Formal Theory of Concurrent ML

Thomas Logan

October 15, 2018

1 Summary

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 simplied 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 topologigies 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 mechannically 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 philsophocial equivalence is hopefully apparent. Thus, recasting Reppy and Xiao's work was far more nuanced than a straight forward 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 nonequational constraints [**?**].

2 Overview

In programming languages, concurreny is a program structuring technique that allows evaluation steps to hop back and forth between disjoint syntactc 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 structrues 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 elengant concurrent programming 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 Programming 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 reqChn = channel ()
  fun loop state = let
    val (v, replCh) = recv reqChn in
    send (replCh, state);
    loop v end in
  spawn (fn () => loop 0);
  S reqChn end

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

end
```

Concurrent ML actually allows for events other than sending and receiving to occur during synchronization. In fact, the syncronization 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 syncronization 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 constiuent 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 `then` 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 `then` combinator, rather than duplicating code with the addition of stronger isolation. When the event constructed by the `then` 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 then : 'a event * ('a -> 'b event) -> 'b event
```

—-Edit From Here—-

3 Syncrhonization Implementation

A uniprocessor implementation of synchronous communication is inexpensive. Using a fairly course-grain interleaving, the communication on a channel can proceed by checking if the channel is in one of two possible states: either a corresponding thread is waiting or there's nothing waiting. The implementation doesn't need to consider states where competing threads are also trying to communicate on the same channel, since the course-grain interleaving ensures that competing threads have made no partial communication progress. In a multiprocessor setting, threads can run in parallel and multiple threads can simultaneously make partial progress on the same channel. The multiprocessor implementation of communication is more expensive than that of the uniprocessor, since it must consider additional states related to competing threads making partial communication progress.[10]

Channels known to have only one sender or one receiver can have lower communication costs than those with arbitrary number of senders and arbitrary number of receivers, since some of the cost of handling competing threads can be avoided. Concurrent ML does not provide language features for multiple types of channels distinguished by their communication topologies, or the number of threads that may end up sending or receiving on it. However, channels can be classified into various topologies based on their potential communication. A many-to-many channel has any number of senders and receivers; a fan-out channel has one sender and any number of receivers; a fan-in channel has any number of senders and exactly one receiver; a one-to-one channel has exactly one of each; a one-shot channel has exactly one sender, one receiver, and sends data only once. The server implementation in Listing 2 with the following calling code exhibits these topologies.

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 fan-in. 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 reqChn = FanIn.channel()

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

    spawn (fn () => loop 0);
    S reqChn end

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

  end
```

A program 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 restricted channels, it can conservatively verify the correct usage of restricted channels. Listing 2 demonstrates the language extension based on an example from Reppy and Xiao[11].

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, resulting in runtime behavior inconsistent with the general semantics of channel synchronization.

The utility of the program analysis additionally depends on it being informative, sound, and computable. The analysis is informative 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 operational 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.

Program analyses, like operational semantics, describe information about the execution or behavior of programs. Yet, while an operational semantics may be viewed as ground truth, the correctness of an analysis is derived from its relation to an operational semantics. In practice, program analyses often describe computable information with respect to operational semantics that are universal and capable of describing

uncomputable information. To allow for computability, program analyses often describe approximate information.

There are a large number of program analyses with a variety of practical uses. Some constructions of programs might be considered bad, by describing operations that don't make sense, like True * 5 / "hello", or accessing the 7th element of an array with 6 elements. A type systems, or static semantics, is an analysis that can help ensure programs are well constructed. It describes how programs and expressions can be composed, such that the programs won't get stuck or result in certain kinds of undesired behavior. 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 some runtime checks unnecessary.

Other analyses are useful for describing opportunities for program optimizations. Many analyses used for optimizations describe how data flows with information related to every point in the program. Each point refers to a term, from which the small-step semantics may take a step. Some programs may mention the same expression multiple times, possibly resulting in redundant computations. These redundant computations can be detected by available expressions analysis, one of many data flow analyses. An available expressions analysis describes which expressions must have been computed by each program point.

The expression (!x + 1) is available by line 9 but (!y + 2) and (w - 3) are not, because y was modified in line 8 and w was rebound in line 6. Another inefficiency is that programs may perform computations, but then ignore their results. Such dead code can be detected by a liveness analysis. The analysis describes for each program point, the set of variables and references whose values might be used in the remainder of the program.

Since the variables x and z and the dereference !z are used in line 8, they are live at line 7. Since z is reassigned at line 7, !z is no longer live at line 6. Since x is bound at line 5 and not used above, it is not live at line 4 and above. Since z is bound at 4 and not used above, it is not live at line 3 and above. The liveness information demonstrates that the expression (4 * 73) doesn't need to be computed, and lines 2 and 3 can simply be removed.

The information at each program point is derived from control structures in the program that dictate how information may flow between program points. Some uses of control structures are represented as literals in the syntax, while other uses are expressions that may evaluate to control structures, or function parameters that may bind to control structures. Function abstraction is a control structure allowing multiple parts of a program to flow into a section of code via a binding. In ML, function abstractions are higher order, and may be unknown without some form of evaluation. These control structures may be revealed by an abstract value flow analysis, which associates each program point with a set of abstract values that the point's expression may evaluate to.

The abstract values of f, g, h are simply their let bound expressions fn x => x 1, fn y => y + 2, fn z => z + 3, respectively. x has the abstract values of fn y => y + 2, fn z => z + 3, so x 1 has the abstract values of 3, 4; (f g) has abstract values of 3, 4. Since the abstract values depend on the flow of information, which depends on the abstract values, the description of abstract values is inductive or recursive. The

historical motivation for describing the abstract value information was really for its the control information, so the original approaches to these analyses are known as control flow analyses or CFAs. With the control flow information, other data flow analyses like available expression analysis and liveness analysis can provide greater coverage.

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


let
  val x = 1
  val y = 2
  val z = ref (4 * 73)
  val x = 4 in
  z := 1;
  x * !z end



let
  val f = fn x => x 1
  val g = fn y => y + 2
  val h = fn z => z + 3 in
  (f g) + (f h) end
```

Analyses can be described in a variety of ways. An algorithm that take programs as input and produce behavior information as output are necessary for automation in compilers. A specification that states a proposition in terms of programs and execution information may be more suitable for showing clarity of meaning and correctness with respect to the operational semantics. The specification can be translated into an algorithm involving two parts. The first part generates a comprehensive set of data structures representing constraints of all program points, mirroring the specification's description, and the second part solves the constraints.

For a subseteq of Concurrent ML without event combinators, Reppy and Xiao developed an efficient algorithmic analysis that determines for each channel all abstract 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[12, 13] that results in multiple mappings. One mapping is from variables to abstract values that may bind to the variables. Another mapping is from channel-bound variables to abstract values that are sent on the respective channels. Another is from function-bound variables 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 variable. 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, fan-in, fan-out, or many-to-many.

Reppy and Xiao informally prove soundness of their analysis by showing that their analysis claims that more than one thread sends (or receives) on a channel if the execution allows more than one to send (or receive) on a 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 syntax, semantics, and analysis need to describe many details. Proving propositions relating all of these definitions requires manipulation of all those details. To ensure the correctness of proofs, it is necessary to check that there are no subtle errors in 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 the one in Isabelle, has no difficulty discerning between valid and invalid derivations of statements. Mechanical checking of proofs can notify us of errors in the proofs or definitions far better and faster than manual checking. I have already benefitted from Isabelle's proof checker in order to correctly define the language semantics and abstract value flow analysis for this work. While trying to prove soundness of the analysis, the proof assistant would not accept my proof unless I provided derivation of facts that I believed to be false. I determined that my intuition was correct but my definitions had errors. After correcting the errors, I was able to complete the proof, such that the proof checker was satisfied.

4 Hypothesis

I will derive a static analysis from Reppy and Xiao's algorithm, describing for each channel in a program, all threads that possibly send or receive on the channel. Additionally, it will classify channels as one-shot, one-to-one, fan-out, fan-in, or many-to-many. Instead of Serrano's algorithm[18] for the CFA used in Reppy and Xiao's algorithm, I will define a constraint-based specification and algorithm for the CFA. The method of determining topologies will be fairly similar to Reppy and Xiao's. The analysis of this work will also consider event combinators, which are not considered in Reppy and Xiao's work. I will show that the static analysis is informative by demonstrating programs for which the static analysis classifies some channels as fan-in, fan-out, and so on. I will show that the static analysis is sound by showing that for any program, the execution of the program results in the same sends and receives or fewer compared to the possible sends and receives described by the analysis. I will show that the static analysis is computable by demonstrating the existence of a computable function that takes any program as input and generates all sends and receives described by the analysis.

