

# Formal Theory of Communication in CML

Thomas Logan

January 16, 2019

# 1 Introduction

For this master’s thesis, I have developed a formal semantics of a language with concurrent processes (or threads), an initial formal analysis, along with related theorems and formal proofs. The language under analysis is a very simplified version of *Concurrent ML* [18]. The formal analysis recasts an analysis with informal proofs developed by Reppy and Xiao [17]. It categorizes communication described by programs into simple topologies. One description of topologies is static; that is, it describes all static topologies of a program in a finite number of steps. 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 contains many more details making it more challenging to prove formally than the lower precision analysis. The proofs for the soundness theorems of the lower precision analysis have been mechanically verified using Isabelle/HOL [16], while the higher precision analysis is currently under development. Indeed, one of the motivations for implementing the analysis in a mechanical 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. Thus, recasting Reppy and Xiao’s work was far more nuanced than a straightforward syntactic transliteration. Although the definitions are structurally quite different, their philosophical equivalence is hopefully apparent. In this formal theory, the dynamic semantics of Concurrent ML consists of a CEK machine [5]. The static semantics consists of a 0CFA control flow analysis [19], defined in terms of constraints [14].

## 1.1 Concurrent ML

In programing languages, concurrency is a program structuring technique that allows evaluation steps to hop back and forth between disjoint syntactic structures within a program. It is useful when conceptually distinct tasks need to overlap in time, but are easier to understand if they are written as distinct structures within the program. Concurrent languages may also allow the evaluation order between steps of terms 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 to pick the most efficient execution order. A common use case for concurrency is for programs that interact with humans, in which a program has to process various requests while remaining responsive to subsequent user inputs, and it must continually provide the user feedback with latest information it has processed.

*Concurrent ML* is a particularly elegant programing language for concurrency. 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. Additional threads can be spawned in order to share data asynchronously.

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 updates 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` with a number `v` and returns a number from the server. Internally, it extracts the request channel `reqCh` from the server handle and creates a new channel `replCh`. It makes a request to the server with the number `v` and the reply channel `replCh` by calling `send (reqCh, (v, replCh))`. Then it receives the reply with the new number by calling `recv replCh`.

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

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

```

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

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

end

```

*Concurrent ML* actually allows for events other than sending and receiving to occur during synchronization. In fact, the synchronization mechanism is decoupled from events, like sending and receiving, much in the same way that function application is decoupled from `.` 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 chan -> 'a event
val sendEvt : 'a channel * 'a -> unit event

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

```

An advantageous consequence of decoupling synchronization from events, is that events can be combined with other events via event combinators, and synchronized on exactly once. One such event combinator is `choose`, which constructs a new event consisting of two constituent events, such that when synchronized on, exactly one of the two events may take effect. There are many other useful combinators, such as the `wrap` and `guard` combinators designed by Reppy[8].

Additionally, Donnelly and Fluet extended *Concurrent ML* with the `thenEvt` combinator described in their work on transactional events [4]. Transactional events enable more robust structuring of programs by allowing non-isolated code to be turned into isolated code via the `thenEvt` combinator, instead of duplicating code with the addition of stronger isolation. When the event constructed by the `thenEvt` combinator is synchronized on, either all of its constituent events and functions evaluate in isolation, or none evaluates.

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

## 1.2 Isabelle/HOL

An interactive theorem proving assistant, or proof assistant, is a machine (typically software) that helps its user specify propositions and prove theorems. Like typical programming systems, it checks that propositions are lexically correct, syntactically correct, and even compositionally correct according to its type system. To determine if a proposition is valid, a proof assistant often requires the user to supply a proof. Once the proof assistant has verified that the user's proof is correct, then the proposition may be considered a theorem. In addition to checking the correctness of proofs and propositions, the proof assistant also assists the user in constructing proofs.

Isabelle/HOL is a popular proof assistant that assists in specifying and proving properties formulated in a higher order logic (HOL)[]. Predicates and functions may be higher order. That is, they may take other predicates or functions as arguments, and functions may return predicates or functions. In HOL, a predicate is actually just a function that returns a value of type `boolean`, and a proposition is simply a term of type `boolean`. Terms are deemed compositionally correct according to a system of simple types, similar to that of Standard ML. Isabelle/HOL excels at assisting with proving propositions containing numerous details.

A proof is a sequence of manipulations from axioms to the claim. Typically, unassisted informal proofs, are written in a declarative form. One first states the claim, and then starts off the proof by stating an axiom or theorem. Subsequently derived theorems follow, with or without the details of the manipulation explained. For instance, say you want to prove  $P1 \vee P2 \rightarrow Q$ . You could begin by stating the axiom  $P1 \vee P2 \vdash P1 \vee P2$ , commonly written as **assume**  $P1 \vee P2$ . You can also state  $P1 \vdash P1$  and  $P2 \vdash P2$ . Since these are intended to form complementary cases of  $P1 \vee P2$ , they are written as **case**  $P1$ : ... **case**  $P2$ . Perhaps you know **theorem**  $A$ :  $\vdash P1 \rightarrow Q$ . Using modus ponens, you derive  $P1 \vdash Q$ . Let's say you also know **theorem**  $B$ :  $\vdash P2 \rightarrow Q$ . Using modus ponens with theorem  $B$ , you derive  $P2 \vdash Q$ . These two theorems may be combined into  $P1 \vee P2 \vdash Q$ , which reduces to  $\vdash P1 \vee P2 \rightarrow Q$ .

```
 $\vdash P1 \vee P2 \rightarrow Q$ 
proof
  assume  $P1 \vee P2$ :
```

```

case P1:
  have  $\vdash P1 \rightarrow Q$  by A
  have  $\vdash Q$  by modus ponens
case P2:
  have  $\vdash P2 \rightarrow Q$  by B
  have  $\vdash Q$  by modus ponens
have  $P1 \vdash Q, P2 \vdash Q$ 
have  $\vdash Q$  by disjunction elimination
have  $P1 \vee P2 \vdash Q$ 
have  $\vdash P1 \vee P2 \rightarrow Q$  by implication introduction
qed

```

For proving simple propositions, it may be easy to conjure up the theorems and axioms needed to combine and manipulate into the goal. However, for proving complex or unfamiliar propositions, it may be less clear. In addition to the declarative forward proof style, Isabelle/HOL also supports an imperative backwards proof style, in which you start with the goal and break it into simpler and simpler subgoals until all subgoals are manipulated into axioms. The backwards style of reasoning, allows the you to focus on the manipulation rule, rather than remembering and gathering up all the theorems that are necessary to combine to reach the goal. The interface of the interactive theorem prover displays the subgoals resulting from applying each manipulation rule.

```

 $\vdash P1 \vee P2 \rightarrow Q$ 
apply (rule impI)
   $P1 \vee P2 \vdash Q$ 
apply (erule disjE)
   $P1 \vdash Q$ 
  *  $P2 \vdash Q$ 
apply (insert A)
   $P1, P1 \rightarrow Q \vdash Q$ 
  *  $P2 \vdash Q$ 
apply (erule mp)
   $P1 \vdash P1$ 
  *  $P2 \vdash Q$ 
apply assumption
   $P2 \vdash Q$ 
apply (insert B)
   $P2, P2 \rightarrow Q \vdash Q$ 
apply (erule mp)
   $P2 \vdash P2$ 
apply assumption
done

```

Note that the syntax shown here differs slightly from that of Isabelle/HOL. In order to make these examples more accesible to those unfamiliar with Isabelle/HOL, I have chosen to use a syntax that is more typical of mathematical logic and language theory literature.

In addition to creating theorems by proving propositions, Isabelle/HOL also allows creating

theorems by defining predicates and functions. This feature is critical for constructing inductive propositions that may hold over infinite ranges. Infinite ranges may be defined as inductive datatypes, similar to Standard ML. For instance, you can define the infinite set of natural numbers (with zero) as an inductive datatype, and the binary relation that a natural number is less than or equal to another as an inductive predicate.

```
datatype nat = Z | S nat

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

Additionally, you can define the infinite set of lists inductively, and the higher order predicate that checks sortedness using a supplied binary relation.

