    then
      static_eval env_a convo_a (Let x Unt e')),

  (∀ x env_a convo_a e' .
    if
      (SChn x) ∈ env_a x,
      static_eval env_a convo_a e'
    then
      static_eval env_a convo_a (Let x MkChn e')),

  (∀ x_c x_m env_a x convo_a e' .
    if
      (SPrm (SendEvt x_c x_m)) ∈ env_a x,
      static_eval env_a convo_a e'
    then
      static_eval env_a convo_a (Let x (Prim (SendEvt x_c x_m)) e')),
```

```
(∀ x_c env_a x convo_a e' .
  if
    (SPrm (RecvEvt x_c)) ∈ env_a x,
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Prim (RecvEvt x_c)) e')),

(∀ x1 x2 env_a x convo_a e' .
  if
    (SPrm (Pair x1 x2)) ∈ env_a x,
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Prim (Pair x_1 x_2)) e)),

(∀ x_s env_a x convo_a e' .
  if
    (SPrm(Lft x_s)) ∈ env_a x,
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Prim (Lft x_s)) e')),

(∀ x_s env_a x convo_a e' .
  if
    (SPrm(Rght x_s)) ∈ env_a x,
    static_eval env_a convo_a e
  then
    static_eval env_a convo_a (Let x (Prim (Rght x_s)) e')),

(∀ f x_p e_b env_a convo_a x e' .
  if
    (SPrm (Abs f x_p e_b)) ∈ env_a f,
    static_eval env_a convo_a e_b,
    (SPrm (Abs f x_p e_b)) ∈ env_a x,
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Prim (Abs f x_p e_b)) e')),

(∀ f x_p e_b env_a convo_a x e' .
  if
    SUnt ∈ env_a f,
    static_eval env_a convo_a e_c,
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Spwn e_c) e')),

(∀ env_a x_e x convo_a  e'.
  if
    (∀ x_sc x_m x_c .
      if
        (SPrm (SendEvt x_sc x_m)) ∈ env_a x_e,
```

```
        SChn x_c ∈ env_a x_sc
      then
        SUnt ∈ env_a x, env_a x_m ⊆ convo_a x_c),
    (∀ x_rc x_c .
      if
        (SPrm (RecvEvt x_rc)) ∈ env_a x_e,
        SChn x_c ∈ env_a x_rc,
      then
        convo_a x_c ⊆ env_a x),
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Sync x_e) e')),

(∀ env_a x_p x convo_a e' .
  if
    (∀ x1 x2 . if (SPrm (Pair x1 x2)) ∈ env_a x_p then
      env_a x1 ⊆ env_a x),
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Fst x_p) e')),

(∀ env_a x_p x convo_a e' .
  if
    (∀ x1 x2 . if (SPrm (Pair x1 x2)) ∈ env_a x_p then
      env_a x2 ⊆ env_a x),
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Snd x_p) e')),

(∀ env_a x_s x_l e_l x convo_a x_r e_r e' .
  if
    (∀ x_c . if (SPrm (Lft x_c)) ∈ env_a x_s then
      env_a x_c ⊆ env_a x_l,
      env_a (result_name e_l) ⊆ env_a x,
      static_eval env_a convo_a e_l),
    (∀ x_c . if (SPrm (Rght x_c)) ∈ env_a x_s then
      env_a x_c ⊆ env_a x_r,
      env_a (result_name e_r) ⊆ env_a x,
      static_eval env_a convo_a e_r),
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Case x_s x_l e_l x_r e_r) e')),

(∀ env_a f x_a x convo_a e' .
  if
    (∀ f_p x_p e_b . if (SPrm (Abs f_p x_p e_b)) ∈ env_a f then
      env_a x_a ⊆ env_a x_p,
      env_a (result_name e_b) ⊆ env_a x),
    static_eval env_a convo_a e'
  then
```

```
      static_eval env_a convo_a (Let x (App f x_a) e'))
```

It is very straightforward to follow the rules of static evaluation in order to build up functions mapping names to static values for the static environment and the static conversation. The example server implementation recasted into the ANF grammar is helpful for demonstrating this informal procedure.

```
let u1 = unt in
let r1 = rght u1 in
let l1 = lft r1 in
let l2 = lft l1 in

let mksr = (fun _ x2 =>
  let k1 = mk_chn in
  let srv = fun srv' x3 =>
    let e1 = recv_evt k1 in
    let p1 = sync e1 in
    let v1 = fst p1 in
    let k2 = snd p1 in
    let e2 = send_evt k2 x3 in
    let z5 = sync e2 in
    let z6 = srv' v1 in
    let u4 = unt in
    rslt u4 in
  let z7 = spawn (
    let z8 = srv r1 in
    let u5 = unt in
    rslt u5) in
  rslt k1) in

let rqst = (fun _ x4 =>
  let k3 = fst x4 in
  let v2 = snd x4 in
  let k4 = mk_chn in
  let p2 = pair v2 k4 in
  let e3 = send_evt k3 p2 in
  let z9 = sync e3 in
  let e4 = recv_evt k4 in
  let v3 = sync e4 in
  rslt v3) in

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

We start at the top of the program and pick a rule from the definition of static evaluation that could hold. Then we choose the smallest function that satifies that rule's conditions. In the server implementation, the program starts with
`let u1 = unt in ...`, which only matches the rule concluding with
`static_eval env_a convo_a (Let (Nm "u1") Unt ...)`. The premises for that rule require that `SUnt ∈ env_a u1 and static_eval env_a convo_a ....`
Therefore, we choose the smallest function we can determine so far for the static environment `env_a`. The smallest function we can determine for `SUnt ∈ env_a (Nm "u1")` is `λ x . if x = (Nm "u1") then {SUnt} else {}`.
Since there's no premise that directly states what's required of the static conversation, we can simply choose an empty set to start with. The second premise is an inductive static evaluation on the subprogram, which indicates that we should repeat this procedure again for the remainder of the program, incrementally adding more static values for each name in the program. We then repeat the procedure from the top of the program until there's nothing more to add to the functions. To make the presentation clear, the syntactic sugar `(r1 -> {right u1}, ...)` is used to mean `λ x . if x = (Nm "r1") then {SPrm (Rght (Nm "u1"))} else ... else {}`. The representation of static values closely resembles the concrete syntax.

```
val server_static_env of name -> static_vale set:
  server_static_env = (
    u1 -> {unt},
    r1 -> {rght u1},
    l1 -> {lft r1},
    l2 -> {lft l1},
    mksr -> {fun  _ x2 => ...},
    x2 -> {unt},
    k1 -> {chn k1},
    srv -> {fun srv' x3 => ...},
    srv' -> {fun srv' x3 => ...},
    x3 -> {rght u1, lft r1, lft l1},
    e1 -> {recv_evt k1},
    p1 -> {pair v2 k4},
    v1 -> {lft r1, lft l1},
    k2 -> {chn k4},
    e2 -> {send_evt k2 x3},
    z5 -> {unt},
    z6 -> {unt},
    u4 -> {unt},
    z7 -> {unt},
    z8 -> {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 -> {send_evt k3 p2},
```

```
    z9 -> {unt},
    e4 -> {recv_evt k4},
    v3 -> {rght u1, lft r1, lft r2},
    srvr -> {chn k1},
    z10 -> {unt},
    p3 -> {pair srvr l1},
    z11 -> {rght u1, lft r2},
    p4 -> {pair srvr l2},
    z12 -> {rght u1, lft l1})

val server_static_convo of name -> static_vale set:
  server_static_convo = (
    k1 -> {pair v2 k4},
    k4 -> {rght u1, lft l1, lft l2})
```

The static reachability is described by subprograms that might be reached from a larger program. A program is reachable from itself. An initial program might also reach any that is reachable from its sub program.

```
predicate static_reachable of program -> program -> bool:
  only

  (∀ e .
    static_reachable e e),

  (∀ e_c e_z x e' .
    if
      static_reachable e_c e_z
    then
      static_reachable (Let x (Spwn e_c) e') e_z),

  (∀ e_l e_z x x_s x_l x_r e_r e' .
    if
      static_reachable e_l e_z
    then
      static_reachable (Let x (Case x_s x_l e_l x_r e_r) e') e_z),

  (∀ e_r e_z x x_s x_l e_l x_r e' .
    if
      static_reachable e_r e_z
    then
      static_reachable (Let x (Case x_s x_l e_l x_r e_r) e') e_z),

  (∀ e_b e_z x f x_p e_b e' .
    if
      static_reachable e_b e_z
    then
      static_reachable (Let x (Prim (Abs f x_p e_b)) e') e_z),

  (∀ e' e_z x b .
```

```
    if
      static_reachable e' e_z
    then
      static_reachable (Let x b e') e_z)
```

13 Static Communication

To describe communication statically, it is helpful to identify the entry point of each subprogram with a short description. The program ID of a let-binding is identified by the binding name, and indication that it's a let-binding. The program ID of a result is indicated by its embedded name, and indication that it's a result.

```
datatype program_id = NLet name | NResult var

fun top_program_id of program -> program_id:
  (∀ x b e' .
    top_program_id (Let x b e') = NLet x),
  (∀ x .
    top_program_id (Rslt x) = NResult x)

type program_id_map = program_id -> name set
```

The static communication is described by an estimation of the static paths that communicate on static channels. The static send ID classification means that an program ID might represent a sycnronization to send on a given abstract channel. The static receive ID classification means that an program ID might represent a synchronization to receive on a given abstract channel.

```
predicate static_send_id of
  static_value_map -> program -> name -> program_id -> bool:
  only
  (∀ e0 x x_e e' x_sc x_m env_a x_c .
    if
      static_reachable e0 (Let x (Sync x_e) e'),
      (SPrm (SendEvt x_sc x_m)) ⊆ env_a x_e,
      (SChn x_c) ∈ env_a x_sc
    then
      static_send_id env_a e0 x_c (NLet x))

predicate static_recv_id of
  static_value_map -> program -> name -> program_id -> bool:
  only
  (∀ e0 x x_e e' x_rc env_a x_c .
    if
      static_reachable e0 (Let x (Sync x_e) e'),
      (SPrm (RecvEvt x_rc)) ∈ env_a x_e,
      (SChn x_c) ∈ env_a x_rc
    then
```

```
      static_recv_id env_a e0 x_c (NLet x))
```

In the server implementation, the static channel identified by the name `k1` is waited on by the server. It has one receiving ID in the server abstraction at ID `let p1` and a sending ID in the request abstraction at ID `let z9`. The channel idetifiend by the name `k4` is sent with a client's request for the server to reply on. It has a receiving ID in the request abstraction at `let v3` and a sending ID in the server abstraction at `let z5`.