5 Evaluation

The main contributions of this work will be formal and mechanically verified proofs of communication properties of Concurrent ML, including an analysis derived from Reppy and Xiao's analysis. This work extends that of Reppy and Xiao by demonstrating formal proofs of soundness and extending the analysis to encompass event combinators for choice and transactions.

6 Architecture

To enable mechanical verification of the correctness of the proofs, I will construct the semantics, analysis and theorems in the formal language of Isabelle/HOL. To aid the development of formal proofs, I will design the analysis as a declarative specification as opposed to an algorithm. However, the declarative analysis will make the proof of computability less direct. To aid the scrutiny of the theorems' adequacy, I will express the definitions and propositions with the fewest number of structures, judgements, inferences rules , and axioms necessary. In order to relate the analysis to the operational semantics, I will borrow Reppy and Xiao's strategy of stepping between sets of control paths tied to terms.

In this thesis work, I'm interested in communication topology soundness, rather than flow soundness. Nevertheless, I will need to prove additional flow soundness theorems en route to proving communication topology soundness. Restricting the grammar to a form that requires every abstraction and application to be bound to a variable would allow the operational semantics to maintain static term information necessary for proofs of flow soundness[3, 5]. The semantics would be defined as an environment based operational semantics, rather than a substitution based operational semantics. By avoiding simplification of terms in the operational semantics, it

will be possible to relate the abstract values of the analysis to the values produced by the operational semantics, which in turn is relied on to prove flow soundness.

I will incorporate the restricted grammar and the environment based semantics into this work. The restricted grammar is impractical for a programmer to write, yet it is still practical for a language under automated analysis since there is a straight forward procedure to transform more flexible grammars into the restricted form as demonstrated by Flanagan et al [2]. Additionally, the restricted grammar melds nicely with the control path semantics. Instead of defining additional meta-syntax for program points of primitive operations, I can simply use the required variables of the restricted grammar to identify program points, and control paths will simply be sequences of let bound variables. A modification of Listing 2 illustrates the restrictive grammar applied to Concurrent ML.

7 Implementation

We describe possible implementations of specialized and unspecialized Concurrent ML using 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. Depending on the low level features provided by existing language implementations, Concurrent ML could be implemented in terms of lower level features, as is the case in SML/NJ and MLton. It could also be implemented as primitive features within a compiler and runtime or interpreter. Analyzing and optimizing Concurrent ML would require treating the language as an object, so implementing its features as primitives would make the most sense. Thus, one can think of the implementation shown here as an intermediate representation 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 ins multiprocessor implementations rather than single processor implementations. A single processor implementation could avoid overhead caused by contention to acquire locks, by coupling the implementation of channels with scheduling and only scheduling send and recv 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 low overhead. However, a multiprocessor implementation would allow threads to run on different processors for increased parallelism and would not be able to mandate when threads are attempted relative to others without losing the parallel advantage. The cost of trying to achieve parallelism is increased overhead due to contention over acquiring locks.

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. Additionally a channel is composed of a mutex lock, so that send and recv 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 accessible by receiving threads and conditions waited on by the receiving threads. The three states are represented by the datatype chan_content. The channel is represented by the channel datatype, which is composed of a reference to chan_content and a mutex lock.

The send 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 send 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 send 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 send operation returns with the unit value.

The recv 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 recv 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 recv 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 recv 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 fan-out 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 send operation checks if the channel's state is inactive and tries to use the compareAndSwap 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 send 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 send operation, then there might be data races. Likewise, due to the expectation of a single thread sending on the channel, the send operation should never witness the state of threads already trying to send.

The recv operation acquires the lock and checks the state of the channel, just as the recv 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 recv 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 fan-in channels is very similar to that of fan-out 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 send operation checks if the channel's state is inactive and tries to use the compareAndSwap 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 send operation should never witness the state of another thread already trying to send, if it is truly one-to-one.

The recv operation checks if the channel's state is inactive and tries to use the compareAndSwap 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 recv 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 send 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 recv operation checks if the channel's state is inactive and tries to use the compareAndSwap 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 relases 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 send operation deposits the message in the cell, signals on the recvCond, waits on the sendCond, and then returns the unit value. The recv 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
```

Although there are proofs that the communication topologies are sound with respect to the semantics, it would additionally 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.

8 Objectives

If an algorithm for synchronization is specific to a maxiumum number of threads, it may be more efficient than an algorithm that is generic for any number of threads. For instance, if there is only ever one thread sending and one thread receiving on a channel, then no locks are needed, which saves time. However, if there are multiple senders and multiple receivers, then some form of locks or trials and aborts would be needed, which is costly.

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 sender and one receiver, then you will lower synchronization costs by using an implemnetaiton 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 implementaiton, 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 oneo-to-one communication topology, and that the relation between the procedure's input program and output topology upperbound 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 programming language.

In this work I demonstostrate formal proofs that the a relation between programs and topologies is sound with respect to the semantics of the programming language. I refer to this as a static relation, because the intent is that the topologies are computable from the programs, although I do not formally prove computability in this work. The static relation is defined with a syntax directed structure, which gives strong evidence of computability. Additionally, I do not formally prove that the specialized implementations are behaviorally equivalent to a generic implementation, but I suggest some plausible example implementations.

9 Syntax Definitions

The relations are defined over a simple language featuring a small subseteq of Concurrent ML features. The features include recursive function abstraction with application, left and right construction with pattern matching, pair construction with first and second projections, send and receive event construction with synchronization, channel creation, thread spawning, and unit literal. The syntax and semantics are defined in a peculiar way to enable relations between static and dynamic properties of the language. A primititive construct is one that contains references to names but cannot be evaulated any further. Send and receive events, left and right constructs, pairs, and function abstractions are primitive constructs.

The syntax is in a very restrictive administrative normal form (ANF), in which ev-

ery value or evaluatable construct is bound to a name. Furthermore, constructs only accept names for eagerly evaluted inputs, rather than expression. A control path is basically defined as a list of bound names, where the names are in order of their respective constructs' evaluation. The names are also annoted with information related to aspects of the control flow.

```
datatype name = Name 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 var |
    Lft name | Rght name |
    Abs name var exp
```

The unit literal corresponds to a unit value. A dynamic channel value is uniquely identified by the full control path up to the point of channel creation along with the name bound to the channel creation construct. The value of a primitive construct is a closure over the construct with an environment that maps the names to further values.

10 Dynamic Semantics Definitions

Bound expressions of the unit literal, primitive constructs, and first and second constructs can be evalued to values in one step. The seq_eval relation describes the evaluation of bound expressions with environments to values. The unit construct simply evaluates to the unit value. A primitive construct is simply wrapped up with the environment. The first projection is evaluated to the abstract value of the first argument of the associated pair. The second projection is evaluated to the abstract value of the second argument of the associated pair.

Other constructs for spawn, application, and case matching cannot be described with the seq_eval because they do not evaluate to values in one step and require

updating additional information about the mode of the transition.

```
datatype program_point =
  LNxt name | LSpwn name | LCll name | LRtn name

type program_path = program_point list

datatype channel =
  Chan program_path name

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

type environment =
  name -> program_value option

predicate seq_eval of expression -> environment -> program_value -> bool:
only
(∀ env .
    seq_eval Unt env VUnt) and
(∀ p env .
    seq_eval (Prim p) env (VPrm p env)) and
(∀ env x_p x1 x2 env_p v.
  if
    env x_p = Some (VPrm (Pair x1 x2) env_p) and
    env_p x1 = Some v
  then
    seq_eval (Fst x_p) env v) and
(∀ env x_p x1 x2 env_p v .
  if
    env x_p = Some (VPrm (Pair x1 x2) env_p) and
    env_p x2 = Some v
  then
    seq_eval (Snd x_p) env v)
```

The seq_eval up relation handles the evaluation of function application and case pattern matching. These bound expressions with associated environments are evaluated to full expressions along with a new environment. Case matching is evaluated based on the case of the construct's first argument. If the first argument is a left case, then the case matching is evauluated to the left expression along with an environment updated with a name specified for use in that expression. The value associated with the left case pattern is bound to the specified name in the new environment. If the first argument is a right case, then then the case matching is evaluated to the right expression along with an environment updated with a name specified for use in that expression. The value associated with the right case is bound to the specified name in the new environment. Function application evaluates to the expression within the applied function abstraction, along with the function abstraction's environment with a couple of modifications. The environment is updated with the recursive function name bound to the function abstraction, and the parameter name is bound to the function argument specified in the application expression.