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

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

Each case of the predicate definition is considered a theorem, and you have the option to give it a name. Additionally, these are the only cases that can hold for the predicate. Therefore, the predicate applied to some free variables is equivalent to the disjunction of all the cases, with the variables equal to their respective patterns in each case. This kind of theorem, and other similar theorems for inversion and induction are created with each predicate definition.

```
theorem sorted.simps: r xs .
  ⊢ sorted r xs ≡
    (xs = Nil)
  ∨ ∃ x . (xs = (Cons x Nil))
  ∨ ∃ x y ys . (xs = (Cons x (Cons y ys)) ∧ r x y ∧ sorted r (Cons y ys))
```

By composing these definitions, you can state and prove a list of natural numbers is sorted in non-decreasing order.

```
⊢ sorted lte (Cons (Z) (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil))))
apply (rule cons)
  ⊢ lte Z (S Z)
```

```

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

```

The learning curve for using proof assistants in general and Isabelle/HOL in particular is very steep. Nevertheless, once one has gained some fluency, there are great benefits for certain kinds of projects. In a theory of Concurrent ML, with various propositions about semantics and communication, there are many tedious details that must be specified. The automatic checking of propositions and proofs is excellent at finding errors buried in numerous tedious details. Furthermore, since greater complexity follows from greater number of language features, or greater precision of propositions, it is very useful to start with simple features and propositions to create a minimal viable theory, and then incrementally increase complexity and modify proofs, accordingly. The proof assistant eases the process incremental extension by pinpointing where the proofs and propositions break as features and complexity is added.

The Isabelle/HOL formalization of this work consists of 12 theory files containing definitions, theorems, and proofs, for the syntax, dynamic semantics, static semantics, soundness of semantics, dynamic communication, static communication, soundness of communication, helper definitions, and theorems. There are roughly 1421 lines of definitions, and 3052 lines of completed proofs.



### 1.3 Static Analysis for Concurrent ML

A static analysis that describes communication topologies of channels has practical benefits in at least two ways. It can highlight which channels are candidates for optimized implementations of communication; or in a language extension allowing the specification of specialized channels, it can conservatively verify their correct usage. Without a static analysis to check the usage of the special channels, one could inadvertently use a channel intended for just one sender when really the program has multiple senders, thereby violating the intended semantics.

The utility of the static analysis depends on it being precise, sound, and computable. The analysis is precise if it describes information that isn't invariantly true for all programs. The analysis is sound if the information it describes about a program is the same or less precise than the information described by the dynamic semantics of the program. The analysis is computable if there is an algorithm that determines values sufficient for the analysis to hold from an input program.

Analyses can be described in a variety of ways. An algorithm that takes programs as input and produces information about the behavior as output is ideal for automation. A set of inference rules may be more suitable for clarity of meaning and correctness with respect to the dynamic semantics. However, inference rules can be translated into an algorithm. One rather mechanical method essentially involves specifying a reasoner associated with the rules. First, the reasoner generates a comprehensive set of data structures representing constraints from the rules' premises; then the reasoner solves the constraints.

For a subset of Concurrent ML without event combinators, Reppy and Xiao developed an efficient algorithm that determines for each channel, all possible threads that send and receive on it. The algorithm depends on each operation in the program being labeled with a program step. A sequence of program steps 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.

Reppy and Xiao's algorithm proceeds in multiple steps that produce intermediate data structures used for efficient lookup in the subsequent steps. It starts with a control flow analysis that results in multiple mappings. One mapping is from names to abstract values. Another mapping is from channel-bound names to abstract values that are sent on the respective channels. Another is from function-bound names to abstract values that are the result of the respective function applications. It constructs a control flow graph with possible paths for conditional tests and thread spawning determined directly from the atoms 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 and increases precisions. 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 steps to all possible control paths leading into the respective program steps. Using the CFA's mappings to abstract values, the algorithm determines the program steps for sends and receives per channel name. Then it uses the mapping to control paths to

determine all control paths that send or receive on each channel, from which it classifies channels as one-to-many, many-to-one, many-to-many, or one-shot.

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

## 2 Synchronization

Synchronization of sending threads and receiving threads requires determining which threads should wait, and which threads should be dispatched. The greater the information needed to determine this scheduling, the higher the performance penalty. A uniprocessor implementation of synchronization can have very little penalty. Since only one thread can make progress at a time, only one thread requests synchronization at a time, meaning the scheduler won't waste steps checking for threads competing for the same synchronization opportunity, before dispatching. A multiprocessor implementation, on the other hand, must consider that competing threads may exists, therefore perform additional checks. Additionally, there may be overhead in sharing data between processors due to memory hierarchy designs [10].

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

The following reimplementation of `Serv` is annotated to indicate the communication topologies derived from its usage. Since there are four threads that make calls to the server, the server's particular `reqCh` has four senders. Servers are created with only one thread listening for requests, so the `reqCh` of this server has just one receiver. So the server's `reqCh` is classified as many-to-one. Each application 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.

```

structure Serv : SERV =
struct

  datatype serv = S of (int * int chan) channel

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

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

end

val server = Serv.make ()

val () =
spawn (fn () => Serv.call (server, 35));
(spawn fn () =>
  Serv.call (server, 12);
  Serv.call (server, 13)

```

```
);
spawn (fn () => Serv.call (server, 81));
spawn (fn () => Serv.call (server, 44))
```

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

In a language with low-level support for concurrency, Concurrent ML could be implemented as a library, which is the case for SML/NJ [3] and MLton [21]. It could also be implemented by a compiler and runtime or interpreter. Thus, the implementations shown here can be viewed either as a library or as an intermediate representation within a compiler or interpreter presented with concrete syntax.

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

The benefits of specialization would be much more significant in multiprocessor implementations than in uniprocessor implementations. A uniprocessor implementation could avoid overhead caused by contention to acquire locks, by coupling the implementation of channels with scheduling and only scheduling the sending and receiving operations when no other pending operations have yet to start or have already finished. Reppy's implementation of Concurrent ML uses SML/NJ's first class continuations to implement scheduling and communication as one with very low overhead. In contrast, a multiprocessor implementation would allow threads to run on different processors for increased parallelism, therefore it would not be able to mandate when threads attempt synchronization relative to others without losing the parallel advantage. The cost of trying to achieve parallelism is increased overhead due to contention over acquiring synchronization rights.

## 2.1 Many-to-many Synchronization

A channel can be in one of three states. Either some threads are trying to send on it, some threads are trying to receive on it, or no threads are trying to send or receive on it. Additionally a channel is composed of a mutex lock, so that sending and receiving operations can yield to each other when updating the channel state. When multiple threads are trying to send on a channel, the channel is associated with a queue consisting of messages to be sent, along with conditions waited on by sending threads. When multiple threads are trying to receive on a channel, the channel is

associated with a queue consisting of initially empty cells that are accessible by receiving threads and conditions waited on by the receiving threads. The channel content holds one of three three potential states and their associated content of queues and conditions. The channel is composed of the channel content and also a mutex lock that regulates access to the channel content.

The sending operation acquires the channel's lock to ensure that it updates the channel based on its current state. If the channel is in the receiving state, i.e. there are threads trying to receive from the channel, then the sending operation dequeues an item from the state's associated queue. The item consists of a condition waited on by a receiving thread and an empty cell that can be accessed by the receiving thread. The sending operation deposits the message in the cell and signals on the receiving state's condition. Then, if there are no further receiving threads waiting, it updates the channel's state to inactive; otherwise, it leaves the state in the receiving state. Next, it releases the lock, signals on the receiving state's condition and returns the unit value.

If there are no threads receiving on the channel, the sending operation updates the channel state to the sending state, and enqueues a condition and the message. It releases the lock and waits on the enqueued condition. Once a receiving thread signals on the same condition, the sending operation returns with the unit value.

The receiving operation acquires the channel's lock to ensure that it updates the channel based on its current state. If there are threads sending on the channel, the receiving operation dequeues an item from the sending state's associated queue. The item consists of a condition waited on by a sending thread along with a message. The receiving operation signals on the sending state's condition. If there are no further sending threads waiting, it updates the channel's state to inactive; otherwise, it leaves the state in the sending state. Next, it releases the lock and returns the message from the sending state. If there are no sending threads on the channel, the receiving operation updates the channel state to the receiving state, and enqueues a new condition `recvCond` and an empty cell. It releases the lock and waits on the its condition `recvCond`. Once a sending thread signals on its condition, the receiving operation returns with the value deposited in its cell.

```
structure ManyToManyChan : CHANNEL =
struct

  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 (cntntRef, lock)) m =
```

```

    acquire lock;
    (case !cntntRef of
      Recv q =>
        let
          val (recvCond, msgCell) = dequeue q
        in
          msgCell := Some m;
          if (isEmpty q) then cntntRef := 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
        in
          cntntRef := Send (queue [(sendCond, m)]);
          release lock;
          wait sendCond
        end
    )

fun recv (Chn (cntntRef, lock)) =
  acquire lock;
  (case !cntntRef of
    Send q =>
      let
        val (sendCond, m) = dequeue q
      in
        if (isEmpty q) then cntntRef := Inac else ();
        release lock;
        signal sendCond;
        m
      end
    | Recv q =>
      let
        val recvCond = condition ()
        val msgCell = ref NONE
      in