Reppy and Xiao's work relies on detecting the liveness of channels in order to gain higher precision in the static classification of communication. Since formal proofs are inherently complicated with numerous details, it was easier to first formally prove soundness for a version without the added complication of considering liveness of channel.

The definitions are purposely structured to allow adding live channel analysis to the definition fairly straightforward with just a few alterations. Section ? expands on these alterations and outlines a strategy that is likely to result in formal proofs of soundness, although the actual formal proof of the version with live channel analysis is not yet been complete.

For lower precision version without the liveness of channels, there are four modes, indicating how the program ID transitions to the program ID of one of its subprograms. The modes of transition are sequencing, calling, spawning, and returning. A transition is a triplet of a program ID, a mode of transition, and an program ID of a subprogram. A static point is just a program ID along with the mode it uses to transition to its subprogram. A static path is a list of static points.

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

type transition = program_id * mode * program_id

type static_program = transition set

type static_point = program_id * mode

type static_path = static_point list
```

The static traversability means that a set of transitions describes all the transitions that might be traversed for a program, given bindings of abstract values. For a result expression, any transition might be traversed. For all let expressions, except those binding to case matching and function application, the sequential transition from the top expression to the sequenced expression might be traversable, and the traversable transitions are also the traversable transitions for the sequenced expression. For binding to function abstraction, the traversable transitions are also traversable transitions for the inner expression of the function abstraction. For binding to spawning, the spawning transition from the top expression to the spawned expression might be traversed, and the traversable transitions are also traversable transitions for the spawned expression.

In the case of matching, the the calling transition from the top program to the left subprogram might be traversable, and the calling transition from the top program to the right subprogram might be traversable. The returning transition from the result of the left subprogram to the sequenced subprogram might be traversable, and the returning transition from the result of the right subprogram to the sequenced subprogram might be traversable. Additionally, the traversable transitions for the top program are also traversable transitions the left subprogram, right subprogram, and the sequenced subprogram.

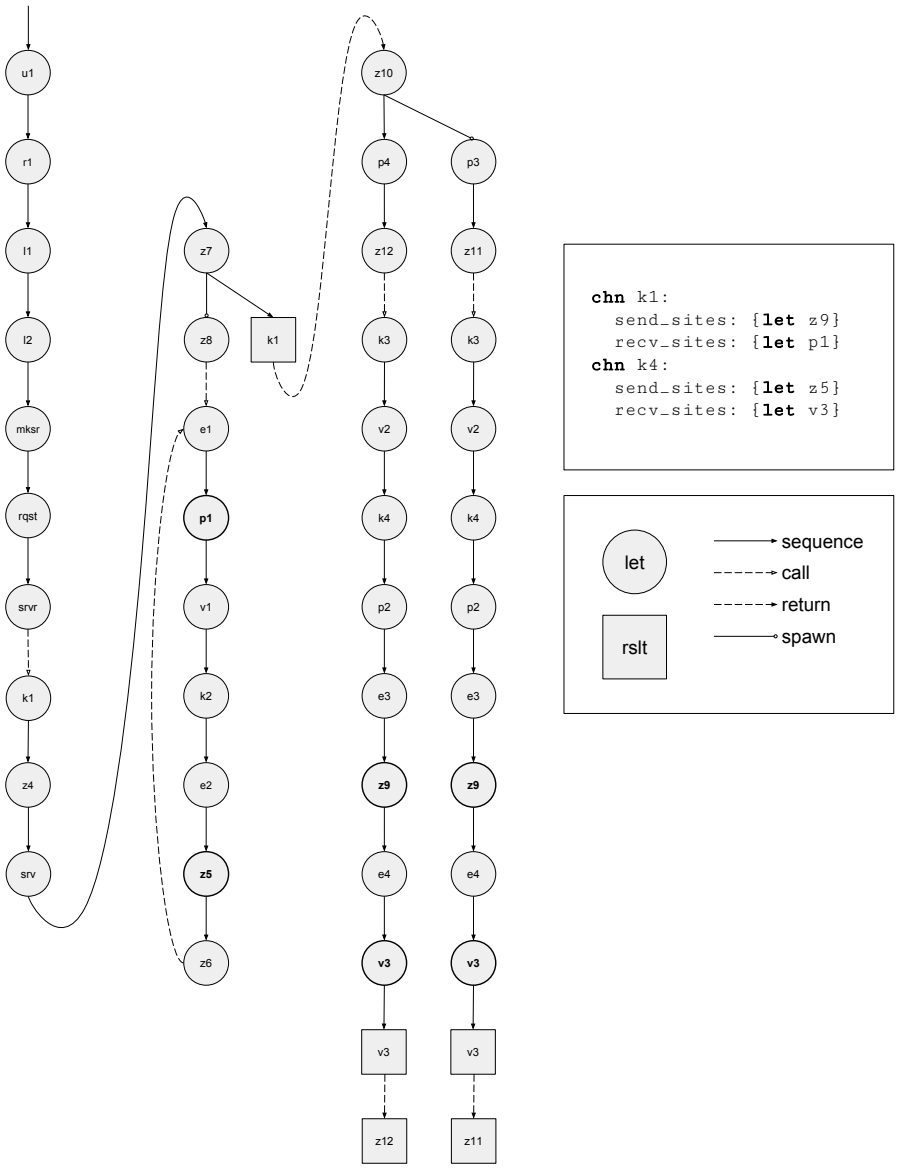
In the case of application, if the applied name is actually bound to a function abstraction, then a calling transition from the top program to the abstraction's subprogram might be traversable, and the returning transition from the result of the function abstraction to the sequenced subprogram might be traversable. Additionally, the traversable transitions for the top program are also traversable transitions for the sequenced subprogram.

```
predicate static_traversable of
  static_value_map -> static_program -> program -> bool:
  only

  (∀ env_a prog_a x .
    static_traversable env_a prog_a (Rslt x)),

  (∀ x e' prog_a env_a  .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x Unt e')),

  (∀ x e' prog_a env_a  .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x MkChn e')),

  (∀ x e' prog_a env_a  x_c x_m .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x (Prim (SendEvt x_c x_m)) e')
  ),

  (∀ x e' prog_a env_a x_c .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
```

```
    static_traversable env_a prog_a (Let x (Prim (RecvEvt x_c)) e')),

(∀ x e' prog_a env_a x1 x2 .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Prim (Pair x1 x2)) e')),

(∀ x e' prog_a env_a x_s .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Prim (Lft x_s)) e')),

(∀ x e' prog_a env_a x_s .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Prim (Rght x_s)) e')),

(∀ x e' prog_a env_a e_b f x_p .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e',
    static_traversable env_a prog_a e_b
  then
    static_traversable env_a prog_a (Let x (Prim (Abs f x_p e_b)) e')),

(∀ x e' e_c prog_a env_a.
  if
    {(NLet x, MNxt, top_program_id e'),
      (NLet x, MSpwn, top_program_id e_c)} ⊆ prog_a,
    static_traversable env_a prog_a e_c,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Spwn e_c) e')),

(∀ x e' prog_a env_a x_se .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Sync x_se) e')),

(∀ x e' prog_a env_a x_p .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
```

```
    static_traversable env_a prog_a e',
  then
    static_traversable env_a prog_a (Let x (Fst x_p) e')),

(∀ x e' prog_a env_a x_p .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e',
  then
    static_traversable env_a prog_a (Let x (Snd x_p) e')),

(∀ x e_l e_r e' prog_a env_a x_s .
  if
    {(NLet x, MCll, top_program_id e_l),
      (NLet x, MCll, top_program_id e_r),
      (NResult (result_name e_l), MRtn, top_program_id e'),
      (NResult (result_name e_r), MRtn, top_program_id e')} ⊆ prog_a,
    static_traversable env_a prog_a e_l,
    static_traversable env_a prog_a e_r,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Case x_s x_l e_l x_r e_r) e
')),

(∀  env_a f x e' x x_a .
  if
    (∀ f_p x_p e_b . if (SPrm (Abs f_p x_p e_b)) ∈ env_a f then
      {(NLet x, MCll, top_program_id e_b),
        (NResult (result_name e_b), MRtn, top_program_id e')} ⊆ prog_a
),
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (App f x_a) e'))
```

The server implementation represented as a graph illustrates how static traversability can interpret a static program from a dynamic program.



The static traceability means that a static path with a given starting point, and ending condition, can be traced by traversing the transitions in a static program. The empty path is statically traceable if the starting point meets the ending condition. Otherwise, a path is statically traceable if the last static point corresponds to a transition that meets the ending condition, and the longest strict prefix of the path is statically traceable.

```
predicate static_traceable of
  transition set -> program_id -> (program_id -> bool) -> static_path ->
  bool:
  only

  (∀ start prog_a is_end .
    if
      is_end start
    then
      static_traceable prog_a start is_end []),

  (∀  prog_a star middle path is_end end mode .
    if
      static_traceable prog_a start (λ l . l = middle) path,
      is_end end,
      (middle, mode, end) ∈ prog_a
    then
      static_traceable prog_a start is_end (path @ [(middle, mode)]))
```

In the static program of the server implementation, there are two paths each corresponding to its own thread that lead to sending on static channel **chn** k1 and a potentially infinite number of paths that lead to receiving on channel **chn** k1, but all on the same thread. There are an infinite number of paths that lead to sending on static channel **chn** k4, and two paths that lead to receiving on static channel **chn** k4. This is certainly imprecise, as the static **chn** k4 corresponds to multiple distinct dynamic channels, each with just one sender and one receiver. The higher precision analysis discussed in section ? addresses this issue.

The static inclusion means that two static paths might be traced in the same run of a program. Ordered paths might be inclusive, and also a path that diverges from another at a spawn transition might be inclusive. This concept is useful for achieving greater precision, since if two paths cannot occur in the same run of a program, only one needs to be counted towards the communication topology.