```
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) and
    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))) and

(∀ 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))) and

(∀ 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)))
```

A continuation is composed of a name, an expression and an environment. It represents an expression and environment that should be evaluated once the name can be resolved to a value. A state is composed of an expression, an evnironment, and a stack of continuations. It represents information that might be evaluted to another state in the present of control path information. A trace pool associated control paths with states. The associated control path represent the path taken to reach the associated state. The leaf predicate desribes that a path is has no descendants within a trace pool.

The concurent_eval relation describes the evaluation of a trace pool to another tracel pool. Trace pools grow montonicially with respect to evaluation, retaining a full history. A trace pool is evaluated to a new trace pool based only on information associated with leaf paths and their associated states. If a leaf path is associated with a bound expresion that can be evaluated to a value in one sequential step, then the trace pool is updated with the leaf path extended by a label indicating a sequential transition, and the extended path is associated with the let binding's next expression and the leaf path's associated environment updated with the value bound to the let binding's name. If the bound expression can be evaluated to an expression and a new environment, then the leaf path is extended with a label indicating a calling transition. The extended path is associated with the the new expression and new environment resulting from the evaluation of the bound expression, and the name and expression of the let binding are pushed onto the stack to be evaluated in later steps. If a leaf path is associated with a result expression, then the trace pool is updated with the leaf path extended with a label indicating a returning transition. The new path is

associated with a state containing the expression popped from the continuation stack and the environment popped from the stack and updated with continuation's name bound to the result's associated value. If a leaf path is associated with a channel creation construct, then the trace pool is updated with the leaf path extended with a sequential transition, and its associated state contains the next expression, and an environment updated with a new channel value. If a leaf path is associated with a let expression binding to a spawn contstruct, then the trace pool is updated with two new paths extending the leaf path. For one, the leaf path is extended with a sequential label 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 a label indicating a spawning transition. Its state has the spawned expression, the original environment, and an empty continuation stack.

If two leaves in the trace pool correspond to synchronization constructs on the same channel, where one synchronizes on a send event and the other synchronizes on a receive event, then the trace pool is updated with two new paths extending the two synchronizing leaf paths. The leaf path for the send event is extended by a label indicating sequential transition. For its state , the next expression of the let expression is used, and the environment is updated with the unit value bound to the let expression name. The leaf path for the receive event is also extended by a label indicating sequential transition. For its state, the next expression of the let expression is used, and the environment is updated with the value associated with the send event's second argument. Additionally, the communication set is updated with the send and receive paths, and the channel that is communicated on.

```
datatype continuation =
  Ctn name program env

type stack = continuation list

datatype state =
  Stt program env (continuation list)

type pool =
  program_path -> state option

predicate leaf of pool -> program_path -> bool
(∀ pool path stt .
  if
    pool path = Some stt and
    (∄ path' stt' . (pool path' = Some stt') and (strict_prefix path path')
    )
  then
    leaf pool path)

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

```
predicate concurrent_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 and
    pool path = Some (Stt (Rslt x) env ((Ctn x_k e_k env_k) # stack')) and
    env x = Some v
  then
    concurrent_eval
      pool convo
      (pool(path @ [LRtn x] -> (Stt e_k env_k(x_k -> v) stack'))) convo)
    and

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

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

(∀ pool path x e' env stack .
  if
    leaf pool path and
    pool path = Some (Stt (Let x MkChn e') env stack)
  then
    concurrent_eval
      pool convo
      (pool(path @ [LNxt x] -> (Stt e' (env(x -> (VChn (Chan path x))))
    stack))) convo) and

(∀ pool path x e_c e' env stack convo .
  if
    leaf pool path and
    pool path = Some (Stt (Let x (Spwn e_c) e') env stack)
```

```
  then
    concurrent_eval
    pool convo
    (pool(
        path @ [LNxt x] -> (Stt e' (env(x -> VUnt)) stack),
        path @ [LSpwn x] -> (Stt e_c env [])))  convo) and

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

11 Dynamic Communication Definitions

The one shot predicate states that in a trace pool, there is only one control 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 control paths of the trace pool. If two paths are ordered, that is, one is the prefix of the other and vice versa, then the dynamic semantics states that they necessarily occur in sequence, so the shorter path must synchronize before the longer path. Two paths may be competitive only if they are unordered. The fan in predicate states that there is no competition on the receiving end of a channel, by stating that any two paths that synchonize to receive on a channel are ordered. The fan out predicate states that there is no competition on the sending end of a channel, by stating that any two paths that syncronize to send on a channel are ordered. The one to one predicates states that there is no competition on either the receiving or the sending ends of a channel.

```
predicate is_send_path of pool -> channel -> program_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) and
```

```
    env x_e = Some (VPrm (SendEvt x_sc x_m) env_e) and
    env_e x_sc = Some (VChn c)
  then
    is_send_path pool c path)

predicate is_recv_path of pool -> channel -> program_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) and
    env x_e = Some (VPrm (RecvEvt x_rc) env_e) and
    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 and 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) and
(∀ path2 path1 . if prefix path2 path1 then
  ordered path1 path2)

predicate one_shot of pool -> channel -> bool:
(∀ pool c .
  if
    every_two (is_send_path pool c) (op =)
  then
    one_shot pool c)

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

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

```
    fan_in pool c)

predicate one_to_one of pool -> channel -> bool:
only
(∀ pool c.
  if
    fan_out pool c and
    fan_in pool c
  then
    one_to_one pool c)
```

12 Static Semantics Definitions

In order to determine communication topologies with some decent amount of precision, it is necessary to determine values associated with expressions and names of a program. In a language that allows inifinite recursion, it is not always possible to determine the exact values. However, it's also not necessary. Abstract values, which are imprecise approximations of the values at runtime, are good enough.

abstract values consist of abstract unit, abstract channels, and abstract primitive values. The abstract unit is actually no less precise than the actual unit value. The abstract channel is identified only by the name at which it is created, rather than the full control path that leads up to it. The abstract primitive value is simply a primitive construct without an environment for looking up its named arguments.

The static evaluation predicate is a relation from an input program to two outputs, each a map of names to sets of abstract values. A set of abstract values can be thought as a further abstraction, allowing less precision. The first output represents a the values that might bind to names. The second output represents values that might be passed over channels.

The definition for static evaluation is nondeterministic, but its syntax-directed structure suggests a refinement to computable definition, thus proving soundness with this definition should be sufficient to prove soundness for the more practical computable definition.

For a result expression, there may be any abstract value bindings or communication. For all let expressions, the bindings and communication are outputs for the next expression. For unit binding, the binding name is bound to the abstract unit. For channel creation, the binding name is bound to an abstract channel identified by the same binding name. For all primtive contruct bindings, the binding name is bound to the abstract primtive value represented by the primitive construct. Additionaly for the primitive construct of function abstraction, the the recursive parameter to the function abstraction is bound to the function abstraction, and the bindings and communication are also outputs for the function abstraction's internal expression. For spawn, the bindings and communication are also outputs for the spawned expression, and the the binding name is bound to the abstract unit.

For synchronization, if the send event is bound to the syncrhonization argument, then the binding name is bound to abstract unit, and the send event's channel com-

municates the argument associated with the send event's message argument. If the receive event is bound to the synchronization argument, then the binding name is bound to the abstract value communicated by the receive event's channel.

For the first projection, if its argument is a pair, then the binding name is bound to the abastract value associated with the pair's first argument. For the second projection, if its argument is a pair, then the binding name is bound to the abstract value associated with the pair's second argument.