```

```

        enqueue (q, (recvCond, msgCell));
        release lock;
        wait recvCond
        valOf (!msgCell)
    end
| Inac =>
    let
        val recvCond = condition ()
        val msgCell = ref NONE
    in
        cntntRef := Recv (queue [(recvCond, msgCell)]);
        release lock;
        wait recvCond
        valOf (!msgCell)
    end
)

end

```

## 2.2 One-to-many Synchronization

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

The sending operation starts by creating a condition `sendCond`, then checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to a sending state. If successful, it simply waits on its condition `sendCond`. After the receiving thread signals on `sendCond`, the sending operation returns the unit value. If the transactional update fails and the state is that of threads trying to receive on the channel, then the sending operation acquires the lock, then dequeues an item from the associated queue where the item consists of a receiving condition `recvCond`, and a cell for depositing the message to the receiving thread. If there are no further items on the queue, the sending operation updates the state to inactive; otherwise, it leaves the state in the receiving state. Next, it releases the lock it, then signals on the receiving 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 classification analysis were incorrect and there were actually multiple threads that could call the sending operation, then there might be data races. Likewise, due to the expectation of a single thread sending on the channel, the sending operation will never witness the state in the sending state, which would mean another thread is in the process of sending a message.

The receiving operation acquires the lock and checks the state of the channel, just like the receiving operation for many-to-many channels. If the channel is in a state where there is no sending thread waiting, then it updates the state to receiving, behaving the same as the receiving operation of many-to-many channels. If there is already a sending thread waiting, then it updates the state to inactive and releases the lock. Then it signals on the sending state's condition and returns the message held in the sending state.

```

structure OneToManyChan : 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 (cntntRef, lock)) m =
  let
    val sendCond = condition ()
  in
    case (cas (cntntRef, Inac, Send (sendCond, m))) of
      Inac =>
        (* cntntRef is already set to sending state by cas *)
        wait sendCond
    | Recv q =>
      let
        (*
          the current thread is the only one that
          updates from this state
        *)
        val () = acquire lock
        val (recvCond, msgCell) = dequeue q
      in
        msgCell := SOME m;
        if (isEmpty q) then cntntRef := Inac else ();
        release lock;
        signal (recvCond)
      end
    | Send _ => raise NeverHappens
  end

```



```

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

```

### 2.3 Many-to-one Synchronization

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

```

structure ManyToOneChan : 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 (cntntRef, lock)) m =
    acquire lock;
    (case !cntntRef of
      Recv (recvCond, msgCell) =>
        (
          msgCell := SOME m;
          cntntRef := 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
        cntntRef := Send (queue [(sendCond, m)]);
        release lock;
        wait sendCond
      end
    )

fun recv (Chn (cntntRef, lock)) =
let
  val recvCond = condition ()
  val msgCell = ref NONE
in
  case cas (cntntRef, Inac, Recv (recvCond, msgCell)) of
    Inac =>
      (
        (* cntntRef is already set to receiving state by cas *)
        wait recvCond;
        valOf (!msgCell)
      )
    | Send q =>
      let

```

```

    (*
      the current thread is the only one that
      updates the state from this state
    *)
    val () = acquire lock
    val (sendCond, m) = dequeue q
  in
    if case (isEmpty q) then cntntRef := Inac else ();
    release lock;
    signal sendCond;
    m
  end
| Recv _ => raise NeverHappens
end

end

```

## 2.4 One-to-one Synchronization

A one-to-one channel can also be in one of three possible states, but there is no associated lock. Additionally, none of the states is associated a queue. Instead, the potential states are that of a thread trying to send, with a condition and a message, that of a thread trying to receive with a condition and an empty cell, or the inactive state.

The sending operation creates a condition `sendCond` and checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to a sending state. If successful, it simply waits on its condition `sendCond`, then returns the unit value. If the transactional update fails and the state is a receiving state, then it deposits the message in the receiving state's associated cell, updates the channel state to inactive, then signals on the receiving state's condition and returns the unit value. If the communication analysis for the channel is truly one-to-one, then no other thread will be trying to update the state while in the receiving state, so no locks are necessary. Additionally, if the channel is truly one-on-one, the sending operation will never witness a preexisting sending state since it is running on the one and only sending thread.

The receiving operation creates a condition `recvCond` and an empty cell, then checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to the receiving state. If successful, it simply waits on its condition `recvCond`. If the transactional update fails and the state is a sending state, then it updates the channel state to inactive, then signals on the sending state's condition and returns the message held in the sending state. If the communication analysis for the channel is truly one-to-one, then no other thread will be trying trying to send, so no locks are necessary. Additionally, if the channel is truly one-to-one, the receiving operation will never witness a preexisting receiving state since it is running on the one and only receiving thread.

```

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 cntntRef) m =
  let
    val sendCond = condition ()
  in
  case cas (cntntRef, Inac, Send (sendCond, m)) of
    Inac =>
      (* cntntRef is already set to sending state by cas *)
      wait sendCond
    | Recv (recvCond, msgCell) =>
      (
        (*
          the current thread is the only one that
          accesses cntntRef for this state
        *)
        msgCell := SOME m;
        cntntRef := Inac;
        signal recvCond
      )
    | Send _ => raise NeverHappens
  end

  fun recv (Chn cntntRef) =
  let
    val recvCond = condition ()
    val msgCell = ref NONE
  in
  case cas (cntntRef, Inac, Recv (recvCond, msgCell)) of
    Inac =>
      (
        (* cntntRef is already set to receiving state by cas *)
        wait recvCond;
        valOf (!msgCell)
      )
  end

```

```

    )
  | Send (sendCond, m) =>
  (
    (*
      the current thread is the only one that
      accesses cntntRef for this state
    *)
    cntntRef := Inac;
    signal sendCond;
    m
  )
  | Recv _ => raise NeverHappens
end

end

```

## 2.5 One-shot Synchronization

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

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

The receiving operation creates a condition `recvCond` and an empty cell, then checks if the channel's state is inactive and tries to use the compare-and-swap operator to transactionally update the state of the channel to the receiving state. If successful, it simply waits on its condition `recvCond`, then returns the message deposited in its cell. If the transactional update fails and the state is a sending state, then it acquires the lock, signals on the state's associated condition and returns the message held in the sending state. It never releases the lock, blocking any additional attempts to receive, which is fine if there is truly at most one message ever sent on the channel. If the state is a receiving state, then the receiving operation attempts to acquire the lock, but it will never actually acquire it since the thread associated with the receiving state will never release it.

```

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 (cntntRef, lock)) m =
let
  val sendCond = condition ()
in
case cas (cntntRef, Inac, Send (sendCond, m)) of
  Inac =>
    (* cntntRef is already set to sending state by cas *)
    wait sendCond
  | Recv (recvCond, msgCell) =>
    (
      msgCell := SOME m;
      signal recvCond
    )
  | Send _ => raise NeverHappens
end

fun recv (Chn (cntntRef, lock)) =
let
  val recvCond = condition ()
  val msgCell = ref NONE
in
case cas (cntntRef, Inac, Recv (recvCond, msgCell)) of
  Inac =>
    (
      (* cntntRef is already set to receiving state by cas*)
      wait recvCond;
      valOf (!msgCell)
    )
  | Send (sendCond, m) =>
    (
      acquire lock;
      signal sendCond;
      (* never releases lock, so blocks others forever *)
      m
    )
  | Recv _ =>
    (
      acquire lock;
      (* never able to acquire lock, so blocked forever *)
      raise NeverHappens
    )
end

```

**end**

## 2.6 One-sync Synchronization

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, such as `replCh` in the server example. The one-sync channel is composed of a possibly empty message cell, a condition for a sending thread to wait on, and a condition for a receiving thread to wait on.

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

```
structure OneSyncChan : CHANNEL =  
struct  
  
  datatype 'a channel =  
    Chn of condition * condition * 'a option ref  
  
  fun channel () =  
    Chn (condition (), condition (), ref NONE)  
  
  fun send (Chn (sendCond, recvCond, msgCell)) m =  
    (  
      msgCell := SOME m;  
      signal recvCond;  
      wait sendCond  
    )  
  
  fun recv (Chn (sendCond, recvCond, msgCell)) =  
    (  
      wait recvCond;  
      signal sendCond;  
      valOf (!msgCell)  
    )  
  
end
```

## 2.7 Discussion

The example implementations of generic synchronization and specialized synchronization suggest that cost savings of specialized implementation are significant. For example, if you know that a channel has at most one sending thread and one receiving thread, then you will lower synchronization costs by using an implementation that is specialized for one-to-one communication.