```
predicate static_inclusive of static_path -> static_path -> bool:
  only

  (∀ path1 path2 .
    if
      prefix path1 path2
    then
      static_inclusive path1 path2),

  (∀ path2 path1 .
    if
      prefix path2 path1
    then
      static_inclusive path1 path2),

  (∀ path x path1 path2 .
```

```
    static_inclusive
      (path @ (NLet x, MSpwn) # path1)
      (path @ (NLet x, MNxt) # path2)),

  (∀ path x path1 path2 .
    static_inclusive
      (path @ (NLet x, MNxt) # path1)
      (path @ (NLet x, MSpwn) # path2))
```

The singularness means that two paths are the same or only of them can occur in a given run of a program. The noncompetitiveness means states that two paths can't compete during a run of a program, since they are ordered or cannot occur in the same run of a program.

```
predicate singular of static_path -> static_path -> bool:
  only

  (∀ path .
    singular path path),

  (∀ path1 path2 .
    if
      not (static_inclusive path1 path2)
    then
      singular path1 path2)

predicate noncompetitive of static_path -> static_path -> bool:
  only

  (∀ path1 path2 .
    if
      ordered path1 path2
    then
      noncompetitive path1 path2),

  (∀ path1 path2 .
    if
      not (static_inclusive path1 path2)
    then
      noncompetitive path1 path2)
```

The static one-shot classification means that there is at most one attempt to synchronize to send on a static channel in any run of a given program.

The static one-to-one classification means that there is at most one thread that attempts to send and at most one thread that attempts to receive on a given static channel for any time during a run of a given program.

The static one-to-many classification means that there is at most one thread that attempts to send on a given static channel at any time during a run of a given program, but there may be many threads that attempt to receive on the channel. The static

many-to-one predicate means that there may be many threads that attempt to send on a static channel, but there is at most one thread that attempts to receive on the channel for any time during a run of a given program.

```
predicate static_one_shot of static_value_map -> program -> name -> bool:
  only
  (∀ prog_a e env_a x_c .
    if
      every_two
        (static_traceable prog_a (top_program_id e) (static_send_id env_a
   e x_c))
        singular,
      static_traversable env_a prog_a e
    then
      static_one_shot prog_a e x_c)

predicate static_one_to_one of static_value_map -> program -> name ->
  bool:
  only
  (∀ prog_a e env_a x_c .
    if
      (every_two
        (static_traceable prog_a (top_program_id e) (static_send_id env_a
   e x_c))
        noncompetitive),
      (every_two
        (static_traceable prog_a (top_program_id e) (
  static_recv_program_id env_a e x_c))
        noncompetitive),
      static_traversable env_a prog_a e
    then
      static_one_to_one env_a e x_c)

predicate static_one_to_many of static_value_map -> program -> name ->
  bool:
  only
  (∀ prog_a e env_a x_c .
    if
      (every_two
        (static_traceable prog_a (top_program_id e) (static_send_id env_a
   e x_c))
        noncompetitive),
      static_traversable env_a prog_a e
    then
      static_one_to_many env_a e x_c)

predicate static_many_to_one of static_value_map -> program -> name ->
  bool:
  only
  (∀ prog_a e env_a x_c .
```

```
  if
    (every_two
      (static_traceable prog_a (top_program_id e) (
static_recv_program_id env_a e x_c))
      noncompetitive),
    static_traversable env_a prog_a e
  then
    static_many_to_one env_a e x_c)
```

14 Reasoning

The reasoning involved in proving the soundness of each communication topology classification is based around breaking the goal into simpler subgoals, and generalizing assumptions to create useful induction hypotheses. It is often useful to create helper definitions that can be deduced from premises of the theorem being proved and enable general reasoning across arbitrary programs. A frequent pattern is to define predicates in terms of semantic structures, like the environment, stack, and pool, and deduce the instantiation of these predicates on the initial program state.

Some aspects of the generalized predicate definitions exist simply to prove that they imply instantiations of the original program based predicates. However, the generalized definitions exist in order to allow direct access to properties that would otherwise be deeply nested in an inductive structure and inaccessible by a predictable number of logical steps for an arbitrary program.

One of the most difficult aspects of formal reasoning is in developing adequate definitions. It is often possible to define a single semantics in multiple ways. For instance, the sortedness of a list could be defined in terms of the sortedness of its tail or in terms of the sortedness of its longest strict prefix. To prove theorems relating sortedness to other relations, it may be important that the other relations are inductively defined on the same subpart of the list. Some relations may only be definable on the tail, while others can be defined only on the strict prefix. In such cases, it is necessary to define sortedness in two ways, and prove their equivalence, in order to prove theorems relating to less flexible relations.

```
predicate sorted_a of nat list -> nat list -> bool:
  only

  (sorted_a []),

  (∀  x .
      sorted_a [x]),

  (∀  x y zs .
    if
      x ≤ y,
      sorted_a (y # zs)
    then
      sorted_a (x # y # zs))
```

```
predicate sorted_b of nat list -> nat list -> bool:
  only

  (sorted_b []),

  (∀  x .
      sorted_b [x]),

  (∀  xs y z .
    if
      sorted_b (xs @ [y]),
      y ≤ z
    then
      sorted_b (xs @ [y] @ [z]))
```

The theorem for soundness of static one-shot classification states that if a static channel is statically classified as one-shot for a given program and static environment consistent with the program, then any corresponding dynamic channel is classified as one-shot over any pool that results from running the program. The theorem for soundness of static fan-out classification states that if a static channel is statically classified as fan-out for a given program and static environment consistent with that program, then any corresponding dynamic channel is classified as fan-out over any pool that results from running the program. The theorems for soundness of many-to-one classification and one-to-one classification follow the same pattern.

```
theorem static_one_shot_sound:
  ∀ env_a convo_a e0 x_c pool convo path_c .
    if
      static_eval env_a convo_a e0,
      static_one_shot env_a e0 x_c,
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo
    then
      one_shot pool (Chan path_c x_c)

theorem static_one_to_many_sound:
  ∀ env_a convo_a e0 x_c pool convo path_c.
    if
      static_eval env_a convo_a e0,
      static_one_to_many env_a e0 x_c,
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo
    then
      one_to_many pool (Chan path_c x_c)

theorem static_many_to_one_sound:
  ∀ env_a convo_a e0 x_c pool convo path_c.
    if
      static_eval env_a convo_a e0,
      static_many_to_one env_a e0 x_c,
```

```
      star dynamic_eval [[] -> (Stt e [->] [])] {} pool convo
    then
      many_to_one pool (Chan path_c x_c)

theorem static_one_to_one_sound:
  ∀ env_a convo_a e0 x_c pool convo path_c.
    if
      static_eval env_a, convo_a e0,
      static_one_to_one env_a e0 x_c,
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo
    then
      one_to_one pool (Chan path_c x_c)
```

The soundness theorem for static fan-out classification is proved by a few simpler lemmas and the definitions of static and dynamic fan-out classification. The three main lemmas state the soundness of the lack of static traceability, the soundness of the lack of inclusiveness, and the soundness of a program point not being a static send ID. These lemmas depend on a correspondence between static paths and dynamic paths, which is bijective for the lower precision analysis. The lemma for soundness of lack of static inclusiveness states that any two dynamic paths traced by running a program correspond to statically inclusive static paths. It follows from a straightforward case analysis of static inclusivity. The lemma for soundness of lack of static traceability states that for any dynamic path traced by running a program, there is a corresponding static path that is statically traceable. The lemma for soundness of a program point not being a static send ID states that running a program reaches a synchronization on a sending event, then that synchronization is statically identified as a send ID by its program ID.

```
lemma not_static_traceable_sound:
  ∀ e0 pool convo path x b e' env stack evn_a convo_a prog_a is_end .
    if
      star dynamic_eval ([[] -> (Stt e0 [->] [])], {}) (pool, convo),
      pool path = Some (Stt (Let x b e') env stack),
      static_eval env_a convo_a e0,
      static_traversable env_a prog_a e0,
      is_end (NLet x)
    then
      ∃ path_a .
        paths_correspond path path_a,
        static_traceable prog_a (top_program_id e0) is_end path_a

lemma not_static_inclusive_sound:
  ∀ e0 pool convo path1 stt1 path2 stt2 path_a1 path_a2 .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo
      pool path1 = Some stt1,
      pool path2 = Some stt2,
      paths_correspond path1 path_a1,
      paths_correspond path2 path_a2
    then
```

```
    static_inclusive path_a1 path_a2

lemma not_static_send_id_sound:
  ∀ e0 pool convo path x x_e e' env stack x_sc x_m env' path_c x_c .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo,
      pool path = Some (Stt (Let x (Sync x_e) e') env stack),
      env x_e = Some (VPrm (SendEvt x_sc x_m) env'),
      env' x_sc = Some (VChn (Chan path_c x_c)),
      static_eval env_a convo_a e0
    then
      static_send_id env_a e0 x_c (NLet x)
```

The soundness of lack of static traceability is proved by generalizing static traversability and static evaluation over pools, such that information about a point in the program can be deduced by a fixed number of logical steps regardless of the location of the program point or the size of the program. Without such generalization, it would be possible to prove soundness for a fixed program, but not any arbitrary program.

The generalization of static traversability is composed of in static traversable over values, static traversability over environments, static traversability over stacks, and static traversability over pools. In most cases, it simply states that a subprogram of some semantic element is also statically traversable. The exception is in the case of static traversability over a non-empty stack, where there is an additional condition that the transition from a result id to the program ID of the continuation program exists in the static program. This information is consistent with static traversability over programs, but provides direct information about a transition in the static program, which would otherwise only be deducible by a varying number of logical steps depending on the program.