For the case matching, if its first argument is a left case, then the left name is bound to the abstract value of the left case, the let binding name is bound to the result value of the left expression, and the bindings and communication are outputs for the left expression. If its first argument is a right case, then the right name is bound to the abstract value of the right case, the right binding name is ound to the result value of the right expression, and the bindings and communication are outputs for the right expression.

```
datatype static_value =
  SChn name | SUnt | SPrm primitive

type static_environment = name -> static_value set

fun result_name of program -> var:

(∀ x .
  result_name (Rslt x) = x) and

(∀ x b e' .
  result_name (Let x b e') = (result_name e))


predicate static_eval of static_environment -> static_environment ->
    program -> bool:
only

(∀ env_a convo_a x.
    static_eval env_a convo_a (Rslt x)) and

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

(∀ x env_a convo_a e' .
  if
    (SChn x) ∈ env_a x and
    static_eval env_a convo_a e'
  then
```

```
    static_eval env_a convo_a (Let x MkChn e')) and

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

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

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

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

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

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

(∀ f x_p e_b env_a convo_a x e' .
  if
    SUnt ∈ env_a f and
    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 and
        SChn x_c ∈ env_a x_sc
      then
        SUnt ∈ env_a x and env_a x_m ⊆ convo_a x_c) and

    (∀ x_rc x_c .
      if
        (SPrm (RecvEvt x_rc)) ∈ env_a x_e and
        SChn x_c ∈ env_a x_rc and
      then
        convo_a x_c ⊆ env_a x) and

    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Sync x_e) e')) and

(∀ 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) and
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Fst x_p) e')) and

(∀ 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) and
    static_eval env_a convo_a e'
  then
    static_eval env_a convo_a (Let x (Snd x_p) e')) and

(∀ 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 and
      env_a (result_name e_l) ⊆ env_a x and
      static_eval env_a convo_a e_l) and

    (∀ x_c . if (SPrm (Rght x_c)) ∈ env_a x_s then
      env_a x_c ⊆ env_a x_r and
      env_a (result_name e_r) ⊆ env_a x and
```

```
      static_eval env_a convo_a e_r) and

    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')) and

(∀ 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 and
      env_a (result_name e_b) ⊆ env_a x) and

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

The static reachable predicate states that an expression can be reached from another expression, by simply stating that an expression is reachable from itself, and an expression is reachable from an initial expression, if it is also reachable from the sub expressions of the initial expression.

for function application, if a function abstraction is bound to the first argument of application, then the abstract value of the second argument of application (the function argument) is bound to the abstraction's parameter name, and the resulting abstract value of the abstraction is bound the the let binding name.

```
predicate static_reachable of program -> program -> bool:
only
(∀ e .
    static_reachable e e) and
(∀ e_c e_z x e' .
  if
    static_reachable e_c e_z
  then
    static_reachable (Let x (Spwn e_c) e') e_z) and

(∀ 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) and

(∀ 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) and

(∀ 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) and

(∀ 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, yet also somewhat precisely, it will be advantageous to describe every expression in a with a short description. There are two forms of expressions, let binding expressions and result expressions. A let binding expression is associated with a label indicating its form and contains the name used for its first binding. A result expression is associated with a label indicating its form and contains the name of the result's argument. The dynamic description also contains labels for representing the mode of transition, which are slightly different due to the differences in how the static and dynamic relations can be defined. Since a goal of the static relations is that they imply a computable refinement, there is an additional restriction to consider, resulting in the slightly different labeling of transitions between expressions.

the static send label predicate states that a label might represent a sycnronization to send on a given abstract channel, in a given a program with given abstract value bindings. The static receive label predicate state that a label might represent a synchronization to receive on a given abstract channel, in a given program with given abstract value bindings.

```
datatype label = NLet name | NResult var

fun top_label of program -> label:
(∀ x b e' .
  top_label (Let x b e') = NLet x) and
(∀ x .
  top_label (Rslt x) = NResult x)

type label_map = label -> name set

predicate static_send_label of static_environment -> program -> name ->
    label -> bool:
only
(∀ prog0 x x_e e' x_sc x_m env_a x_c .
  if
    static_reachable prog0 (Let x (Sync x_e) e') and
```

```
    (SPrm (SendEvt x_sc x_m)) ⊆ env_a x_e and
    (SChn x_c) ∈ env_a x_sc
  then
    static_send_label env_a prog0 x_c (NLet x))

predicate static_recv_label of static_environment -> program -> name ->
    label -> bool:
only
(∀ prog0 x x_e e' x_rc env_a x_c .
  if
    static_reachable prog0 (Let x (Sync x_e) e') and
    (SPrm (RecvEvt x_rc)) ∈ env_a x_e and
    (SChn x_c) ∈ env_a x_rc
  then
    static_recv_label env_a prog0 x_c (NLet x))
```

——low precision // no live channel analysis —- Reppy and Xiao's work relies on detecting the liveness of channels in order to gain higher precision in the communication analysis. Since formal proofs are inherently complicated and packed with details, I decided to leave out the static analysis initially in order to simplify the problem and remove certain complications in formally proving soundness.

However, I have purposely structured the definitions to make adding live channel analysis to the definition fairly straight forward with just a few alterations to the definitions. In section ? I expand on these alterations and outline a strategy that I belive will result in formal proofs of soundness, although the actual formal proof of the version with live channel analysis has not yet been completed.

For the simpler version, there are four kinds of control edges, indicating how one point in a program transitions to another point in a program. The modes of transition are sequencing, calling, spawning, and returning. A transition is a triplet of a label, representing a point in the progra, a control edge, and another label. A static_point is just a label and a mode, without the destination label. An static_path is a list of static_points.

the static traversable predicate states 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.

For binding to case matching, the the calling transition from the top expression to the left case expression might be traversable, and the calling transition from the top expression to the right case expression might be traversable. The returning transition from the result of the left case expression to the sequenced expression might be

traversable, and the returning transition from the result of the right case expression to the sequenced expression might be traversable. Additionally, the traversable transitions are traversable transitions the left case expression, right case expression, and the sequenced expression.

For binding to function application, if the applied name is acutally bound to a function abstraction, then a calling transition from the top expression to the inner expression of the abstraction might be traversable, and the returning transition from the result of the function abstracted expression to the sequenced expression. Additionally, the traversable transitions are traversable transitions for the sequenced expression.

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

type transition = label * mode * label

type static_point = label * mode

type static_path = static_point list


predicate static_traversable of static_environment -> transition_set ->
    program -> bool:
only
(∀ env_a prog_a x .
    static_traversable env_a prog_a (Rslt x)) and

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

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


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

(∀ x e' prog_a env_a x_c .
```

```
  if
    (NLet x , MNxt, top_label e') ∈ prog_a and
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Prim (RecvEvt x_c)) e')) and


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


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


(∀ x e' prog_a env_a e_b f x_p .
  if
    (NLet x , MNxt, top_label e') ∈ prog_a and
    static_traversable env_a prog_a e' and
    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')) and

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

(∀ x e' prog_a env_a x_se .
  if
```

```
    (NLet x , MNxt, top_label e') ∈ prog_a and
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Sync x_se) e')) and

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

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

(∀ x e_l e_r e' prog_a env_a x_s .
  if
    {(NLet x, MCll, top_label e_l),
      (NLet x, MCll, top_label e_r),
      (NResult (result_name e_l), MRtn, top_label e'),
      (NResult (result_name e_r), MRtn, top_label e')} ⊆ prog_a and
    static_traversable env_a prog_a e_l and
    static_traversable env_a prog_a e_r and
    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'))
    and

(∀  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_label e_b),
        (NResult (result_name e_b), MRtn, top_label e')} ⊆ prog_a) and

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

The static traceable predicate states that an abstract control path can be traced according to a set of transitions from a starting label to a final label matching a given condition. An empty path might be traceable if the start label is also the final label. A path of one static_point might be traceable if its label is the required start label, and there is a transition containing the static_point and the transition's destination label meets the final label requirement. For a path of two or more static_points, the first label is the start label, there is a transition from the first static_point to the second static_point, and the path from the second static_point onward also might be

traceable.

```
predicate static_traceable of
  transition set -> label -> (label -> bool) -> static_path -> bool:
only
(∀ start prog_a is_end .
  if
    is_end start
  then
    static_traceable prog_a start is_end []) and

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

The static inclusive predicate states that two abstract paths represent might be traversed 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) and

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

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

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

The singular predicate states that two paths are the same or cannot occur in the same run of a program. The noncompetitive predicates states that two paths are ordered or cannot occur in the same run of a program. The static one shot predicate states that there is at most one attempt to syncrhonize to send on an abstract channel in a run of a given program with given abstract value bindings.