To be certain that the new program with the specialized implementation behaves the same as the original program with the generic implementation, you need to be certain that the specialized program behaves the same, assuming one-to-one or less communication, that static classification as one-to-one 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.

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

### 3 Mechanized Theory

The definitions and theorems of this work were constructed in the formal language of Isabelle/HOL, to enable mechanical verification of the correctness of the proofs. However, in this presentation, the syntax of the stated definitions and propositions will differ from the actual Isabelle/HOL syntax, in order to be more intuitive for those unfamiliar with Isabelle/HOL.

A static analysis for detecting communication patterns in Concurrent ML would be necessary for useful optimizations or checks. The optimization to replace general synchronization with specialized synchronization based on the results of static analysis is one that could have significant performance benefit, as shown empirically by Reppy and Xiao []. However, if the static analysis of communication patterns is wrong, then the optimization could produce an incorrect program. What you need is proof that the static analysis is sound with to the actual communication patterns that occur when running a program. However, the proof could also be erroneous, and with so many details could be extremely difficult for a human to manually check for errors. Isabelle/HOL can automatically check for errors in proofs and specifications and is much more reliable than a human. By using an optimization without a soundness proof, you are trusting that it is sound. If soundness has been proved, you are trusting that the proof has no errors. If the proof has been verified by Isabelle/HOL, you are trusting that the Isabelle/HOL kernel is sound. The less you need to trust the better.

I extend Reppy and Xiao's work by developing a mechanically checked specification and proofs for communication topologies in Concurrent ML. A proof checker requires that manipulations are constructed from a small kernel of primitive operations. It's possible that an intuitive step of reasoning in an informal proof could require multiple tedious steps of precise manipulation in a mechanically checked proof. To avoid the chance of running into these tricky scenarios, I specified the static analysis as inference rules, rather than as an algorithm. In Isabelle/HOL, algorithms are specified as computable functions that must be proved to terminate and be total, either automatically or with additional details supplied by the user. Reppy and Xiao's algorithm,



as with many static analysis algorithms, relies on accumulating values in a growing set, with each recursive call until reaching a fixpoint. In Isabelle/HOL it is easy to prove termination for functions that branch on an inductive datatype, however it is far from straight forward to prove termination for those that branch on whether or not a fixpoint has been reached.

Concurrent ML separates itself from other languages for concurrency with its generalized concept of events, event synchronization, and event combinators. Reppy and Xiao's work produces a static analysis for a subset, which only contained synchronization on sending and receiving events. I originally, wanted to extend Reppy and Xiao's analysis to encompass a more generalized notion of events, along with the event combinator for choice. However, I quickly realized that the combination of creating a new specification, in an unfamiliar proof assistant, with a more complex semantics in the language, would be too much to detail with at once. Instead, I adhered to Reppy and Xiao's decision to construct the specification and proofs for a simplified subset of Concurrent ML, containing synchronization on just sending and receiving events. Since I constructed these specifications and proofs in Isabelle/HOL, it will be possible to easily extend the semantics later, by using the proof assistant to pinpoint where the proofs and specifications break upon changes.

The language for the mechanized specification contains features found in untyped lambda calculus with some extensions, such as functions, function application, products, product elimination, sums, sum elimination, and unit. Additionally, for concurrency it contains channel creation, sending events, receiving events, synchronization, and thread spawning. Features may be used together by binding the use of one feature to a name, and then sequencing to another term using that name. Additionally, spawning, function application, and sum elimination require use of other features.

The dynamic evaluation is represented by a small step relation for a variant of a CEK machine, in which one pool evaluates to the next pool. A pool consists of many states associated with paths taken to reach each state. A state consists of a term to be evaluated, an environment for looking up values from names, and a stack of continuations. A valid small step relation always steps from one pool to the pool extended with at least one new state. Terms containing names are not simplified. Instead, an environment can be used to look the values of names. Names are locally scoped so that they can be reused with a precise meaning, such as function parameter being bound to different argument each time its called. As such, for any term containing names, its corresponding value is simply the term paired with its own environment.

The initial pool of a running program is a pool with an empty path associated with a state containing the program, an empty environment and an empty continuation stack. Full evaluation is represented by the star predicate composed with the small step relation, applied to the initial pool. It states that an initial pool takes zero or more transitive steps to reach another pool.

The static evaluation is represented by a control flow analysis relation. Control flow analysis is a "may" analysis, meaning that the analysis is sound if it holds for all possible evaluations of the dynamic evaluation. It may also hold for evaluations that are impossible to occur in the dynamic evaluation. Instead of associating each term with its own environment, the control flow analysis simply has one static environment for the entire program. If a name takes on different values in

different scopes during dynamic evaluation, then the static environment simply considers both to be options.

The soundness theorem of the static evaluation with respect to dynamic evaluation states that if full dynamic evaluation of a program reaches a pool with containing some environment, and the static evaluation holds for the program and some static environment, then the static environment is an abstraction of the dynamic environment. That is, if a name maps to some value in the dynamic environment, then, in the static environment, the name maps to a set containing the same value or an abstraction of the value. The static evaluation relates static environments to initial programs, but soundness relates static environments to dynamic environments. Furthermore, the dynamic environment of a state in a pool depends on, the terms, environments, and continuation stacks of states in previous pools. Thus, if the static evaluation holds for a static environment and a program, then there must be a relation that holds for the same static environment and any pool that the program may dynamically evaluate to. Generalizing static evaluation to work for pools, environments, and stacks, gives a version that does not lose precision. The generalized static evaluation on the initial pool is defined to follow from the static evaluation on the program. The preservation of the static evaluation of a pool across multiple steps of dynamic evaluation is proved by induction on the star relation. The static evaluation on a pool is taken apart into a static evaluation on a dynamic environment, thus relating the dynamic environment with the static environment, which is essentially sound by definition.

The dynamic communication classification consists of four relations to categorize channels in a pool, as one-to-many, many-to-one, one-to-one, one-shot, or one-sync. All four classification relations are defined in terms of how any two paths in the pool relate to each other. The one-to-many classification holds if any two paths that send on the channel are ordered. If the sending paths are all ordered or there are no sending paths, then there is simply only at most one thread that sends on the channel. Likewise, The many-to-one classification holds if any two paths that receive on the channel are ordered. The one-to-one classification holds if both all the receiving paths on the channel are ordered and all the sending paths on the channel are ordered. The one-shot classification holds if there is at most one sending path on a channel, regardless of which paths receive on the channel. The one-sync classification holds if there is at most one sending and one receiving path on a channel.

The static communication classification relations, as with dynamic communication, are defined in terms of paths that send and receive a channel of interest. However, these sites of sending and receiving the channel, are approximations taken from the static environment, whose content is influenced by the rules of static evaluation. The static flows acceptance relation states that a set of flows, i.e. a graph, accepts a program or term, where each flow is an abstract representation of a small step in the dynamic evaluation. The relation is defined such that it is decidable if a given finite set of flows holds. The static paths are formed from the chaining of flows, according to the rules of the static traceable relation. The set of static paths to a sending or receiving synchronization site is bounded by the static flows acceptance relation. Due to cycles in the graph, the set of paths may be infinite. The static one-to-many classification holds if any two static sending paths are noncompetitive, meaning that either they are ordered, like the dynamic classification,

or there's no way for the paths to occur in the same run of the program. The many-to-one and one-to-one classification also rely on the notion of noncompetitive paths, rather than simply unordered paths. Likewise, the static one-shot classification holds if any two static sending paths are singular, meaning that either they're equal, like the dynamic classification, or they cannot occur in the same run of the program.

The soundness theorem of the static communication classification relations to their dynamic counterparts states that if a channel is statically classified in some way, then it is dynamically classified the same way. For instance, if the static one-to-many classification holds for a static channel, then the dynamic one-to-many classification holds for all dynamic counterparts of that channel. The static communication classification is a "must" analysis, meaning that if the classification holds statically, then it must hold dynamically. Soundness depends on the soundness and preservation of its constituent parts, including the soundness of static evaluation, the soundness of the noncompetitive relation with respect to the unordered relation, the soundness of the singular relation with respect to equality of paths, and the soundness and preservation of the static traceable relation. As with the static evaluation, many proofs of soundness for the static communication classification, rely on bridging the gap between the terms used in dynamic versions and the terms used in the static versions. Generalizations of the static communication classification relations to apply to dynamic terms, such as pools, environments, stacks, and values are derived to complete these proofs.

This work does not contain formal proofs that the specialized implementations are behaviorally equivalent to a generic implementation, but the example implementations should provide good evidence for that.