```
predicate static_traversable_val of
  static_value_map -> static_program -> dynamic_value -> bool:
  only

  (∀ env_a prog_a .
    static_traversable_val env_a prog_a VUnt),

  (∀ env_a prog_a c .
    static_traversable_val env_a prog_a (VChn c)),

  (∀ env_a prog_a env x_c x_m.
    if
      static_traversable_env env_a prog_a env
    then
      static_traversable_val
        env_a prog_a (VPrm (SendEvt x_c x_m) env)),

  (∀ env_a prog_a env x_c.
    if
      static_traversable_env env_a prog_a env
    then
```

```
      static_traversable_val
        env_a prog_a (VPrm (RecvEvt x_c) env)),

  (∀ env_a prog_a env x_p .
    if
      static_traversable_env env_a prog_a env
    then
      static_traversable_val
        env_a prog_a (VPrm (Lft x_p) env)),

  (∀ env_a prog_a env x_p .
    if
      static_traversable_env env_a prog_a env
    then
      static_traversable_val
        env_a prog_a (VPrm (Rght x_p) env)),

  (∀ env_a prog_a e_b env f x_p .
    if
      static_traversable env_a prog_a e_b,
      static_traversable_env env_a prog_a env
    then
      static_traversable_val
        env_a F (VPrm (Abs f x_p e_b) env)),

  (∀ env_a prog_a env .
    if
      static_traversable_env env_a prog_a env
    then
      static_traversable_val
        env_a prog_a (VPrm (Pair x1 x2) env))

predicate static_traversable_env of
  static_value_map -> static_program -> env -> bool:
  only
  (∀ env_a prog_a env .
    if
      (∀ x v . if env x = Some v then
        static_traversable_val env_a prog_a v)
    then
      static_traversable_env env_a prog_a env)

predicate static_traversable_stack of
  static_value_map -> static_program ->
  name -> continuation list -> bool:
  only
  (∀ env_a prog_a y .
    static_traversable_stack env_a prog_a y []),

  (∀ y e prog_a env_a prog_a env stack x env .
```

```
    if
      {(NResult y, MRtn, top_program_id e)} ⊆ prog_a,
      static_traversable env_a prog_a e,
      static_traversable_env env_a prog_a env,
      static_traversable_stack env_a prog_a (result_name e) stack
    then
      static_traversable_stack env_a prog_a y ((Ctn x e env) # stack))

predicate static_traversable_pool of
  static_value_map -> static_program -> pool -> bool:
  only
  (∀ env_a prog_a pool .
    if
      (∀ path e env stack .
        if
          env path = Some (Stt e env stack)
        then
          static_traversable env_a prog_a e,
          static_traversable_env env_a prog_a env,
          static_traversable_stack env_a prog_a (result_name e) stack)
    then
        static_traversable_pool env_a prog_a pool)
```

The transitions described by the various versions of static traversability depend on static environments in order to look up the control flow in the application case. The static environment results from the static evaluation of the program that is dynamically evaluated. Thus, generalized versions of static evaluation enable further deduction about transitions. As with the generalized versions of static traversability, the generalized versions of static evaluation are designed to preserve static environments across dynamic evaluations of pools. They also provide direct access to binding information from names to static values in a fixed number of logical steps. Static evaluation of programs correlates program syntax to static values, but the generalized static evaluations correlate dynamic semantic structures, like, value, environments, and stacks, to static values. The abstraction function relates dynamic values to static values and helps the larger goal of relating dynamic semantic elements to static values and static environments.

```
fun abstract of dynamic_value -> static_value:
  abstract VUnt = SUnt,
  (∀ path x .
    abstract (VChn (Chan path x)) = SChn x),
  (∀ primitive env .
    abstract (VPrm primitive env) = SPrm prim)

predicate static_eval_value of
  static_value_map -> abstract_convo -> dynamic_value -> bool:
  only

  (∀ env_a convo_a .
```

```
    static_eval_val env_a convo_a VUnit),

  (∀ env_a convo_a c .
    static_eval_val env_a convo_a (VChn c)),

  (∀ env_a convo_a env x_c x_m .
    if
      static_eval_env env_a convo_a env
    then
      static_eval_val env_a convo_a
        (VPrm (SendEvt x_c x_m) env)),

  (∀ env_a convo_a env x_c  .
    if
      static_eval_env env_a convo_a env
    then
      static_eval_val env_a convo_a
        (VPrm (RecvEvt x_c) env)),

  (∀ env_a convo_a env x_p  .
    if
      static_eval_env env_a convo_a env
    then
      static_eval_val env_a convo_a
        (VPrm (Lft x_p) env)),

  (∀ env_a convo_a env x_p  .
    if
      static_eval_env env_a convo_a env
    then
      static_eval_val env_a convo_a
        (VPrm (Rght x_p) env)),

  (∀ f x_p e_b env_a convo_a env .
    if
      {APrim (Abs f x_p e_b)} ⊆ env_a f,
      static_eval env_a convo_a e_b,
      static_eval_env env_a convo_a env
    then
      static_eval_val env_a convo_a
        (VPrm (Abs f x_p e_b) env)),

  (∀ env_a convo_a env x1 x2 .
    if
      static_eval_env env_a convo_a env
    then
      static_eval_val env_a convo_a
        (VPrm (Pair x1 x2) env))

predicate static_eval_env of
```

```
  static_value_map -> static_value_map -> env -> bool:
  only
  (∀ env_a convo_a env .
    if
      (∀ x v . if env x = Some v then
        {abstract v} ⊆ env_a x,
        static_eval_val env_a convo_a v)
    then
      static_eval_env env_a convo_a env)

predicate static_eval_stack of
  static_value_map -> static_value_map ->
  static_value set -> continuation list -> bool:
  only
  (∀ env_a convo_a res_a .
    static_eval_stack env_a convo_a res_a []),
  (∀ res_a env_a convo_a .
    if
      res_a ⊆ env_a x,
      static_eval env_a convo_a e,
      static_eval_env env_a convo_a env,
      static_eval_stack env_a convo_a env_a (result_name e) stack
    then
      static_eval_stack env_a convo_a res_a ((Ctn x e env) # stack))

predicate static_eval_pool of
  static_value_map -> static_value_map -> pool -> bool:
  only
  (∀ env_a convo_a pool .
    if
      (∀ path e env stack .
        if
          pool path = Some (Stt e env stack)
        then
          static_eval env_a convo_a e,
          static_eval_env env_a convo_a env,
          static_eval_stack env_a convo_a env_a (result_name e) stack)
    then
      static_eval_pool env_a convo_a pool)
```

A variant of star that inducts toward the left of the transitive connection is helpful for relating dynamic traceability to static traceability, since it mirrors the direction that way paths grow, which influenced the choice of induction in the definition of static traceability.

```
predicate star_left of ('a -> 'a -> bool) -> 'a -> 'a -> bool:
  only

  (∀ r z z .
    star_left r z z),
```

```
  (∀ r x y z .
    if
      star_left r x y, r y z
    then
      star_left r x z)

lemma star_implies_star_left:
  ∀ r x y .
    if
      star r x z
    then
      star_left r x z

lemma star_left_trans:
  ∀ r x y z .
    if
      star_left r x y,
      star_left r y z
    then
      star_left r x z
```

The soundness of lack of static traceability follows from the generalized lemma of soundness of lack of static traceability over pools, which contains the generalized premise of static traceability, and also it follows from the lemma of preservation of static traversability over pools from an initial pool to any pool resulting from multiple steps of dynamic evaluation.

```
lemma not_static_traceable_pool_sound:
  ∀ e0 pool convo path x b e' env stack evn_a convo_a prog_a is_end .
    if
      star dynamic_eval ([[] -> (Stt e0 [->] [])], {}) (pool, convo),
      pool path = Some (Stt (Let x b e') env stack),
      static_eval env_a convo_a e0,
      static_traversable env_a prog_a pool,
      is_end (NLet x)
    then
      ∃ path_a .
        paths_correspond path path_a,
        static_traceable prog_a (top_program_id e0) is_end path_a

lemma static_traversable_pool_preserved_star:
  ∀ e0 pool convo env_a convo_a prog_a .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo,
      static_eval env_a convo_a e0,
      static_traversable_pool env_a prog_a [[] -> (Stt e0 [->] [])]
    then
      static_traversable_pool env_a prog_a pool
```

The lemma for the soundness of the generalized form follows from the generalized definitions of static traceability. The preservation of static traversability over pools is proved by the equivalence between star and its leftward variant, and induction on the leftward variant.

The lemma for soundness of a static point not being a send ID is proved using the lemma for soundness of a name in a sending event of a synchronization not statically binding to a static channel value, and the lemma for soundness of a name in any state's environment not statically binding to a static value. Since only send IDs are relevant the soundness of lack of static reachability is used to ensure that the static point discussed is indeed a send ID.

```
lemma send_chan_not_static_bound_sound:
  ∀ e0 pool convo env_a convo_a path
    x x_e e' env stack x_sc x_m env_e path_c x_c .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])], {} pool convo,
      static_eval env_a convo_a e0,
      pool path = Some (Stt (Let x (Sync x_e) e') env stack),
      env_y x_e = Some (VPrm (SendEvt x_sc x_m) env_e),
      env_e x_sc = Some (VChn (Chan path_c x_c))
    then
      SChn x_c ∈ env_a x_sc

lemma not_static_bound_sound:
  ∀ e0 pool convo env_a convo_a path e env stack x v .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo,
      static_eval env_a convo_a e0,
      pool path = Some (Stt e env stack),
      env x = Some v
    then
      abstract v ∈ env_a x

lemma not_static_reachable_sound:
  ∀ e0 pool convo env_a convo_a path e env stack .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo,
      pool path = Some (Stt e env stack)
    then
      static_reachable e0 e
```

Both the soundness of a name in a sending event of a synchronization not statically binding to a static channel value, and the soundness of a name in any state's environment not statically binding to a static value from the preservation of static evaluation over multiple steps of dynamic evaluation.