```
predicate singular of static_path -> static_path -> bool:
only
(∀ path .
  singular path path) and

(∀ 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) and

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

The static one to one predicate states that there is at most one thread attempting to send and one thread attempting to receive on a given channel at any time during a run of a given program.

The static fan out predicate states that there is at most one thread attempting to send on a given channel at any time during a run of a given program. The static fan in predicate states that there is at most one thread attempting to receive on a given channel at any time during a run of a given program.

```
predicate static_one_shot of static_environment -> program -> name -> bool:
only
(∀ prog_a e env_a x_c .
  if
    every_two
      (static_traceable prog_a (top_label e) (static_send_label env_a e x_c
    ))
      singular and
    static_traversable env_a prog_a e
```

```
  then
    static_one_shot prog_a e x_c)

predicate static_one_to_one of static_environment -> program -> name ->
    bool:
only
(∀ prog_a e env_a x_c .
  if
    (every_two
      (static_traceable prog_a (top_label e) (static_send_label env_a e x_c
    ))
      noncompetitive) and
    (every_two
      (static_traceable prog_a (top_label e) (static_recv_label env_a e x_c
    ))
      noncompetitive) and
    static_traversable env_a prog_a e
  then
    static_one_to_one env_a e x_c)

predicate static_fan_out of static_environment -> program -> name -> bool:
only
(∀ prog_a e env_a x_c .
  if
    (every_two
      (static_traceable prog_a (top_label e) (static_send_label env_a e x_c
    ))
      noncompetitive) and
    static_traversable env_a prog_a e
  then
    static_fan_out env_a e x_c)

predicate static_fan_in of static_environment -> program -> name -> bool:
only
(∀ prog_a e env_a x_c .
  if
    (every_two
      (static_traceable prog_a (top_label e) (static_recv_label env_a e x_c
    ))
      noncompetitive) and
    static_traversable env_a prog_a e
  then
    static_fan_in env_a e x_c)
```

14 Formal Reasoning

The reasoning involved in proving each soundness theorem is based around breaking the goal into simpler subgoals, and generalizing assumptions to create useful induc-

tion hypotheses. It is often useful to create helper definitions that can be deduced from premises 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 use of these predicates on the initial program state.

Some parts of the generalized predicate definitions exist simply to prove that they imply uses of the original expression predicate. However, the reason for the existence of the generalized definitions is to allow direct access to properties that would otherwise be deeply nested in an inductive structure and inaccessble for proofs.

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 are defineable on only the strict prefix. In such cases, it would be necessary to define sortedness in two ways, and prove their equivalence, in order to prove theorems relating to less flexible relations.

The lemma for soundness of static one shot states that if an abstract channel is statically classified as one shot for a given program, and value bindings derived from the program, then any corresponding dynamic channel is classified as one shot over any trace pool that results from running the program. The lemma for static_fan_out_sound states that if an abstract channel is statically classified as fan_out for a given program and an abstract environment derived from that program, then any corresponding concrete channel is classified as fan_out over any pool that results from running the program. The lemmas for soundness of fan in, and one to one follow similar patterns.

The static predicates for communication topologies should be generalizations of procedures that compute the topologies from programs. The soundness of the static predicates must be provable to be certain that statically described topologies are also observed when running the program.

```
-----
theorem static_one_shot_sound:

∀ env_a convo_a prog0 x_c pool convo path_c .
if
  static_eval env_a convo_a prog0 and
  static_one_shot env_a prog0 x_c and
  star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo
then
  one_shot pool (Chan path_c x_c)

-----
theorem static_fan_out_sound:
∀ env_a convo_a prog0 x_c pool convo path_c.
  if
    static_eval env_a convo_a prog0 and
    static_fan_out env_a prog0 x_c and
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo
```

```
  then
    fan_out pool (Chan path_c x_c)"

-----
theorem static_fan_in_sound:

∀ env_a convo_a prog0 x_c pool convo path_c.
  if
    static_eval env_a convo_a prog0 and
    static_fan_in env_a prog0 x_c and
    star concurrent_eval [[] -> (Stt e [->] [])] {} pool convo
  then
    fan_in pool (Chan path_c x_c)

-----
theorem static_one_to_one_sound:

∀ env_a convo_a prog0 x_c pool convo path_c.
  if
    static_eval env_a, convo_a prog0 and
    static_one_to_one env_a prog0 x_c and
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo
  then
    one_to_one pool (Chan path_c x_c)

-----
```

Theorem static_fan_out_sound is proved by a few simpler lemmas and the definitions of static_fan_out and fan_out. The three main lemmas are not_static_send_site_sound, not_static_inclusive_sound, and not_static_traceable_sound. Additionally, the inductive definition paths_corresponds, which simply relates abstract paths to concrete paths, is helpful.

Lemma not_static_inclusive_sound states that if any two paths traced by running a programm correspond two abstract paths that are statically inclusive. It follows from a straight forward case analysis of static_inclusive.

Lemma not_static_traceable_sound states that if running a programming traces a concrete path, and a static_program contains the statically traversable static_points of the program, then there is an abstract path corresponding to the traced concrete path is statically traceable.

Lemma not_static_send_site_sound states that if a program is run to reach a synchronization on a send event, then the label corresponding to the syncrhonization expression is statically identified as a send site.

```
------

lemma not_static_inclusive_sound:

∀ prog0 pool convo path1 stt1 path2 stt2 path_a1 path_a2 .
  if
```

```
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo
    pool path1 = Some stt1 and
    pool path2 = Some stt2 and
    paths_correspond path1 path_a1 and
    paths_correspond path2 path_a2
  then
    static_inclusive path_a1 path_a2

-----
lemma not_static_traceable_sound:

∀ prog0 pool convo path x b e' env stack evn_a convo_a prog_a is_end .
  if
    star concurrent_eval ([[] -> (Stt prog0 [->] [])], {}) (pool, convo)
     and
    pool path = Some (Stt (Let x b e') env stack) and
    static_eval env_a convo_a prog0 and
    static_traversable env_a prog_a prog0 and
    is_end (NLet x)
  then
    ∃ path_a .
      paths_correspond path path_a and
      static_traceable prog_a (top_label prog0) is_end path_a

-----

lemma not_static_send_site_sound:

∀ prog0 pool convo path x x_e e' env stack x_sc x_m env' path_c x_c .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo and
    pool path = Some (Stt (Let x (Sync x_e) e') env stack) and
    env x_e = Some (VPrm (SendEvt x_sc x_m) env') and
    env' x_sc = Some (VChn (Chan path_c x_c)) and
    static_eval env_a convo_a prog0
  then
    static_send_label env_a prog0 x_c (NLet x)

-----
```

Lemma not_static_traceable_sound is proved by generalizing static_traversable and static_eval over pools, such that information about the a point in the program can be deduced by a fixed number of static_points regardless of where the location of the program point or the size of the program. Without such generalization, it would be possibe to prove soundness for a fixed program, but not an arbitrary programs.

The generalization of static_traversable is captured in the predicates static_traversable_val, static_traversable_env, static_traversable_stack, and static_traversable_pool. The generalization is designed to have two main features. First, the static information that describes one pool should also describe subsequent pools from running the program,

as information shifts from the expression to the environment, and stack. Second, certain should be described directly by a fixed number of logical steps, in constract to its representation in the original predicate, which requires knowledge of features particular context in a program. Most of the rules simply state that the nearest subexpression is statically traversable, and static_traversable offers direct information about statically described static_points in a program. The exception is in the definition of static_traversable_stack, in the rule for a nonempty stack, where there is the additional clause that the edge from a return point to the label of the continuation expression exists in the static_program. This information is consistent with the definition of static_traversable for expressions, but provides information about an edge in the static program with a fixed number of logical steps, which would otherwise only be decucible by a varying number of logical steps dependent on location of an expression in a program.

```
predicate static_traversable_val of static_environment -> transition_set ->
     program_value -> bool:

only

(∀ env_a prog_a . static_traversable_val env_a prog_a VUnt) and

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

(∀ 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)) and

(∀ 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)) and

(∀ 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)) and

(∀ 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)) and
```