### 3.1 Syntax

The syntax used in this formal theory contains a very small subset of *Concurrent ML*'s features. The features include recursive with application, left and right construction with pattern matching, pair construction with first and second selections, sending and receiving event construction with synchronization, channel creation, thread spawning, and the unit literal. The syntax is defined in a way to make it possible to relate the dynamic semantics of programs to the syntax programs. The syntax is defined in administrative normal form (ANF) [6], in which every term is bound to a name. Furthermore, terms only accept names in place of eagerly evaluated inputs.

Restricting the grammar to ANF allows the operational semantics to maintain graph information by associating values with succinct names. Maintaining the values' ties to the syntax, simplifies proofs of soundness, since they must relate dynamic evaluation information to static information based on the syntax.

Additionally, ANF melds nicely with the semantics of control paths, which succinctly identify the evaluation taken to reach some intermediate result. Instead of relying on additional meta-syntax to associate atom operations with identifiers, the analysis can simply use the required names of ANF syntax to identify locations in the program.

The ANF syntax is impractical for a programmer to write, yet it is still practical for a language

under automated analysis since there is a straightforward procedure to transform more user-friendly syntax into ANF.

```

datatype name = Nm string

datatype term =
  Bind name complex term
| Rslt name

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

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

```

### 3.2 Dynamic Semantics

The dynamic semantics describes how programs evaluate to values. A history of execution is represented as a list of steps, where a step is a binding name or resulting name of a term, paired with a mode of control indicating flows by sequencing, spawning, calling, or returning. Channels have no literal representation, but each time a channel is created, it is uniquely identified by the history of the execution up until the step of creation. Atomic terms are not simplified. Instead, atoms are evaluated to closures consisting of the atom syntax, along with an environment that maps its constituent names to their values.

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

The semantics are defined as a CEK machine, rather than a substitution based operational semantics. By avoiding simplification of terms in the operational semantics, it is possible to relate the abstract evaluations of the static semantics to the evaluations produced by the dynamic semantics, which in turn is relied on to prove soundness of the static semantics.

```

datatype dynamic_step =

```

```

    DNxt name
  | DSpwn name
  | DCll name
  | DRtn name

type dynamic_path = dynamic_step list

datatype chan =
  Chan dynamic_path name

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

type environment =
  name -> dynamic_value option

```

The evaluation of some complex terms results in sequencing, meaning there is no coordination with other threads, and there is no need to save terms on the continuation stack for later evaluation. These terms are the unit literal, atoms, pairs, and first and second selections. The evaluation depends only on the syntax and an environment for looking up the values of names within the syntax. Additionally, all these terms evaluate to values in a single step.

```

predicate seqEval:
  complex -> environment -> dynamic_value -> bool where
  unit: env .
  ⊢ seqEval Unt env VUnt
* atom: a env .
  ⊢ seqEval (Atom a) env (VAtm a env)
* first: env np n1 n2 envp v .
  env np = Some (VAtm (Pair n1 n2) envp),
  envp n1 = Some v
  ⊢ seqEval (Fst np) env v
* second: env np n1 n2 envp v .
  env np = Some (VAtm (Pair n1 n2) envp),
  envp n2 = Some v
  ⊢ seqEval (Snd np) env v

```

The evaluation of a complex term for application or conditional testing results in flowing by calling. A calling flows is characterized by the need to save a subterm in the continuation stack for later evaluation. The evaluation depends on the syntax and an environment for looking up the values of names within the syntax. A term is evaluated to a subterm, and a new environment that will later be used in the evaluation of the subterm. For conditional testing, either the left or the right term is called, and the environment is updated with the corresponding name mapped to the value extracted from the pattern. For application, the term inside of an applied function is

called, and the environment is updated with the function's parameter mapped to the application's argument. The environment is also updated with the recursive name mapped to the same applied function.

```
predicate callEval: complex -> env -> term -> env -> bool where
  caseLeft: env ns nc envs v nl tl nr tr .
    env ns = Some (VAtm (Lft nc) envs),
    envs nc = Some v
    ⊢ callEval (Case ns nl tl nr tr) env tl (env(nl -> v))
  * caseRight: env ns nc envs v nl tl nr tr .
    env ns = Some (VAtm (Rht nc) envs),
    envs nc = Some v
    ⊢ callEval (Case ns nl tl nr tr) env tr (env(nr -> v))
  * app: env nf nf' np tb envf na v .
    env nf = Some (VAtm (Fun nf' np tb) envf),
    env na = Some v
    ⊢ callEval
      (App nf na) env tb
      (envf(
        nf' -> (VAtm (Fun nf' np tb) envf),
        np -> v
      ))
```

The continuation stack maintains a record of terms that should be evaluated once a corresponding called branch of the evaluation has returned. Each continuation in the stack consists of a term, the environment for resolving the term's names, an unresolved name, to be resolved when the corresponding branch returns. The initial state of execution consists of a program, an empty environment, and an empty stack of continuations. With each sequential step, the program is reduced to a subterm, and the environment is updated with the name bound to the value of the term. Each time a embedded term is sidestepped to evaluate a dependent term, a continuation is formed around it and pushed onto a stack of continuations. A continuation is popped off the stack when a state's program is reduces to a result program. A pool of states keeps track of all the states that have been reached through the evaluation of an initial program. Each state is indexed by the dynamic path taken to reach it. A pool's leaf path indicates a state that has yet to be evaluated. Additionally, the communication between threads is also recorded as a set of correspondences consisting of the path to the sending state, the path to the receiving state, and the channel used for communication.

```
datatype contin = Ctn name tm env
```

```
type stack = contin list
```

```
datatype state =
  Stt program env stack
```

```
type pool =
```

```

dynamic_path -> state option

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

type corresp = dynamic_path * chan * dynamic_path

type communication = corresp set

```

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

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

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

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

In the case of spawning, the evaluation is updated with two new paths extending the leaf path. For one, the leaf path is extended with a sequential program step whose state has the next term and the environment updated with the unit value bound to the bind name, and the original continuation stack. For the other, the leaf path extended with an program step indicating a spawning flow. Its state has the spawned term, the original environment, and an empty continuation stack.

In the case where two leaf paths in the pool correspond to synchronization on the same channel, and one synchronizes on a send event and the other synchronizes on a receive event, then

evaluation updates the pool with two new paths and corresponding states. It updates the send event's state with its embedded term, the environment updates with the unit value, and the stack unchanged. It updates the receive event's state with its embedded term, the environment updates with the sent value, and the stack unchanged. Additionally, the communication is updated with the sending and receiving paths, and the channel that the synchronization used for communication.

```

predicate dynamicEval:
  pool -> communication -> pool -> communication -> bool
where
  return: pool path n env nk tk envk stack' v comm .
    leaf pool path,
    pool path = Some (Stt (Rslt n) env ((Ctn nk tk envk) # stack')),
    env n = Some v
  ⊢ dynamicEval
    pool comm
    (pool(
      path @ [DRtn n] ->
        (Stt tk envk(nk -> v) stack')
    ))
    comm
  * seq: pool path n c t' env stack v .
    leaf pool path,
    pool path = Some (Stt (Bind n c t') env stack),
    seqEval c env v
  ⊢ dynamicEval
    pool comm
    (pool(
      path @ [DNxt n] -> (Stt t' (env(n -> v)) stack)
    ))
    comm
  * call: pool path n c t' env stack tc envc comm .
    leaf pool path,
    pool path = Some (Stt (Bind n c t') env stack),
    callEval c env tc envc
  ⊢ dynamicEval
    pool comm
    (pool(
      path @ [DCll n] -> (Stt tc envc ((Ctn n t' env) # stack)
    ))
    comm
  * makeChan: pool path n t' env stack .
    leaf pool path,
    pool path = Some (Stt (Bind n MkChn t') env stack)
  ⊢ dynamicEval
    pool comm