```
lemma static_eval_pool_preserved:
  ∀ pool convo pool' convo' env_a convo_a .
    if
      star dynamic_eval pool convo pool' convo'
```

```
    static_eval_pool env_a convo_a pool
  then
    static_eval_pool env_a convo_a pool'
```

The lemma for soundness of lack of static reachability relies on the a reformulation of static reachability that inducts on the left program. The definition of static reachability is syntax-directed in order to portray a clear connection to a computable algorithm that can determine the reachable expression from an initial program. However, to show that an expression is reachable from the initial program, it is necessary to show that each intermediate expression is reachable from the initial expression. Thus, the induction needs to enable unraveling the goals from the end to the beginning of the program, maintaining the initial program state in context for each subgoal .

```
predicate static_reachable_left of program -> program -> bool:
  only

  (∀ e0 e .
    static_reachable_left e0 e0),

  (∀ e0 x e_c e' .
    if
      static_reachable_left e0 (Let x (Spwn e_c) e')
    then
      static_reachable_left e0 e_c),

  (∀ e0 x x_s x_l e_l x_r e_r e' .
    if
      static_reachable_left e0 (Let x (Case x_s x_l e_l x_r e_r) e')
    then
      static_reachable_left e0 e_l),

  (∀ e0 x x_s x_l e_l x_r e_r e' .
    if
      static_reachable_left e0 (Let x (Case x_s x_l e_l x_r e_r) e')
    then
      static_reachable_left e0 e_r),

  (∀ e0 x f x_p e_b e' .
    if
      static_reachable_left e0 (Let x (Prim (Abs f x_p e_b)) e')
    then
      static_reachable_left e0 e_b),

  (∀ e0 x f x_p e_b e' .
    if
      static_reachable_left e0 (Let x b e')
    then
      static_reachable_left e0 e')
```

```
predicate static_reachable_over_prim of program -> primitive -> bool:
  only

  (∀ e0 x_c x_m .
    static_reachable_over_prim e0 (SendEvt x_c x_m)),

  (∀ e0 x_c .
      static_reachable_over_prim e0 (RecvEvt x_c)),

  (∀ e0 x1 x2 .
      static_reachable_over_prim e0 (Pair x1 x2)),

  (∀ e0 x_l .
      static_reachable_over_prim e0 (Lft x_l)),

  (∀ e0 x_r
      static_reachable_over_prim e0 (Rght x_r)),

  (∀ e0 e_b f x_p e_b .
    if
      static_reachable_left e0 e_b
    then
      static_reachable_over_prim e0 (Abs f x_p e_b))

predicate static_reachable_val of program -> dynamic_value -> bool:
  only

  (∀ e0 .
      static_reachable_over_val e0 VUnt),

  (∀ e0 c .
      static_reachable_over_val e0 (VChn c)),

  (∀ e0 p env .
    if
      static_reachable_over_prim e0 p,
      static_reachable_over_env e0 env
    then
      static_reachable_over_val e0 (VPrm p env))

predicate static_reachable_env of program -> env -> bool:
  only
  (∀ e0 env
    if
      (∀ x v .
        if
          env x = Some v
        then
          static_reachable_over_val e0 v)
    then
```

```
      static_reachable_over_env e0 env)

predicate static_reachable_over_stack of
  program -> continuation list -> bool:
  only
  (∀ e0 .
    static_reachable_over_stack e0 []),
  (∀ e0 e_k env_k stack' .
    if
      static_reachable_left e0 e_k,
      static_reachable_over_env e0 env_k,
      static_reachable_over_stack e0 stack'
    then
      static_reachable_over_stack e0 ((Ctn x_k e_k env_k # stack')))

predicate static_reachable_pool of program -> pool -> bool:
  only
  (∀ e0 pool .
    then
      (∀ path e env stack . if pool path = Some (Stt e env stack) then
        static_reachable_left e0 e,
        static_reachable_over_env e0 env,
        static_reachable_over_stack e0 stack)
    then
      static_reachable_over_pool e0 pool)
```

The soundness of lack of reachability follows the the definitions a generalized form of soundness over pools.

```
lemma not_static_reachable_pool_sound:
  ∀ e0 pool .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])], {} pool convo
    then
      static_reachable_over_pool e0 pool
```

The soundness over pools follows from the lemma that the leftward static reachability implies the rightward (and syntax-directed) static reachability, and the equivalence between star and the leftward star. It relies on induction of the leftward star and constructs the static reachability proposition using the leftward definition.

```
lemma static_reachable_left_implies_static_reachable:
  ∀ e0 e.
    if
      static_reachable_left e0 e
    then
      static_reachable e0 e
```

```
lemma static_reachable_trans:
  ∀ e1 e2 e3 .
    if
      static_reachable e1 e2,
      static_reachable e2 e3
    then
      static_reachable e1 e3
```

The lemma that the leftward variant of static reachability implies the syntax-directed static reachability follows from induction on the leftward static reachability and the transitivity of static reachability, which follows from induction on static reachability.

15 Higher Precision Communication

In many programs, channels are created within function abstractions. The function abstractions may be applied multiple times, creating multiple distinct channels with each application. It may be that each channel is used just once and then discarded. However, the static analysis just described would identify all the distinct channels by the same name, since each distinct channel is created by the same piece of syntax. Thus, it would classify those channels as being used more than once.

It is possible to be more precise by trimming the program under analysis down to just the part where the static channel is live. The static channel cannot be live between the last use of a dynamic channel and the creation of a new dynamic channel with the same name. Thus, each truncated program would have just one dynamic channel corresponding to the static channel under analysis.

A trimmed graph structure of static program is used static analysis for this higher precision analysis, which can better differentiate between distinct channels. A trimmed static program is specialized for a particular dynamic channel. From the creation point, it must contain transitive transitions to all the program points where the channel is live. It should also be as small as possible, for higher precision.

In the whole static program used in the previous analysis, a spawning transition connects a child thread to the rest of the program. For a trimmed static program, it may be clear the channel of interest is not created until after the spawn point, so there is not need to include the spawning transition. However, later on in the program it may become apparent that the channel of interest is sent via another channel to that spawned thread. Since there is no spawning transition already connecting that thread to the trimmed static program, a transition with a sending mode is used between the send ID and the receive ID of synchronization. Modes for typical control flow of sequencing, calling, returning, and spawning are also included transitions.

```
datatype mode = MNxt | MSpwn | ESend name | MCll | MRtn

type transition = program_id * mode * program_id
```

```
type static_point = program_id * mode

type static_path = static_point list
```

Static traversability is similar to that of the previous analysis. However, it must additionally consider transitions with the sending mode.

```
predicate static_traversable of static_value_map -> static_program ->
  program -> bool:
 only

  (∀ env_a prog_a x .
      static_traversable env_a prog_a (Rslt x)),

  (∀ x e' prog_a env_a  .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x Unt e')),

  (∀ x e' prog_a env_a  .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x MkChn e')),

  (∀ x e' prog_a env_a  x_c x_m .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x (Prim (SendEvt x_c x_m)) e')
  ),

  (∀ x e' prog_a env_a x_c .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x (Prim (RecvEvt x_c)) e')),

  (∀ x e' prog_a env_a x1 x2 .
    if
      (NLet x , MNxt, top_program_id e') ∈ prog_a,
      static_traversable env_a prog_a e'
    then
      static_traversable env_a prog_a (Let x (Prim (Pair x1 x2)) e')),

  (∀ x e' prog_a env_a x_s .
```

```
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Prim (Lft x_s)) e')),

(∀ x e' prog_a env_a x_s .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Prim (Rght x_s)) e')),

(∀ x e' prog_a env_a e_b f x_p .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e',
    static_traversable env_a prog_a e_b
  then
    static_traversable env_a prog_a (Let x (Prim (Abs f x_p e_b)) e')),

(∀ x e' e_c prog_a env_a.
  if
    {(NLet x, MNxt, top_program_id e'),
      (NLet x, MSpwn, top_program_id e_c)} ⊆ prog_a,
    static_traversable env_a prog_a e_c,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Spwn e_c) e')),

(∀ x e' prog_a env_a x_se .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    (∀ x_sc x_m x_c y.
      if
        (SPrm (SendEvt x_sc x_m)) ∈ env_a xSE,
        (SChn x_c) ∈ env_a x_sc,
        static_recv_program_id env_a e' x_c (NLet y)
      then
        (NLet x, ESend x_se, NLet y) ∈ F),
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Sync x_se) e')),

(∀ x e' prog_a env_a x_p .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e',
  then
    static_traversable env_a prog_a (Let x (Fst x_p) e')),
```

```
(∀ x e' prog_a env_a x_p .
  if
    (NLet x , MNxt, top_program_id e') ∈ prog_a,
    static_traversable env_a prog_a e',
  then
    static_traversable env_a prog_a (Let x (Snd x_p) e')),

(∀ x e_l e_r e' prog_a env_a x_s .
  if
    {(NLet x, MCll, top_program_id e_l),
      (NLet x, MCll, top_program_id e_r),
      (NResult (result_name e_l), MRtn, top_program_id e'),
      (NResult (result_name e_r), MRtn, top_program_id e')} ⊆ prog_a,
    static_traversable env_a prog_a e_l,
    static_traversable env_a prog_a e_r,
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Case x_s x_l e_l x_r e_r) e
')),

(∀  env_a f x e' x x_a .
  if
    (∀ f_p x_p e_b . if (SPrm (Abs f_p x_p e_b)) ∈ env_a f then
      {(NLet x, MCll, top_program_id e_b),
        (NResult (result_name e_b), MRtn, top_program_id e')} ⊆ prog_a
),

    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (App f x_a) e'))
```

For the liveness of channel analysis, it is necessary to track any name that binds to structure containing a a channel of interest, since any use of that name could indicate a use of the channel. In the case where the tracked name might be a function abstraction, the channel simply needs to be live in the body of the abstraction, for the name to be considered built on it. An entry function contains the names built on the channel that might be live at the entrance of each program ID.