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

(∀ 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_environment -> transition_set ->
     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_environment -> transition_set -> name -> continuation list -> bool
    :

only

(∀ env_a prog_a y . static_traversable_stack env_a prog_a y []) and

(∀ y e prog_a env_a prog_a env stack x env .
  if
    {(NResult y, MRtn, top_label e)} ⊆ prog_a and
    static_traversable env_a prog_a e and
    static_traversable_env env_a prog_a env and
    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_environment -> transition_set -> 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 and
      static_traversable_env env_a prog_a env and
      static_traversable_stack env_a prog_a (result_name e) stack)
  then
      static_traversable_pool env_a prog_a pool)

-----
```

The abstract program transitions described by the various versions of static_traversable are dependent on abstract value bindings (in the application case), which are described by the static_eval predicate. Thus generalized versions of static evaluation enables further deduction of abstract program transitions. As with the generalized versions of static_traversable, the generalized versions of static_eval are designed to preserve abstract value bindings across program execution static points, and also provide direct access to abstract binding information in a fixed number of logical steps.

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

predicate static_eval_value of
  static_environment -> abstract_convo -> program_value -> bool:

only

(∀ env_a convo_a . static_eval_val env_a convo_a VUnit) and

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

(∀ 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)) and

(∀ 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)) and

(∀ 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)) and

(∀ 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)) and


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

(∀ 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_environment -> static_environment ->
    env -> bool:

only

(∀ env_a convo_a env .
  if
    (∀ x v . if env x = Some v then
      {abstract v} ⊆ env_a x and
      static_eval_val env_a convo_a v)
  then
    static_eval_env env_a convo_a env)


predicate static_eval_stack of
  static_environment -> static_environment -> static_value set ->
    continuation list -> bool:
```

```
only

(∀ env_a convo_a res_a . static_eval_stack env_a convo_a res_a []) and

(∀ res_a env_a convo_a .
  if
    res_a ⊆ env_a x and
    static_eval env_a convo_a e and
    static_eval_env env_a convo_a env and
    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_environment -> static_environment ->
    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 and
      static_eval_env env_a convo_a env and
      static_eval_stack env_a convo_a env_a (result_name e) stack)
  then
    static_eval_pool env_a convo_a pool)
```

Lemma not_static_traceable_sound follows from the generalized lemma not_static_traceable_pool_sound, which contains the generalized premise of static_traversable_pool. The generalized lemma follows from lemma static_traversable_pool_preserved_star, the predicate star_left and its equivalence to star, and induction on star_left. star_left inductively defines a a binary relation as unraveling from right to left, rather than left to right. In the case of steps through a program, it unravels the execution from the end to the beginning, which enables deducing traceablity of a path from its slightly shorter predecessor.

The lemma also relies on the definition of static_traversable_pool and lemma static_traversable_pool_preserved to deduce abstract transition information about the pool containing the traced path.

```
predicate star_left of ('a -> 'a -> bool) -> 'a -> 'a -> bool:
only
(∀ r z z .
  star_left r z z) and
(∀ r x y z .
  if
    star_left R x y and 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 and
    star_left r y z
  then
    star_left r x z)
```

lemma star_implies_star_left follows from induction on star and star_left_trans, which follows from induction on the latter star_left. A similar approach deduces the fact that star_left implies star.

```
lemma not_static_traceable_pool_sound:
∀ prog0 pool convo path x b e' env stack evn_a convo_a prog_a is_end .
  if
    star concurrent_eval ([[] -> (Stt prog0 [->] [])], {}) (pool, convo)
     and
    pool path = Some (Stt (Let x b e') env stack) and
    static_eval env_a convo_a prog0 and
    static_traversable env_a prog_a pool and
    is_end (NLet x)
  then
    ∃ path_a .
      paths_correspond path path_a and
      static_traceable prog_a (top_label prog0) is_end path_a

lemma static_traversable_pool_preserved_star:
∀ prog0 pool convo env_a convo_a prog_a .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo and
    static_eval env_a convo_a prog0 and
    static_traversable_pool env_a prog_a [[] -> (Stt prog0 [->] [])]
  then
    static_traversable_pool env_a prog_a pool
```

Lemma static_traversable_pool_preserved_star follows from induction on star's equivalent star_left and lemma static_eval_pool_preserved_star, and the information about abstract bindings from static_eval_pool.

Lemma not_static_bound_sound follows from static_eval_pool_preserved, which results from induction on star concurrent_eval.

```
lemma static_eval_pool_preserved:
∀ pool convo pool' convo' env_a convo_a .
  if
    star concurrent_eval pool convo pool' convo'
    static_eval_pool env_a convo_a pool
  then
    static_eval_pool env_a convo_a pool'
```

lemma not_static_send_site_sound is proved using the lemmas send_chan_not_static_bound_sound, not_static_bound_sound, and not_static_reachable_sound.

```
lemma send_chan_not_static_bound_sound:
∀ prog0 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 concurrent_eval [[] -> (Stt prog0 [->] [])], {} pool convo and
    static_eval env_a convo_a prog0 and
    pool path = Some (Stt (Let x (Sync x_e) e') env stack) and
    env_y x_e = Some (VPrm (SendEvt x_sc x_m) env_e) and
    env_e x_sc = Some (VChn (Chan path_c x_c))
  then
    SChn x_c ∈ env_a x_sc


lemma not_static_bound_sound:
∀ prog0 pool convo env_a convo_a path e env stack x v .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo and
    static_eval env_a convo_a prog0 and
    pool path = Some (Stt e env stack) and
    env x = Some v
  then
    abstract v ∈ env_a x


lemma not_static_reachable_sound:
∀ prog0 pool convo env_a convo_a path e env stack .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo and
    pool path = Some (Stt e env stack)
  then
    static_reachable prog0 e
```

Both send_chan_not_static_bound_sound, and not_static_bound_sound follow from static_eval_pool_preserved_star descrbied previously. The lemma not_static_reachable_sound relies on the helper definition static_reachable_left and generalizations based on that. The definition of static_reachable is syntax-directed in order to potray 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 show that each intermediate expression is reachable from the

initial expression. Thus, the induction should be from the end to the begininng of the program.

The predicate static_reachable follows from static_reachable_left, and it enables unraveling the reachable information from the end to the beggining. Likewise, induction on star_left is used, instead of star, to unravel the semantics from the end to the beginning, keeping the inital program in context foreach step.

The lemma not_static_reachable_sound follows the the definitions of star_left, static_reachable_left, its generalizations and the the lemma not_static_reachable_pool_sound.

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

only

(∀ prog0 e .
  static_reachable_left prog0 prog0) and

(∀ prog0 x e_c e' .
  if
    static_reachable_left prog0 (Let x (Spwn e_c) e')
  then
    static_reachable_left prog0 e_c) and

(∀ prog0 x x_s x_l e_l x_r e_r e' .
  if
    static_reachable_left prog0 (Let x (Case x_s x_l e_l x_r e_r) e')
  then
    static_reachable_left prog0 e_l) and

(∀ prog0 x x_s x_l e_l x_r e_r e' .
  if
    static_reachable_left prog0 (Let x (Case x_s x_l e_l x_r e_r) e')
  then
    static_reachable_left prog0 e_r) and

(∀ prog0 x f x_p e_b e' .
  if
    static_reachable_left prog0 (Let x (Prim (Abs f x_p e_b)) e')
  then
    static_reachable_left prog0 e_b) and

(∀ prog0 x f x_p e_b e' .
  if
    static_reachable_left prog0 (Let x b e')
  then
    static_reachable_left prog0 e')

predicate static_reachable_over_prim of program -> primitive -> bool:
only
(∀ prog0 x_c x_m .
```