```



```

    (pool(
      path @ [DNxt n] ->
        (Stt t' (env(n -> (VChn (Chan path n)))) stack)
    ))
    comm
* spawn: pool path n tc t' env stack comm .
    leaf pool path,
    pool path = Some (Stt (Bind n (Spwn tc) t') env stack)
⊢ dynamicEval
    pool comm
    (pool(
      path @ [DNxt n] -> (Stt t' (env(n -> VUnt)) stack),
      path @ [DSpwn n] -> (Stt tc env [])
    ))
    comm
* sync: pool paths path ns nse ts envs stacks nsc nm
    envse pathr nr nre tr envr stackr nrc envre c comm .
    leaf pool paths,
    pool paths = Some
      (Stt (Bind ns (Sync nse) ts) envs stacks),
    envs nse = Some
      (VAtm (SendEvt nsc nm) envse),
    leaf pool pathr,
    pool pathr = Some
      (Stt (Bind nr (Sync nre) tr) envr stackr),
    envr nre = Some
      (VAtm (RecvEvt nrc) envre),
    envse nsc = Some (VChn c),
    envre nrc = Some (VChn c),
    envse nm = Some vm
⊢ dynamicEval
    pool comm
    (pool(
      paths @ [DNxt ns] -> (Stt ts (envs(ns -> VUnt)) stacks),
      pathr @ [DNxt nr] -> (Stt tr (envr(nr -> vm)) stackr)
    ))
    (comm ∪ {(paths, c, pathr)})

```

### 3.3 Dynamic Communication

The dynamic one shot classification describes pools where there is only one dynamic path that synchronizes and sends on a given channel. Whether or not two threads compete to synchronize on a channel can be determined by looking at the paths of the pool. If two paths are ordered, that is, one is the prefix of the other or vice versa, then the shorter path synchronizes before the longer path. Two ordered paths either indicates that the two paths occur in the same thread, or

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

```

predicate isSendPath: pool -> chan -> dynamic_path -> bool where
  intro: pool path n ne t' env stack nsc nm enve c .
    pool path = Some (Stt (Bind n (Sync ne) t') env stack),
    env ne = Some (VAtm (SendEvt nsc nm) enve),
    enve nsc = Some (VChn c)
  ⊢ isSendPath pool c path

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

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

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

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

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

```

```

    forEveryTwo (isRecvPath pool c) ordered
  ⊢ manyToOne t0 c

```

```

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

```

```

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

```

```

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

```

### 3.4 Static Semantics

The static semantics describes an estimation of the intermediate static values and embedded terms that might result from running a program. Although the estimations are imprecise with respect to the dynamic semantics, they are certainly accurate, which is confirmed by the formal proofs of soundness. The static semantics enable deduction of static information about channels and events, which is crucial for statically deducing information about synchronization on channels and communication classification. The static values consist of the static unit value, static channels, and static atom values. The static unit value is no less precise than the dynamic unit value, but static channels and static atom values are imprecise versions of their dynamic counterparts. A static channel is identified only by the name it binds to at creation time, rather than the full path that leads up to its creation. A static atom value is simply an atomic term without an environment for looking up its named arguments. A static environment contains the internal evaluation results by associating names to multiple potential static values. Thus, in addition to some static values being imprecise, the results of evaluation may be decrease precision even further by containinng multiple potential static values. In order to find the return value of a program term, it is useful to fetch the name embedded within its eventual result term, which is formally defined by `resultName`.

```

datatype static_value =
  SUnit

```

```

| SChn name
| SAtm atom

type static_value_map =
  name -> static_value set

fun resultName of term -> name: where
  n .
  ⊢ resultName (Rslt n) = n
* n c t' .
  ⊢ resultName (Bind n c t') = (resultName t)

```

The static evaluation is a control flow analysis (0CFA) that describes a relation between a program term and two static environments. The first static environment contains binding names associated with the evaluations of terms that are bound to those names in the program. The second static environment contains names of channels associated with values that might be sent over channels identified by those names.

The definition of static evaluation is syntax-directed, meaning the proof of a static evaluation is defined to be structurally inductive following the self-similar structure of the syntax. Thus, it should be possible to decide if a static evaluation holds by unraveling the program term into smaller and smaller terms, until reaching a term without any smaller terms. Additionally, for any given program, there should be instances of static environments, such that the static evaluation holds, in which case, using well known methods [], there is likely an algorithm to compute the static environments from a program, by following the basic structure of the definitional proof of static evaluation. This certainly appears likely, but it has not been formally proven in this work.

The static evaluation relation is defined in a single definition. The definition is fairly uniform and mimics the structure of the syntax. The static evaluation for each syntactic form is very similar. For instance, if a term has an embedded term, the static evaluation of the term is defined by the static evaluation of its embedded term, whether the original term is a spawning term, a function term, or a conditional test term. In contrast, in the definition of dynamic evaluation, the evaluation of certain syntactic forms is more similar to some forms than others. Conditional test terms are evaluated similarly to application terms. They both require saving some terms on the continuation stack, while evaluating other embedded terms. Function terms are dynamically evaluated similarly to other atomic terms. Additionally, the static evaluation has only a single global environment for looking up values of names in the whole program, whereas the dynamic evaluation associates local environments with different terms, in the program, allowing the same names to resolve to different values depending on the context. Therefore, the static evaluation is less precise.

```

predicate staticEval:
  static_value_map -> static_value_map -> term -> bool
where
  result: staticEnv staticComm n .
  ⊢ staticEval staticEnv staticComm (Rslt n)

```

```

* unit: staticEnv n staticComm t' .
  SUnt ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n Unt t')
* makeChan: n staticEnv staticComm t' .
  (SChn n) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n MkChn t')
* sendEvt: nc nm staticEnv n staticComm t' .
  (SAtm (SendEvt nc nm)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (SendEvt nc nm)) t')
* recvEvt: nc staticEnv n staticComm t' .
  (SAtm (RecvEvt nc)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (RecvEvt nc)) t')
* pair: n1 n2 staticEnv n staticComm t' .
  (SAtm (Pair n1 n2)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Pair n1 n2)) t')
* left: ns staticEnv n staticComm t' .
  (SAtm(Lft ns)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Lft ns)) t')
* right: ns staticEnv n staticComm t' .
  (SAtm(Rht ns)) ∈ staticEnv n,
  staticEval staticEnv staticComm t
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Rht ns)) t')
* function: nf nt tb staticEnv staticComm n t' .
  (SAtm (Fun nf nt tb)) ∈ staticEnv nf,
  staticEval staticEnv staticComm tb,
  (SAtm (Fun nf nt tb)) ∈ staticEnv n,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Atom (Fun nf nt tb)) t')
* spawn: nf nt tb staticEnv staticComm n t' .
  SUnt ∈ staticEnv n,
  staticEval staticEnv staticComm tc,
  staticEval staticEnv staticComm t'
  ⊢ staticEval staticEnv staticComm (Bind n (Spwn tc) t')
* sync: staticEnv ne n staticComm t' .
  ∀ nsc nm nc .
    (SAtm (SendEvt nsc nm)) ∈ staticEnv ne
    → SChn nc ∈ staticEnv nsc
    → SUnt ∈ staticEnv n, staticEnv nm ⊆ staticComm nc,
  ∀ nrc nc .
    (SAtm (RecvEvt nrc)) ∈ staticEnv ne

```

$$\begin{aligned}
& \rightarrow \text{SChn } n_c \in \text{staticEnv } n_{rc} \\
& \rightarrow \text{staticComm } n_c \subseteq \text{staticEnv } n, \\
& \text{staticEval staticEnv staticComm } t' \\
& \vdash \text{staticEval staticEnv staticComm (Bind } n \text{ (Sync } n_e) \text{ } t') \\
* \text{ first: staticEnv } n_t \text{ } n \text{ staticComm } t' . \\
& \quad \forall n_1 \ n_2 . \\
& \quad (\text{SAtm (Pair } n_1 \ n_2)) \in \text{staticEnv } n_t \\
& \quad \rightarrow \text{staticEnv } n_1 \subseteq \text{staticEnv } n, \\
& \quad \text{staticEval staticEnv staticComm } t' \\
& \vdash \text{staticEval staticEnv staticComm (Bind } n \text{ (Fst } n_t) \text{ } t') \\
* \text{ second: staticEnv } n_t \text{ } n \text{ staticComm } t' . \\
& \quad \forall n_1 \ n_2 . \\
& \quad (\text{SAtm (Pair } n_1 \ n_2)) \in \text{staticEnv } n_t \\
& \quad \rightarrow \text{staticEnv } n_2 \subseteq \text{staticEnv } n, \\
& \quad \text{staticEval staticEnv staticComm } t' \\
& \vdash \text{staticEval staticEnv staticComm (Bind } n \text{ (Snd } n_t) \text{ } t') \\
* \text{ caseEither: staticEnv } n_s \ n_l \ t_l \ n \text{ staticComm } n_r \ t_r \ t' . \\
& \quad \forall n_c . \\
& \quad (\text{SAtm (Lft } n_c)) \in \text{staticEnv } n_s \\
& \quad \rightarrow \text{staticEnv } n_c \subseteq \text{staticEnv } n_l, \\
& \quad \text{staticEnv (resultName } t_l) \subseteq \text{staticEnv } n, \\
& \quad \text{staticEval staticEnv staticComm } t_l, \\
& \quad \forall n_c . \\
& \quad (\text{SAtm (Rht } n_c)) \in \text{staticEnv } n_s \\
& \quad \rightarrow \text{staticEnv } n_c \subseteq \text{staticEnv } n_r, \\
& \quad \text{staticEnv (resultName } t_r) \subseteq \text{staticEnv } n, \\
& \quad \text{staticEval staticEnv staticComm } t_r, \\
& \quad \text{staticEval staticEnv staticComm } t' \\
& \vdash \text{staticEval staticEnv staticComm (Bind } n \text{ (Case } n_s \ n_l \ t_l \ n_r \ t_r) \text{ } t') \\
* \text{ app: staticEnv } n_f \ n_a \ n \text{ staticComm } t' . \\
& \quad \forall n_f' \ n_t \ t_b . \\
& \quad (\text{SAtm (Fun } n_f' \ n_t \ t_b)) \in \text{staticEnv } n_f \\
& \quad \rightarrow \text{staticEnv } n_a \subseteq \text{staticEnv } n_t, \\
& \quad \text{staticEnv (resultName } t_b) \subseteq \text{staticEnv } n, \\
& \quad \text{staticEval staticEnv staticComm } t' \\
& \vdash \text{staticEval staticEnv staticComm (Bind } n \text{ (App } n_f \ n_a) \text{ } t')
\end{aligned}$$