```
predicate static_built_on_chan of
  static_value_map -> program_id_map -> name -> name -> bool:
  only

  (∀ x_c env_a x entr .
    if
      SChn x_c ∈ env_a x
    then
      static_built_on_chan env_a entr x_c x),
```

```
(∀ x_sc x_m env_a x entr x_c .
  if
    (SPrm (SendEvt x_sc x_m)) ∈ env_a x,
    (static_built_on_chan env_a entr x_c x_sc or
      static_built_on_chan env_a entr x_c x_m)
  then
    static_built_on_chan env_a entr x_c x),

(∀ x_rc env_a x entr x_c .
  if
    (SPrm (RecvEvt x_rc)) ∈ env_a x,
    static_built_on_chan env_a entr x_c x_rc
  then
    static_built_on_chan env_a entr x_c x),

(∀ x1 x2 env_a x entr x_c .
  if
    (SPrm (Pair x1 x2)) ∈ env_a x,
    (static_built_on_chan env_a entr x_c x1 or
      static_built_on_chan env_a entr x_c x2)
  then
    static_built_on_chan env_a entr x_c x),

(∀ x_a env_a x entr x_c .
  if
    (SPrm (Lft x_a)) ∈ env_a x,
    static_built_on_chan env_a entr x_c x_a
  then
    static_built_on_chan env_a entr x_c x),

(∀ x_a env_a x entr x_c .
  if
    (SPrm (Rght x_a)) ∈ env_a x,
    static_built_on_chan env_a entr x_c x_a
  then
    static_built_on_chan env_a entr x_c x),

(∀ f x_p e_b .
  if
    (SPrm (Abs f x_p e_b)) ∈ env_a x,
    not ((entr (top_program_id e_b) - {x_p}) = {})
  then
    static_built_on_chan env_a entr x_c x)
```

The static liveness of a channel is described in terms of entry functions and exit functions. The entry function maps a program point to a set of names built on the given channel, if those names are live at the entry of that program ID. The exit function maps a program point to a set of names built on the given channel, if those names are live at the exit of that program ID.

```
predicate static_live_chan of
  static_value_map -> program_id_map -> program_id_map -> name -> program
   -> bool:
  only

  (∀ env_a entr x_c y exit .
    if
      (if (static_built_on_chan env_a entr x_c y) then
        {y} ⊆ entr (NResult y))
    then
      static_live_chan env_a entr exit x_c (Rslt y)),

  (∀ exit x entr e' env_a x_c .
    if
      (exit (NLet x) - {x}) ⊆ entr (NLet x),
      entr (top_program_id e') ⊆ exit (NLet x),
      static_live_chan env_a entr exit x_c e'
    then
      static_live_chan env_a entr exit x_c (Let x Unt e')),
  (∀ exit x entr e' env_a x_c .
    if
      (exit (NLet x) - {x}) ⊆ entr (NLet x),
      entr (top_program_id e') ⊆ exit (NLet x),
      static_live_chan env_a entr exit x_c e'
    then
      static_live_chan env_a entr exit x_c (Let x MkChn e')),

  (∀ exit x entr env_a x_c x_sc x_m e' x_c .
    if
      (exit (NLet x) - {x}) ⊆ entr (NLet x),
      (if static_built_on_chan env_a entr x_c x_sc then
        {x_sc} ⊆ entr (NLet x)),
      (if static_built_on_chan env_a entr x_c x_m then
        {x_m} ⊆ entr (NLet x)),
      entr (top_program_id e') ⊆ exit (NLet x),
      static_live_chan env_a entr exit x_c e'
    then
      static_live_chan env_a entr exit x_c
        (Let x (Prim (SendEvt x_sc x_m)) e')),

  (∀ exit x entr env_a x_c x_r .
    if
      (exit (NLet x) - {x}) ⊆ entr (NLet x),
      (if static_built_on_chan env_a entr x_c x_r then
        {x_r} ⊆ entr (NLet x)),
      entr (top_program_id e') ⊆ exit (NLet x),
      static_live_chan env_a entr exit x_c e'
    then
      static_live_chan env_a entr exit x_c
        (Let x (Prim (RecvEvt x_rc)) e')),
```

```
(∀ exit x entr env_a e_c x1 x2 e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (if static_built_on_chan env_a entr x_c x1 then
      {x_1} ⊆ entr (NLet x)),
    (if static_built_on_chan env_a entr x_c x2 then
      {x2} ⊆ entr (NLet x)),
    entr (top_program_id e') ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c (Let x (Prim (Pair x1 x2)) e')
),

(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (if static_built_on_chan env_a entr x_c x_a then
      {x_a} ⊆ entr (NLet x)),
    entr (top_program_id e') ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c (Let x (Prim (Lft x_a)) e')),

(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (if static_built_on_chan env_a entr x_c x_a then
      {x_a} ⊆ entr (NLet x))
    entr (top_program_id e) ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e
  then
    static_live_chan env_a entr exit x_c (Let x (Prim (Rght x_a)) e)),
(∀  exit x entr e_b x_p x env_a x_c e' f .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (entr (top_program_id e_b) - {x_p}) ⊆ entr (NLet x),
    static_live_chan env_a entr exit x_c e_b,
    entr (top_program_id e') ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c
      (Let x (Prim (Abs f x_p e_b)) e')),

(∀ exit x entr e' e_c x_c env_a .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    entr (top_program_id e') ⊆ exit (NLet x),
    entr (top_program_id e_c) ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e_c,
    static_live_chan env_a entr exit x_c e'
```

```
  then
    static_live_chan env_a entr exit x_c
      (Let x (Spwn e_c) e')),

(∀ exit x entr env_a x_c x_e e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (if static_built_on_chan env_a entr x_c x_e then
      {x_e} ⊆ entr (NLet x)),
    entr (top_program_id e') ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e',
  then
    static_live_chan env_a entr exit x_c
      (Let x (Sync x_e) e')),
(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (if static_built_on_chan env_a entr x_c x_a then
      {x_a} ⊆ entr (NLet x)),
    entr (top_program_id e') ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c (Let x (Fst x_a) e')),

(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (if static_built_on_chan env_a entr x_c x_a then
      {x_a} ⊆ entr (NLet x)),
    entr (top_program_id e') ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c
      (Let x (Snd x_a) e')),

(∀ exit x entr e_l x_l e_r x_r env_a x_c x_s e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x),
    (entr (top_program_id e_l) - {x_l}) ⊆ entr (NLet x),
    (entr (top_program_id e_r) - {x_r}) ⊆ entr (NLet x),
    (if static_built_on_chan env_a entr x_c x_s then
      {x_s} ⊆ entr (NLet x)),
    static_live_chan env_a entr exit x_c e_l,
    static_live_chan env_a entr exit x_c e_r,
    entr (top_program_id e') ⊆ exit (NLet x),
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c (Let x (Case x_s x_l e_l x_r
e_r) e')),
```

```
  (∀ exit x entr env_a x_c x_a f e' .
    if
      (exit (NLet x) - {x}) ⊆ entr (NLet x),
      (if static_built_on_chan env_a entr x_c x_a then
        {x_a} ⊆ entr (NLet x)),
      (if static_built_on_chan env_a entr x_c f then
        {f} ⊆ entr (NLet x)),
      entr (top_program_id e') ⊆ exit (NLet x),
      static_live_chan env_a entr exit x_c e'
    then
      static_live_chan env_a entr exit x_c (Let x (App f x_a) e'))
```

The static liveness of a transition is described by checking if a transition exists in a whole static program, and if it meets certain criteria with respect to the entry and exit liveness functions.

```
predicate static_live_transition of
  static_program -> program_id_map -> program_id_map -> transition ->
  bool:
  only

  (∀ l l' prog_a exit entr .
    if
      (l, MNxt, l') ∈ prog_a,
      not (exit l = {}),
      not (entr l' = {})
    then
      static_live_transition prog_a entr exit (l, MNxt, l')),

  (∀ l l' prog_a exit entr .
    if
      (l, MSpwn, l') ∈ prog_a,
      not (exit l = {}),
      not (entr l' = {})
    then
      static_live_transition prog_a entr exit (l, MSpwn, l')),

  (∀ l l' prog_a exit entr .
    if
      (l, MCll, l') ∈ prog_a,
      (not (exit l = {})) or (not (entr l' = {}))
    then
      static_live_transition prog_a entr exit (l, MCll, l')),

  (∀ l l' prog_a entr exit .
    if
      (l, MRtn, l') ∈ prog_a,
      not (entr l' = {})
    then
      static_live_transition prog_a entr exit (l, MRtn, l')),
```

```
  (∀ x_send x_evt x_recv prog_a entr exit .
    if
      ((NLet x_send), ESend x_evt, (NLet x_recv)) ∈ prog_a,
      {x_evt} ⊆ (entr (NLet x_send))
    then
      static_live_transition prog_a entr exit
        ((NLet x_send), ESend x_evt, (NLet x_recv)))
```

The static traceability for the higher precision analysis states that an entire static path can be trace through some static program and is live with respect to some entry and and exit functions.

```
predicate static_traceable of
  static_value_map -> static_program -> program_id_map -> program_id_map
  ->
  program_id -> (program_id -> bool) -> static_path -> bool:
  only

  (∀ is_end start env_a prog_a entr exit .
    if
      is_end start
    then
      static_traceable prog_a entr exit start is_end []),

  (∀ prog_a entr exit start middle path is_end mode.
    if
      static_traceable prog_a entr exit start (λ l . l = middle) path,
      (is_end end),
      static_live_transition prog_a entr exit (middle, mode, end)
    then
      static_traceable prog_a entr exit start is_end (path @ [(middle,
  mode)])
```

As with the lower precision analysis, the higher precision analysis relies on recognizing whether or not two paths can actually occur within in a single run of a program. The static inclusiveness states which paths might occur within the same run of the program. In contrast to the analogous definition for the lower precision alysis, the higher precision definition needs to consider paths containing the sending mode. As mentioned earlier, the path from the synchronization on sending to the synchronization on receiving is necessary to ensure that all uses of a channel are reachable from the channel's creation ID. The singularness means that only one of the two given paths can occur in a run of program. The noncompetitveness means that the two given paths do not compete in any run of a program.