```
  static_reachable_over_prim prog0 (SendEvt x_c x_m)) and
(∀ prog0 x_c .
    static_reachable_over_prim prog0 (RecvEvt x_c)) and
(∀ prog0 x1 x2 .
    static_reachable_over_prim prog0 (Pair x1 x2)) and
(∀ prog0 x_l .
    static_reachable_over_prim prog0 (Lft x_l)) and
(∀ prog0 x_r
    static_reachable_over_prim prog0 (Rght x_r)) and
(∀ prog0 e_b f x_p e_b .
  if
    static_reachable_left prog0 e_b
  then
    static_reachable_over_prim prog0 (Abs f x_p e_b))

predicate static_reachable_val of program -> program_value -> bool:
only
(∀ prog0 .
    static_reachable_over_val prog0 VUnt) and
(∀ prog0 c .
    static_reachable_over_val prog0 (VChn c)) and
(∀ prog0 p env .
  if
    static_reachable_over_prim prog0 p and
    static_reachable_over_env prog0 env
  then
    static_reachable_over_val prog0 (VPrm p env))

predicate static_reachable_env of program -> env -> bool:
only
(∀ prog0 env
  if
    (∀ x v . if env x = Some v  then static_reachable_over_val prog0 v)
  then
    static_reachable_over_env prog0 env)

predicate static_reachable_over_stack of program -> continuation list ->
    bool:
only
(∀ prog0 .
  static_reachable_over_stack prog0 []) and
(∀ prog0 e_k env_k stack' .
  if
    static_reachable_left prog0 e_k and
    static_reachable_over_env prog0 env_k and
    static_reachable_over_stack prog0 stack'
  then
    static_reachable_over_stack prog0 ((Ctn x_k e_k env_k # stack')))

predicate static_reachable_pool of program -> pool -> bool:
```

```
only
(∀ prog0 pool .
  then
    (∀ path e env stack . if pool path = Some (Stt e env stack) then
      static_reachable_left prog0 e and
      static_reachable_over_env prog0 env and
      static_reachable_over_stack prog0 stack)
   then
    static_reachable_over_pool prog0 pool)


lemma not_static_reachable_pool_sound:
(∀ prog0 pool .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] [])], {} pool convo
  then
    static_reachable_over_pool prog0 pool)


lemma not_static_reachable_pool_sound follows via induction on star_left
    and the lemmas
relating static_reachable to static_reachable left and star to star_left.


lemma static_reachable_left_implies_static_reachable:
(∀ prog0 e.
  if
    static_reachable_left prog0 e
  then
    static_reachable prog0 e)

lemma static_reachable_trans:
(∀ e1 e2 e3 .
  if
    static_reachable e1 e2 and
    static_reachable e2 e3
  then
    static_reachable e1 e3)
```

lemma static_reachable_left_implies_static_reachable follows from induction on static_reachable_left and static_reachable_trans, which follows from induction on static_reachable.

15 Static Communication with Channel Liveness

The higher precision defintions for static communication topologies enables the discrimination between channels created by the same piece of code, but at different instances during a run of a program. For example, a function abstraction may specify the creation of a channel with a binding to the name x. This function may be called

twice during a run of the program, resulting in two distinct channels. However, the abstract channels would be indentified by the same name x. Even if each instance of channel x only has one process sending on it, the static analysis discussed so far is only able to classiy a channel identified with x as having multiple threads sending on it. To get around this issue, the static communication definitions can be enhanced with additional information about the liveness of channels. By limiting analysis to only the sub portion of the program wher a particular channel is live, the analysis is able to ignore duplicate instances of channels, thus gain higher precision.

The sub portion of the program is represented by a static program containing all reachable nodes in the part of a program where a given channel is live, and transitive transitions to any of those nodes from the channel creation point. To maintain connetions between a channel creation portion of the program and a thread that received the channel as a message, transitions from the send site to the receive site of a syncrhonization is included in the static program, annoted with the sending mode, as indicated by the new definition of static_traversable. 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 = label * mode * label

type static_point_label = label * mode

type static_path = static_point_label list

predicate static_traversable of static_environment -> transition_set ->
    program -> bool:
only
(∀ env_a prog_a x .
    static_traversable env_a prog_a (Rslt x)) and

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

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


(∀ x e' prog_a env_a  x_c x_m .
  if
    (NLet x , MNxt, top_label e') ∈ prog_a and
```

```
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Prim (SendEvt x_c x_m)) e'))
    and

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


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


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


(∀ x e' prog_a env_a e_b f x_p .
  if
    (NLet x , MNxt, top_label e') ∈ prog_a and
    static_traversable env_a prog_a e' and
    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')) and

(∀ x e' e_c prog_a env_a.
  if
    {(NLet x, MNxt, top_label e'),
      (NLet x, MSpwn, top_label e_c)} ⊆ prog_a and
    static_traversable env_a prog_a e_c and
```

```
    static_traversable env_a prog_a e'
  then
    static_traversable env_a prog_a (Let x (Spwn e_c) e')) and

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

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

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

(∀ x e_l e_r e' prog_a env_a x_s .
  if
    {(NLet x, MCll, top_label e_l),
      (NLet x, MCll, top_label e_r),
      (NResult (result_name e_l), MRtn, top_label e'),
      (NResult (result_name e_r), MRtn, top_label e')} ⊆ prog_a and
    static_traversable env_a prog_a e_l and
    static_traversable env_a prog_a e_r and
    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'))
    and

(∀  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_label e_b),
        (NResult (result_name e_b), MRtn, top_label e')} ⊆ prog_a) and
```

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

For the live analysis it is necessary to track any variable that is bound to a channel of interest, since any use of that variable would indicate a use of the channel. The predicate static_built_on_chan states that a variable is bound to some structure containing a channel. In the case where the tracked variable might be a function abstraction, the channel simply needs to be live in the body of the abstraction, for the variable to be considered built on it.

```
predicate static_built_on_chan of
  static_environment -> label_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) and

(∀ x_sc x_m env_a x entr x_c .
  if
    (SPrm (SendEvt x_sc x_m)) ∈ env_a x and
    (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 and
    static_built_on_chan env_a entr x_c x_rc
  then
    static_built_on_chan env_a entr x_c x) and

(∀ x1 x2 env_a x entr x_c .
  if
    (SPrm (Pair x1 x2)) ∈ env_a x and
    (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) and

(∀ x_a env_a x entr x_c .
```

```
  if
    (SPrm (Lft x_a)) ∈ env_a x and
    static_built_on_chan env_a entr x_c x_a
  then
    static_built_on_chan env_a entr x_c x) and

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

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

Using the predicate static_built_on_chan, the predicate static_live_chan completes the description of channel liveness in terms of entry functions and exit functions. The entry function maps a program point to a set of variables built on the given channel, if those variables are live at the entry of that program point. The exit map maps a program point to a set of variables built on the given channel, if those variabes are live at the exit of that program point.

```
predicate static_live_chan of static_environment -> label_map -> label_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)) and

(∀ exit x entr e' env_a x_c .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    entr (top_label e') ⊆ exit (NLet x) and
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c (Let x Unt e')) and
```

```
(∀ exit x entr e' env_a x_c .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    entr (top_label e') ⊆ exit (NLet x) and
    static_live_chan env_a entr exit x_c e'
  then
    static_live_chan env_a entr exit x_c (Let x MkChn e')) and

(∀ exit x entr env_a x_c x_sc x_m e' x_c .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_sc then
      {x_sc} ⊆ entr (NLet x)) and
    (if static_built_on_chan env_a entr x_c x_m then
      {x_m} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    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
    ')) and

(∀ exit x entr env_a x_c x_r .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_r then
      {x_r} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    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'))
    and


(∀ exit x entr env_a e_c x1 x2 e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x1 then
      {x_1} ⊆ entr (NLet x)) and
    (if static_built_on_chan env_a entr x_c x2 then
      {x2} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    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'))
    and

(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_a then
```

```
      {x_a} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    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')) and

(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_a then
      {x_a} ⊆ entr (NLet x))
    entr (top_label e) ⊆ exit (NLet x) and
    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)) and

(∀  exit x entr e_b x_p x env_a x_c e' f .
  if
    (exit (NLet x) - {x}) ∪ (entr (top_label e_b) - {x_p}) ⊆ entr (NLet x)
     and
    static_live_chan env_a entr exit x_c e_b and
    entr (top_label e') ⊆ exit (NLet x) and
    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'))
     and

(∀ exit x entr e' e_c x_c env_a .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    entr (top_label e') ∪ entr (top_label e_c) ⊆ exit (NLet x) and
    static_live_chan env_a entr exit x_c e_c and
    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')) and

(∀ exit x entr env_a x_c x_e e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_e then
      {x_e} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    static_live_chan env_a entr exit x_c e' and
  then
    static_live_chan env_a entr exit x_c (Let x (Sync x_e) e')) and

(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_a then
```

```
      {x_a} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    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')) and

(∀ exit x entr env_a x_c x_a e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_a then
      {x_a} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    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')) and

(∀ 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) and
    (entr (top_label e_l) - {x_l}) ⊆ entr (NLet x) and
    (entr (top_label e_r) - {x_r}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_s then
      {x_s} ⊆ entr (NLet x)) and
    static_live_chan env_a entr exit x_c e_l and
    static_live_chan env_a entr exit x_c e_r and
    entr (top_label e') ⊆ exit (NLet x) and
    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')) and