It is straightforward to follow the rules of static evaluation in order to build up functions mapping names to static values for the static environment and the static communication. Recasting the example server implementation into the ANF syntax demonstrates this informal procedure. The unit value, and left and right sum constructors are used to represent natural numbers.

```

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

```

```

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

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

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

```

Let's see how an informal procedure can produce the static environments by following the structure of the definitional proof structure of static evaluation. We start at the top of the program and pick the rule from the definition of static evaluation that might hold true for the current syntactic form. Then we choose the smallest environment that satisfies that rule's conditions. In the server implementation, the program starts with **bind**  $u1 = \text{unt}$  in ..., which only unifies with the rule concluding with `staticEval staticEnv staticComm (Bind n Unt ...)`, with  $n = (\text{Nm } "u1")$ . The conditions for that rule require  $\text{SUnt} \in \text{staticEnv } (\text{Nm } "u1")$ , `staticEval staticEnv staticComm ...`. We choose the smallest static environment `staticEnv`, for which  $\text{SUnt} \in \text{staticEnv } (\text{Nm } "u1")$  holds, and that happens to be  $\lambda n . \text{if } n = (\text{Nm } "u1") \text{ then } \{\text{SUnt}\} \text{ else } \{\}$ . Since there's no condition that directly states what's required of the static communication, we can simply choose an empty environment to start with. The second condition is static evaluation on a smaller term, which indicates that we should repeat this procedure again for the remainder of the program, incrementally adding more static values for each binding name in the program. We continually repeat this procedure from the top of the program until there's nothing more we can add to the static environments. The rule for synchronization is the only rule in which there are conditions on the static communication environment. So we will only add to the static communication environment when we encounter synchronization terms. The following static environments result from following this informal procedure on the example ANF server implementation. To make the presentation clear, the syntactic sugar  $(r1 \rightarrow \{\text{rht } u1\}, \dots)$  is used to mean  $\lambda n . \text{if } n = (\text{Nm } "r1") \text{ then } \{\text{SAtm } (\text{Rht } (\text{Nm } "u1"))\} \text{ else } \dots \text{ else } \{\}$ . The representation of static values closely resembles the concrete syntax for complex terms.

```
val serverStaticEnv: name -> static_value set =
(
  u1 -> {unt},
  r1 -> {rht u1},
  l1 -> {lft r1},
  l2 -> {lft l1},
  mksr -> {fun _ x2 => ...},
  x2 -> {unt},
  k1 -> {chn k1},
  srv -> {fun srv' x3 => ...},
  srv' -> {fun srv' x3 => ...},
  x3 -> {rht u1, lft r1, lft l1},
  e1 -> {recvEvt k1},
  p1 -> {pair v2 k4},
  v1 -> {lft r1, lft l1},
  k2 -> {chn k4},
  e2 -> {sendEvt k2 x3},
  z5 -> {unt},
  z7 -> {unt},
  u5 -> {unt},
  rqst -> {fun _ x4 => ...},
  x4 -> {pair srvr l1, pair srvr l2},
```



```

k3 -> {chn k1},
v2 -> {lft r1, lft l1},
k4 -> {chn k4},
p2 -> {pair v2 k4},
e3 -> {sendEvt k3 p2},
z9 -> {unt},
e4 -> {recvEvt k4},
v3 -> {rht u1, lft r1, lft r2},
svr -> {chn k1},
z10 -> {unt},
p3 -> {pair svr l1},
z11 -> {rht u1, lft r2},
p4 -> {pair svr l2},
z12 -> {rht u1, lft l1}
)

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

```

The static reachability describes terms that might be reachable from larger terms, during dynamic evaluation. A sound approximation for dynamically reachable terms are terms that are transitively embedded within larger terms. A term is statically reachable from itself, and an initial term can statically reach any term that its embedded terms can statically reach.

```

predicate staticReachable: term -> term -> bool where
  refl: t .
  ⊢ staticReachable t t
* spawn: tc tz n t' .
  staticReachable tc tz
  ⊢ staticReachable (Bind n (Spwn tc) t') tz
* caseLeft: tl tz n ns nl nr tr t' .
  staticReachable tl tz
  ⊢ staticReachable (Bind n (Case ns nl tl nr tr) t') tz
* caseRight: tr tz n ns nl tl nr t' .
  staticReachable tr tz
  ⊢ staticReachable (Bind n (Case ns nl tl nr tr) t') tz
* function: tb tz n nf nt tb t' .
  staticReachable tb tz
  ⊢ staticReachable (Bind n (Atom (Fun nf nt tb)) t') tz
* seq: t' tz n c .
  staticReachable t' tz
  ⊢ staticReachable (Bind n c t') tz

```

### 3.5 Static Communication

To describe communication statically, it is helpful to identify each term with a short description. The term identifier of a binding term is the binding name, and indication of its use in a binding. The term identifier of a result term is the embedded name, and indication of its use in a result.

```

datatype termId =
  NBnd name
| NRslt name

fun termId: term -> termId where
  n c t' .
  ⊢ termId (Bind n c t') = NBnd n
* n .
  ⊢ termId (Rslt n) = NRslt n

type term_id_map = termId -> name set

```

The static communication describes a sound approximation of the static paths that communicate on static channels. The static sending identifier classification means that a term identifier might represent a synchronization to send on a given static channel. The static receiving identifier classification means that a term identifier might represent a synchronization to receive on a given abstract channel.

```

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

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

```

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

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

The definitions are purposely structured to allow adding live channel analysis to the definition fairly straightforward with just a few alterations. Section 4 expands on these alterations and outlines a strategy that is likely to result in formal proofs of soundness, although the actual formal proof of the version with live channel analysis is not yet complete.

For the lower precision version without the liveness of channels, there are four modes - sequencing, calling, spawning, and returning - indicating how the term identifier flows to the term identifier of one of its embedded terms. A flow is a triplet of a term identifier, a mode of flow, and a term identifier of an embedded term. A static step is just a term identifier along with the mode it uses to flow to its embedded term. A static path is a list of static steps.

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

type flow = termId * mode * termId

type graph = flow set

type static_step = termId * mode

type static_path = static_step list
```

The evaluation of a term results in the flow to new terms, via sequencing, calling, returning, or spawning. These flows are represented concretely by the term identifiers of the starting and ending terms and a mode to represent the nature of the flow. The static acceptance by flows describes a set of flows consisting of all the flows that could be traversed during a program’s evaluation. It depends on the static environment for name bindings. For a result term, there are no demands on the flow graph. For all bind terms, except those binding to case matching and function application, the sequential flow from the top term to the sequenced term might accept the term, and the accepting flows are also the accepting flows for the sequenced term. For binding to a function, the accepting flows are also accepting flows for the body of the function. For binding to thread spawning, the spawning flow from the top term to the spawned term might be traversed, and the accepting flows are also accepting flows for the spawned term.

In the case of conditional testing, the calling flow from the conditional testing term to its left case’s embedded term might accept the conditional testing term, and the calling flow from the a term to the right case’s embedded term might accept the conditional testing term. The returning flow from the result of the left case’s embedded term to the sequenced embedded term might accept the result term, and the returning flow from the result of the right embedded term to the

sequenced embedded term might also accept the result term. Additionally, the accepting flows for a term are also accepting flows for its left case's embedded term, right case's embedded term, and the sequenced embedded term.

In the case of application, if the applied name is actually bound to a function, then a calling flow from the application term to the function's embedded term might accept the term, and the returning flow from the result of the function to the sequenced embedded term might accept the term. Additionally, the accepting flows for the application term are also accepting flows for the sequenced embedded term.