```
predicate static_inclusive of
  static_path -> static_path -> bool:
  only
```

```
  (∀ path1 path2 .
    if
      prefix path1 path2 or path2 path1
    then
      static_inclusive path1 path2),

  (∀ path x path1 path2 .
    static_inclusive
      (path @ (NLet x, MSpwn) # path1)
      (path @ (NLet x, MNxt) # path2)),

  (∀ path x path1 path2 .
    static_inclusive
      (path @ (NLet x, MNxt) # path1) (
      path @ (NLet x, MSpwn) # path2)),

  (∀ path x path1 path2 .
    static_inclusive
      (path @ (NLet x, ESend xE) # path1)
      (path @ (NLet x, MNxt) # path2)),

  (∀ path x path1 path2 .
    static_inclusive
      (path @ (NLet x, MNxt) # path1)
      (path @ (NLet x, ESend xE) # path2))

predicate singular of static_path -> static_path -> bool:
  only

  (∀ path .
      singular path path),

  (∀ path1 path2 .
    if
      not (static_inclusive path1 path2)
    then
      singular path1 path2)

predicate noncompetitive of static_path -> static_path -> bool:
  only

  (∀ path1 path2 .
    if
      ordered path1 path2
    then
      noncompetitive path1 path2),

  (∀ path1 path2 .
    if
      not (static_inclusive path1 path2)
```

```
    then
      noncompetitive path1 path2)
```

The communication topology classifications are described using the liveness properties, but are otherwise similar to the lower precision classifications.

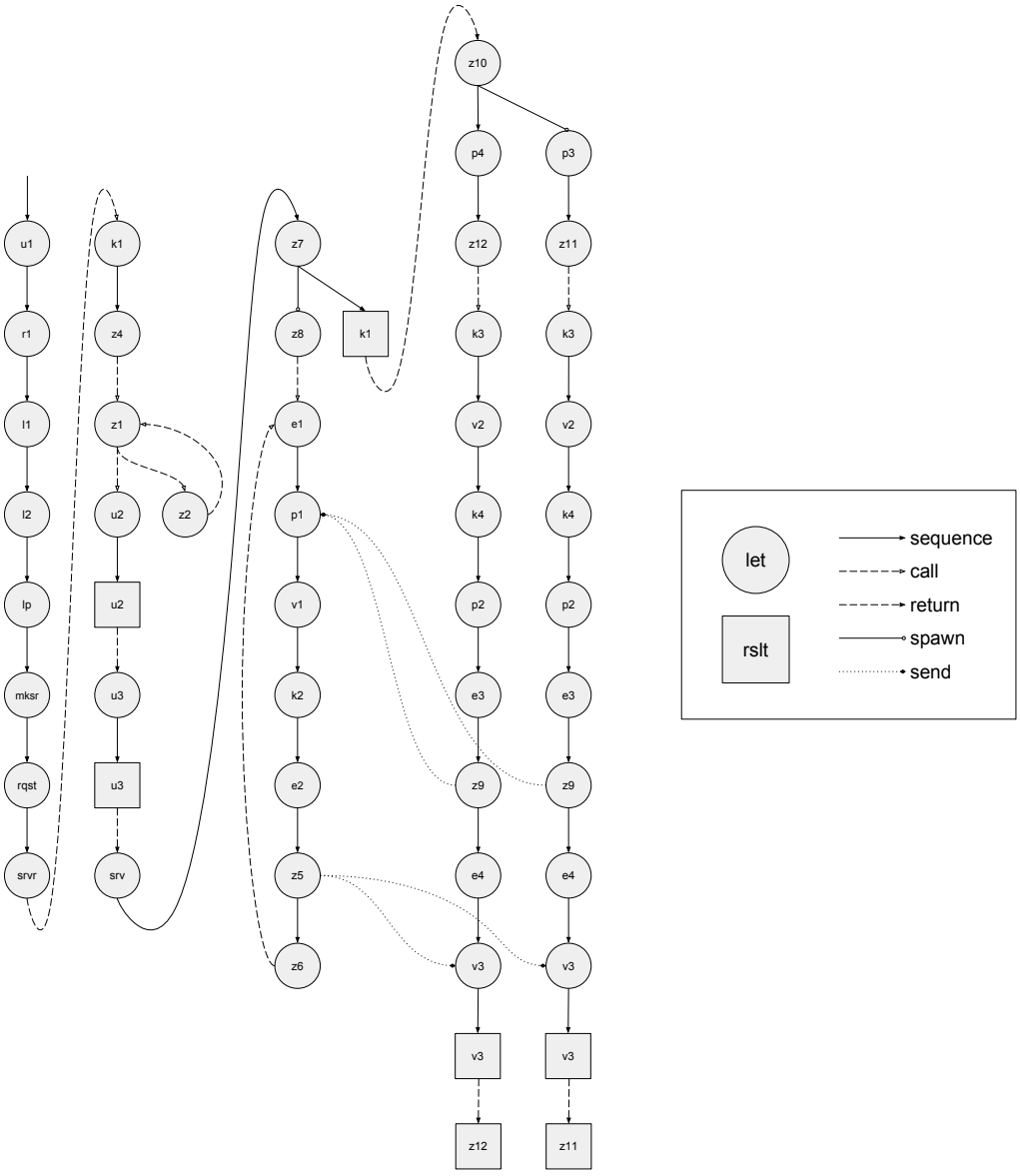
```
predicate static_one_shot of static_value_map -> program -> name -> bool:
  only
  (∀ prog_a entr exit x_c env_a e .
    if
      every_two
        (static_traceable prog_a entr exit
          (NLet x_c) (static_send_id env_a e x_c))
        singular,
      static_live_chan env_a entr exit x_c e,
      static_traversable env_a prog_a e
    then
      static_one_shot V e x_c)

predicate static_one_to_one of static_value_map -> program -> name ->
  bool:
  only
  (∀ prog_a entr exit x_c env_a e .
    if
      every_two
        (static_traceable prog_a entr exit (NLet x_c) (static_send_id
  env_a e x_c))
        noncompetitive,
      every_two
        (static_traceable prog_a entr exit (NLet x_c) (
  static_recv_program_id env_a e x_c))
        noncompetitive,
        static_live_chan env_a entr exit x_c e,
        static_traversable env_a prog_a e
    then
      static_one_to_one env_a e x_c)

predicate static_one_to_many of static_value_map -> program -> name ->
  bool:
  only
  (∀ prog_a entr exit x_c env_a e .
    if
      every_two
        (static_traceable prog_a entr exit (NLet x_c) (static_send_id
  env_a e x_c))
        noncompetitive,
      static_live_chan env_a entr exit x_c e,
      static_traversable env_a prog_a e
    then
      static_one_to_many env_a e x_c)
```

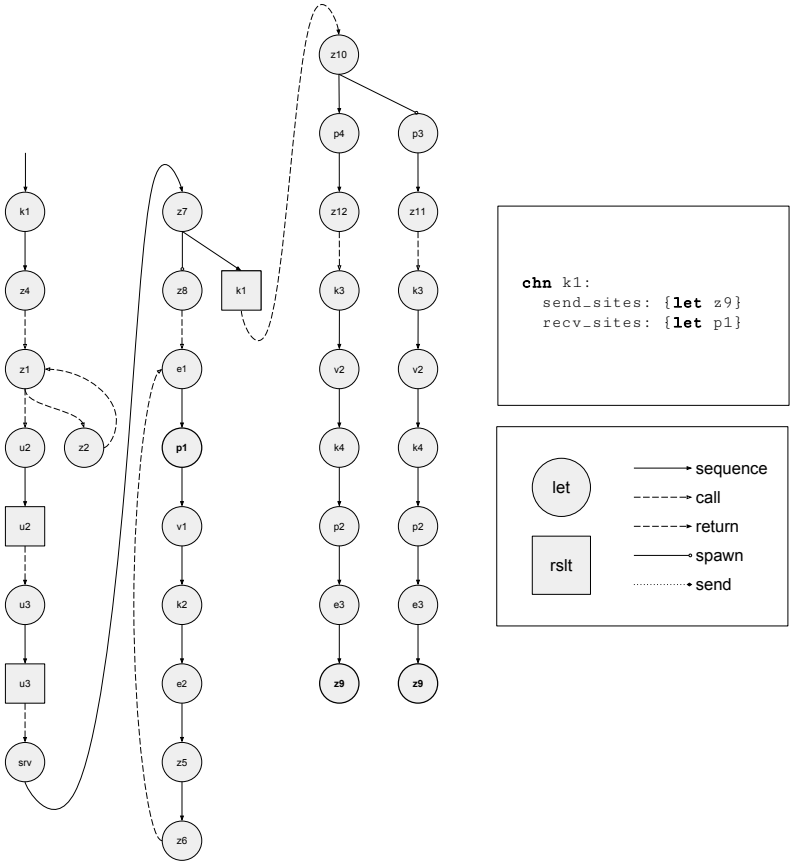
```
predicate static_many_to_one of static_value_map -> program -> name ->
  bool:
  only
  (∀ prog_a entr exit x_c env_a e .
    if
      every_two
        (static_traceable prog_a entr exit (NLet x_c) (
  static_recv_program_id env_a e x_c))
        noncompetitive,
      static_live_chan env_a entr exit x_c e,
      static_traversable env_a prog_a e
    then
      static_many_to_one env_a e x_c)
```

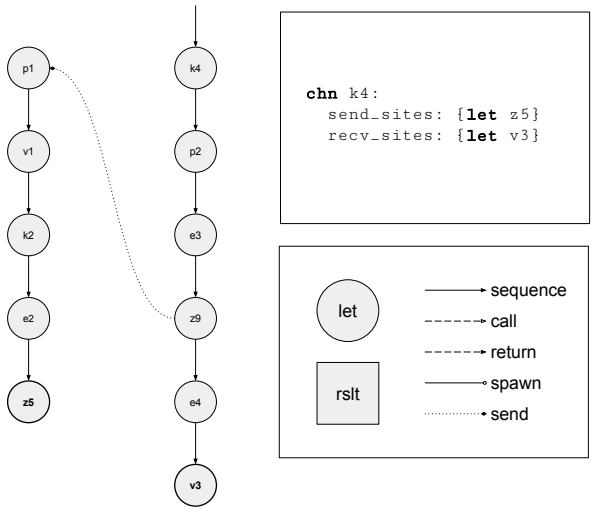
A slightly modified version of the server implementation demonstrates the usage of channel liveness analysis, and liveness traversabilitiy of programs. An additional loop has been added to the server implementation. The loop basically just wastes time, but it is used to demontrate how liveness analysis treats function abstractions that not contain any channel of interest. No channel is considered to be live in the body of the loop. However, a channel may be live before the loop is called and after the loop returns. In such a case, the live traceable path from the creation of a channel to a point after the loop retains the transitions within the loop, even though the points in the loop may not be live according to the entry map.