(∀ exit x entr env_a x_c x_a f e' .
  if
    (exit (NLet x) - {x}) ⊆ entr (NLet x) and
    (if static_built_on_chan env_a entr x_c x_a then
      {x_a} ⊆ entr (NLet x)) and
    (if static_built_on_chan env_a entr x_c f then
      {f} ⊆ entr (NLet x)) and
    entr (top_label e') ⊆ exit (NLet x) and
    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 predicate static_live_traversable states that an abstraction transition exists in some static program, and is considered live with respect to given entry and exit functions. The predicate static_live_traceable states that an entire abstract path exists in some static program and is live with respect to some entry and and exit functions.

```
predicate static_live_traversable of
  transition_set -> label_map -> label_map -> transition -> bool:

only

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

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

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

(∀ l l' prog_a entr exit .
  if
    (l, MRtn, l') ∈ prog_a and
    not (entr l' = {})
  then
    static_live_traversable 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 and
    {x_evt} ⊆ (entr (NLet x_send))
  then
    static_live_traversable prog_a entr exit ((NLet x_send), ESend x_evt, (
    NLet x_recv)))



predicate static_live_traceable of
  static_environment -> transition_set -> label_map -> label_map ->
  label -> (label -> bool) -> static_path -> bool:

only
```

```
(∀ is_end start env_a prog_a entr exit .
  if
    is_end start
  then
    static_live_traceable prog_a entr exit start is_end []) and

(∀ prog_a entr exit start middle path is_end mode.
  if
    static_live_traceable prog_a entr exit start (λ l . l = middle) path
    and
    (is_end end) and
    static_live_traversable prog_a entr exit (middle, mode, end)
  then
    static_live_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 predicate static_inclusive describes relationship between paths for those paths to possible 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 with the containing the sending mode. As mentioned earlier, the path from the synchronization on sending to the syncrhonization on receiving is necessary to ensure that all uses of a channel are reachable from the channel's creation point. The singular predicate states that only one of the two given paths can occur in a run of program. The noncompetitve predicate states 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) and

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

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

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

(∀ 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) and

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


predicate noncompetitive of static_path -> static_path -> bool:

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

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

The predicates static_one_shot, static_one_to_one, static_fan_out, and static_fan_in describe the communication topologies with higher precision that the original definitions by using the predicates based on static programs trimmed down to contain only the parts where a given channel is live. The general structure of the predicate definitions is overall similar to that of the analgous lower precision definitions.

```
predicate static_one_shot of static_environment -> program -> name -> bool:

only

(∀ prog_a entr exit x_c env_a e .
```

```
  if
    every_two
      (static_live_traceable prog_a entr exit (NLet x_c) (static_send_label
     env_a e x_c))
      singular and
    static_live_chan env_a entr exit x_c e and
    static_traversable env_a prog_a e
  then
    static_one_shot V e x_c)

predicate static_one_to_one of static_environment -> program -> name ->
    bool:

only

(∀ prog_a entr exit x_c env_a e .
    every_two
      (static_live_traceable prog_a entr exit (NLet x_c) (static_send_label
     env_a e x_c))
      noncompetitive and
    every_two
      (static_live_traceable prog_a entr exit (NLet x_c) (static_recv_label
     env_a e x_c))
      noncompetitive and
    static_live_chan env_a entr exit x_c e and
    static_traversable env_a prog_a e
  then
    static_one_to_one env_a e x_c)

predicate static_fan_out of static_environment -> program -> name -> bool:

only

(∀ prog_a entr exit x_c env_a e .
  if
    every_two
      (static_live_traceable prog_a entr exit (NLet x_c) (static_send_label
     env_a e x_c))
      noncompetitive and
    static_live_chan env_a entr exit x_c e and
    static_traversable env_a prog_a e
  then
    static_fan_out env_a e x_c)

predicate static_fan_in of static_environment -> program -> name -> bool:

only

(∀ prog_a entr exit x_c env_a e .
  if
```

```
    every_two
      (static_live_traceable prog_a entr exit (NLet x_c) (static_recv_label
     env_a e x_c))
      noncompetitive and
    static_live_chan env_a entr exit x_c e and
    static_traversable env_a prog_a e
  then
    static_fan_in env_a e x_c)
```

16 Reasoning Stategy for Static Communication with Channel Liveness

To prove soundess of the commounication topology classification, it should be possible to use previous techniques of generalizing propositions over pools and other semantic components, along with finding equivalent representaitons of propositions that vary in the inductive subcomponent. One thing that will make carrying out the formal proof particulary tricky is that paths in the dynamic semantics do not correspond one to one with abstract paths in the static program used for static analysis. Essentially, it will be necessary to show that static properties that hold for some abstract path are related in some fashion to dyamic properties that hold for a dynamic path which corresponds to the abstract path module the channel under analysis. These reasoning strategies are demonstrated by the theorem static_one_shot_sound.

```
theorem static_one_shot_sound:

∀ prog0 pool convo env_a convo_a x_c path_x .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] []] {} pool convo and
    static_eval env_a convo_a prog0 and
    static_one_shot env_a prog0 x_c
  then
    one_shot pool (Chan path_c x_c)
```

The theorem static_one_shot_sound depends on correlating dynamic paths with abstract paths, which is described by the predicates paths_correspond and paths_correspond_mod_chan. Additionally the theorem follows from not_static_traceable_sound, not_static_inclusive_sound and not_send_site_sound. The reasoning for the latter one, 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 abstract paths. The proofs depend on finding an abstract path that depends on a given dynamic path. In the lower precision analysi the cor-

relation was straight forward. There was only one possible abstract 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 abstract that correlates with the dynamic path is much more challenging.

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

paths_correspond [] [] and

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

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

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

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


predicate paths_correspond_mod_chan of
  pool -> conversation -> channel -> program_path -> static_path -> bool:

only

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

```
(∀ 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 @ (LNxt x_r) # path_sfx) = Some stt and
    pool path_s = Some (Stt (Let x_s (Sync x_se) e_sy) env_sy stack_sy) and
    pool path_r = Some (Stt (Let x_r (Sync x_re) e_ry) env_ry stack_ry) and
    {(path_s, c_c, path_r)} ⊆ convo and
    dynamic_built_on_chan_var env_ry c x_r and
    paths_correspond_mod_chan pool convo c path_s path_a_pre and
    paths_correspond path_sfx path_a_sfx
  then
    paths_correspond_mod_chan pool convo c
      (path_r @ (LNxt 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:
∀ prog0 pool convo path x b e' env stack env_a convo_a
  entr exit x_c prog_a is_end path_c  .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo and
    pool path = Some (Stt (Let x b e') env stack) and
    static_eval env_a convo_a prog0 and
    static_live_chan env_a entr exit x_c prog0 and
    static_traversable env_a prog_a prog0 and
    is_end (NLet x)
  then
    (∃ path_a .
      paths_correspond_mod_chan pool convo (Chan path_c x_c) path path_a
    and
      static_live_traceable prog_a entr exit (NLet x_c) is_end path_a)


lemma not_static_inclusive_sound:
∀ prog0 pool convo env_a entr exit x_c prog_a convo_a
  path1 stt1 path_c path_a1 path2 stt2 path_a2 .
  if
    star concurrent_eval [[] -> (Stt prog0 [->] [])] {} pool convo and
    static_live_chan env_a entr exit x_c prog0 and
    static_traversable env_a prog_a prog0 and
    static_eval env_a convo_a prog0 and
    pool path1 = Some stt1 and
    paths_correspond_mod_chan pool convo (Chan path_c x_c) path1 path_a1
    and
    static_live_traceable prog_a entr exit
      (NLet x_c) (static_send_label env_a prog0 x_c) path_a1 and
    pool path2 = Some stt2 and
    paths_correspond_mod_chan pool convo (Chan path_c x_c) path2 path_a2
```

```
  and
  static_live_traceable prog_a entr exit
    (NLet x_c) (static_send_label env_a prog0 x_c) path_a2
then
  static_inclusive path_a1 path_a2
```

17 Mathematical Artifacts

18 Syntax

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

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

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

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

```
let z12 = rqst (srvr, L (R ())) in
()
```