```
let u1 = unt in
let r1 = rght u1 in
let l1 = lft r1 in
let l2 = lft l1 in

let lp = (fun lp' x1 =>
  let z1 = case x1 of
    lft y1 => let z2 = lp' y1 in z2 |
    rght y2 => let u2 = unt in rslt u2 in
  let u3 = unt in
  rslt u3) in

let mksr = (fun _ x2 =>
  let k1 = mk_chn in
  let z4 = lp l2 in
  let srv = fun srv' x3 =>
    let e1 = recv_evt k1 in
    let p1 = sync e1 in
    let v1 = fst p1 in
    let k2 = snd p1 in
    let e2 = send_evt k2 x3 in
    let z5 = sync e2 in
    let z6 = srv' v1 in
    let u4 = unt in
```

```
    rslt u4 in
  let z7 = spawn (
    let z8 = srv r1 in
    let u5 = unt in
    rslt u5) in
  rslt k1) in

let rqst = (fun _ x4 =>
  let k3 = fst x4 in
  let v2 = snd x4 in
  let k4 = mk_chn in
  let p2 = pair v2 k4 in
  let e3 = send_evt k3 p2 in
  let z9 = sync e3 in
  let e4 = recv_evt k4 in
  let v3 = sync e4 in
  rslt v3) in

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


u1
r1
l1
l2
lp
mksr
rqst
srvr
k1
z4
z1
u2
z2
u2
u3
u3
srv
z7
z8
k1
e1
p1
v1
k2
e2
z5
z6
z10
p4
p3
z12
z11
k3
k3
v2
v2
k4
k4
p2
p2
e3
e3
z9
z9
e4
e4
v3
v3
v3
v3
z12
z11
let
rslt
sequence
call
return
spawn
send


k1
z4
z1
u2
z2
u2
u3
u3
srv
z7
z8
k1
e1
p1
v1
k2
e2
z5
z6
z10
p4
p3
z12
z11
k3
k3
v2
v2
k4
k4
p2
p2
e3
e3
z9
z9
chn k1:
send_sites: {let z9}
recv_sites: {let p1}
let
rslt
sequence
call
return
spawn
send


p1
v1
k2
e2
z5
k4
p2
e3
z9
e4
v3
chn k4:
send_sites: {let z5}
recv_sites: {let v3}
let
rslt
sequence
call
return
spawn
send

16 Higher Precision Reasoning Strategy

To prove soundness of the communication topology classification, it should be possible to use previous techniques of generalizing propositions over pools and other semantic components, along with finding equivalent representations of propositions that vary in the inductive subcomponent. One thing that will make carrying out the formal proof particularly tricky is that dynamic paths in the dynamic semantics need to corresponding to static paths from the trimmed static programs, which might also contain sending transitions, instead of the dynamic paths spawning transitions. The correspondence between these dynamic paths and static paths is not bijective, as it is for the lower precision analysis. However, finding a satisfactory correspondence for each dynamic and static path is critical for proving soundness.

Essentially, it will be necessary to show that static properties that hold for some static path are related in some fashion to dyamic properties that hold for a dynamic path which corresponds to the static path modulo the channel under analysis. These reasoning strategies are demonstrated by the theorem for soundness of one-shot classification.

```
theorem static_one_shot_sound:
  ∀ e0 pool convo env_a convo_a x_c path_x .
    if
      star dynamic_eval [[] -> (Stt e0 [->] []] {} pool convo,
      static_eval env_a convo_a e0,
      static_one_shot env_a e0 x_c
    then
      one_shot pool (Chan path_c x_c)
```

The theorem for soundness of one-shot classification depends on correlating dynamic paths with static paths.

```
predicate paths_correspond of dynamic_path -> static_path -> bool:
  only

  paths_correspond [] [],

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DNxt x])
        (path_a @ [(NLet x, MNxt)])),

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DSpwn x])
```

```
      (path_a @ [(NLet x, MSpwn)])),

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DCll x])
        (path_a @ [(NLet x, MCll)])),

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DRtn x])
        (path_a @ [(NResult x, MRtn)]))

predicate paths_correspond_mod_chan of
  pool -> conversation -> channel ->
  dynamic_path -> static_path -> bool:
  only

  (∀ pool path_c x_c path_sfx stt path_a convo .
    if
      pool (path_c @ (DNxt x_c) # path_sfx) = Some stt,
      paths_correspond ((DNxt x_c) # path_sfx) path_a
    then
      paths_correspond_mod_chan
        (pool, convo) (Chan path_c x_c)
        (path_c @ (DNxt x_c) # path_sfx) path_a),

  (∀ pool path_r x_r path_sfx stt  path_s x_s x_se e_sy env_sy stack_sy
    x_re e_ry env_ry stack_ry c_c convo c path_a_re path_a_sfx .
    if
      pool (path_r @ (DNxt x_r) # path_sfx) = Some stt,
      pool path_s = Some (Stt (Let x_s (Sync x_se) e_sy) env_sy stack_sy)
  ,
      pool path_r = Some (Stt (Let x_r (Sync x_re) e_ry) env_ry stack_ry)
  ,
      {(path_s, c_c, path_r)} ⊆ convo,
      dynamic_built_on_chan_var env_ry c x_r,
      paths_correspond_mod_chan pool convo c path_s path_a_pre,
      paths_correspond path_sfx path_a_sfx
    then
      paths_correspond_mod_chan pool convo c
        (path_r @ (DNxt x_r) # path_sfx)
        (path_a_pre @ (NLet x_s, ESend x_se) # (NLet x_r, MNxt) #
  path_a_sfx))
```

Additionally the soundness theorem follows from the soundness of lack of static traceability, the soundness of lack of static inclusiveness, and the soundness of a program ID not being a sending ID. The reasoning about the sending ID is identical to that of the lower precision analysis, but the the reasoning for the former two is significantly more complicated and not yet completed. The complication arises from the correlation between dynamic paths and static paths. The proofs depend on finding a static path that depends on a given dynamic path. In the lower precision analysis the correlation was straightforward. There was only one possible static path to choose for it to correlate with the given dynamic path. In the higher precision analysis, the relationship between the two kinds of paths is not so simple, and finding a description of the static path that correlates with the dynamic path is much more challenging.

```
predicate paths_correspond of dynamic_path -> static_path -> bool:
  only

  paths_correspond [] [],

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DNxt x])
        (path_a @ [(NLet x, MNxt)])),

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DSpwn x])
        (path_a @ [(NLet x, MSpwn)])),

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DCll x])
        (path_a @ [(NLet x, MCll)])),

  (∀ path path_a x .
    if
      paths_correspond path path_a
    then
      paths_correspond
        (path @ [DRtn x])
        (path_a @ [(NResult x, MRtn)]))
```

```
predicate paths_correspond_mod_chan of
  pool -> conversation -> channel ->
  dynamic_path -> static_path -> bool:
  only
  (∀ pool path_c x_c path_sfx stt path_a convo .
    if
      pool (path_c @ (DNxt x_c) # path_sfx) = Some stt,
      paths_correspond ((DNxt x_c) # path_sfx) path_a
    then
      paths_correspond_mod_chan
        (pool, convo) (Chan path_c x_c)
        (path_c @ (DNxt x_c) # path_sfx) path_a
    ),
  (∀ pool path_r x_r path_sfx stt  path_s x_s x_se e_sy env_sy stack_sy
    x_re e_ry env_ry stack_ry c_c convo c path_a_re path_a_sfx .
    if
      pool (path_r @ (DNxt x_r) # path_sfx) = Some stt,
      pool path_s = Some (Stt (Let x_s (Sync x_se) e_sy) env_sy stack_sy)
  ,
      pool path_r = Some (Stt (Let x_r (Sync x_re) e_ry) env_ry stack_ry)
  ,
      {(path_s, c_c, path_r)} ⊆ convo,
      dynamic_built_on_chan_var env_ry c x_r,
      paths_correspond_mod_chan pool convo c path_s path_a_pre,
      paths_correspond path_sfx path_a_sfx
    then
      paths_correspond_mod_chan pool convo c
        (path_r @ (DNxt x_r) # path_sfx)
        (path_a_pre @ (NLet x_s, ESend x_se) # (NLet x_r, MNxt) #
  path_a_sfx)
    )

lemma not_static_traceable_sound:
  ∀ e0 pool convo path x b e' env stack env_a convo_a
    entr exit x_c prog_a is_end path_c  .
    if
      star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo,
      pool path = Some (Stt (Let x b e') env stack),
      static_eval env_a convo_a e0,
      static_live_chan env_a entr exit x_c e0,
      static_traversable env_a prog_a e0,
      is_end (NLet x)
    then
      (∃ path_a .
        paths_correspond_mod_chan pool convo (Chan path_c x_c) path
  path_a,
        static_traceable prog_a entr exit (NLet x_c) is_end path_a)

lemma not_static_inclusive_sound:
  ∀ e0 pool convo env_a entr exit x_c prog_a convo_a
```

```
  path1 stt1 path_c path_a1 path2 stt2 path_a2 .
  if
    star dynamic_eval [[] -> (Stt e0 [->] [])] {} pool convo,
    static_live_chan env_a entr exit x_c e0,
    static_traversable env_a prog_a e0,
    static_eval env_a convo_a e0,
    pool path1 = Some stt1,
    paths_correspond_mod_chan pool convo (Chan path_c x_c) path1
path_a1,
    static_traceable prog_a entr exit
      (NLet x_c) (static_send_id env_a e0 x_c) path_a1,
    pool path2 = Some stt2,
    paths_correspond_mod_chan pool convo (Chan path_c x_c) path2
path_a2,
    static_traceable prog_a entr exit
      (NLet x_c) (static_send_id env_a e0 x_c) path_a2
  then
    static_inclusive path_a1 path_a2
```