```

predicate staticFlowsAccept:
  static_value_map -> graph -> term -> bool
where
  result: staticEnv graph n .
  ⊢ staticFlowsAccept staticEnv graph (Rslt n)
* unit: n t' graph staticEnv .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n Unt t')
* makeChan: n t' graph staticEnv .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n MkChn t')
* sendEvt: n t' graph staticEnv nc nm .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept
    staticEnv graph
    (Bind n (Atom (SendEvt nc nm)) t')
* recvEvt: n t' graph staticEnv nc .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Atom (RecvEvt nc)) t')
* pair: n t' graph staticEnv n1 n2 .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Atom (Pair n1 n2)) t')
* left: n t' graph staticEnv ns .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Atom (Lft ns)) t')
* right: n t' graph staticEnv ns .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Atom (Rht ns)) t')
* function: n t' graph staticEnv tb nf nt .
  (NBnd n , MNxt, termId t') ∈ graph,

```

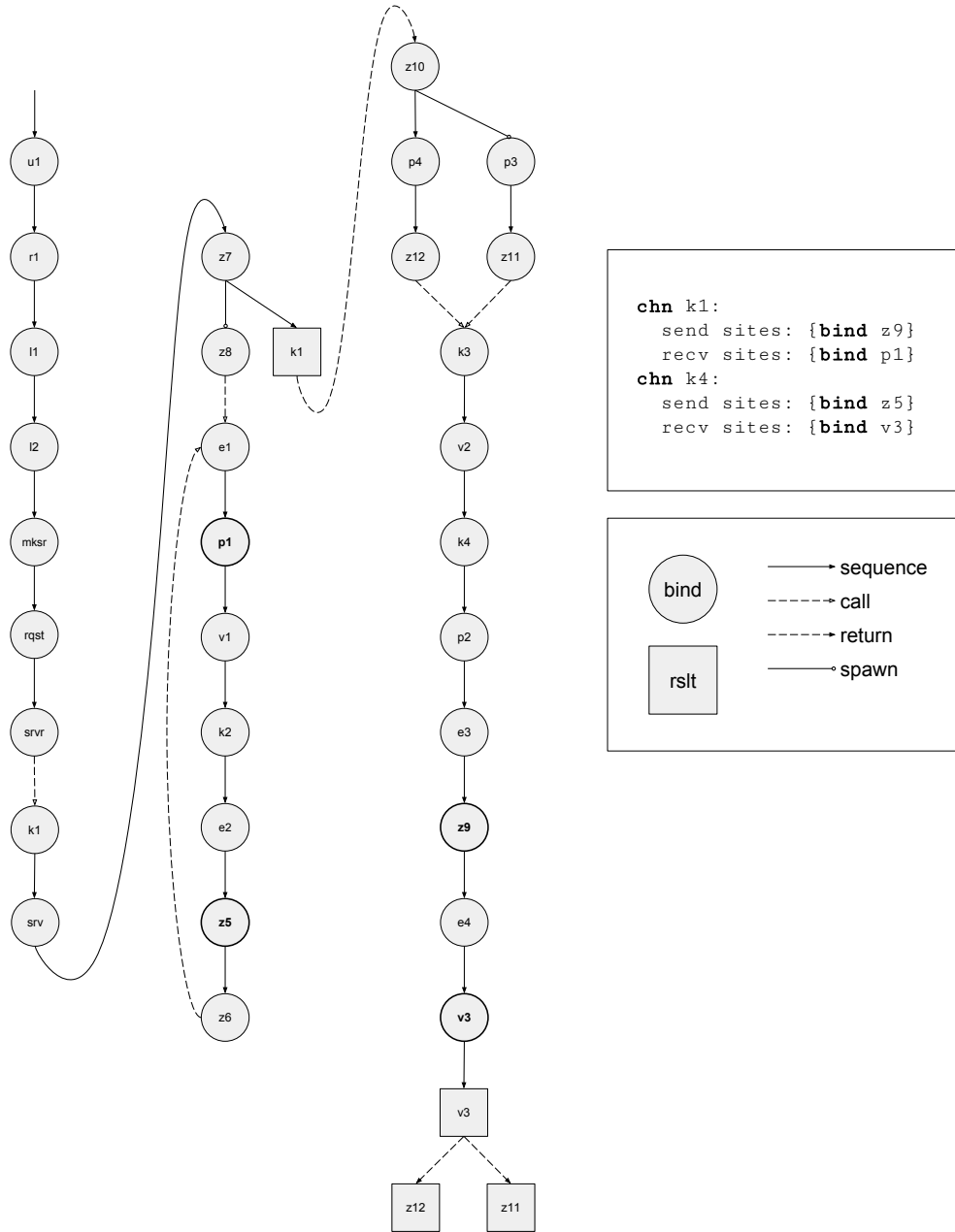
```

    staticFlowsAccept staticEnv graph t',
    staticFlowsAccept staticEnv graph t_b
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Atom (Fun n_f n_t t_b)) t')
* spawn: n t' t_c graph staticEnv .
  {
    (NBnd n, MNxt, termId t'),
    (NBnd n, MSpwn, termId t_c)
  } ⊆ graph,
  staticFlowsAccept staticEnv graph t_c,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Spwn t_c) t')
* sync: n t' graph staticEnv n_se .
  (NBnd n, MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Sync n_se) t')
* first: n t' graph staticEnv n_t .
  (NBnd n, MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t',
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Fst n_t) t')
* second: n t' graph staticEnv n_t .
  (NBnd n, MNxt, termId t') ∈ graph,
  staticFlowsAccept staticEnv graph t',
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Snd n_t) t')
* caseEither: n t_l t_r t' graph staticEnv n_s .
  {
    (NBnd n, MCll, termId t_l),
    (NBnd n, MCll, termId t_r),
    (NRslt (resultName t_l), MRtn, termId t'),
    (NRslt (resultName t_r), MRtn, termId t')
  } ⊆ graph,
  staticFlowsAccept staticEnv graph t_l,
  staticFlowsAccept staticEnv graph t_r,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (Case n_s n_l t_l n_r t_r) t')
* app: staticEnv n_f n t' n_a .
  ∀ n_f' n_t t_b .
    (SAtm (Fun n_f' n_t t_b)) ∈ staticEnv n_f
  →
  {
    (NBnd n, MCll, termId t_b),
    (NRslt (resultName t_b), MRtn, termId t')
  } ⊆ graph,
  staticFlowsAccept staticEnv graph t'
  ⊢ staticFlowsAccept staticEnv graph (Bind n (App n_f n_a) t')

```

The smallest graph of flows that accepts a program is finite. Additionally, the static acceptance

relation is syntax-directed, which offers guidance towards computing the graph from a program. Thus, to statically determine communication classifications, it should be possible to compute the two shortest paths that send or receive on the same channel. The server implementation represented as a control flow graph illustrates how static acceptance by flows can interpret a graph from a program.



The static traceability means that a static path with a given starting step, and ending condition, can be traced by traversing the flows in a graph. The empty path is statically traceable if the starting step meets the ending condition. Otherwise, a path is statically traceable if the last static

step corresponds to a flow that meets the ending condition, and the longest strict prefix of the path is statically traceable.

```
predicate staticTraceable:
  flow set -> termId -> (termId -> bool) -> static_path -> bool
where
  empty: start graph isEnd .
  isEnd start
  ⊢ staticTraceable graph start isEnd []
  * snoc: graph star middle path isEnd end mode .
    staticTraceable graph start (λ l . l = middle) path,
    isEnd end,
    (middle, mode, end) ∈ graph
  ⊢ staticTraceable graph start isEnd (path @ [(middle, mode)])
```

In the graph of the server implementation, there are two paths each corresponding to its own thread that lead to sending on static channel **chn** k1 and a potentially infinite number of paths that lead to receiving on channel **chn** k1, but all on the same thread. There are an infinite number of paths that lead to sending on static channel **chn** k4, and two paths that lead to receiving on static channel **chn** k4. This is certainly imprecise, as the static **chn** k4 corresponds to multiple distinct dynamic channels, each with just one sender and one receiver. The higher precision analysis discussed in section 4 addresses this issue.

The static inclusion means that two static paths might be traced in the same run of a program. Ordered paths might be inclusive, and also a path that diverges from another at a spawn flow 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 classification. The predicates are intended to be applied to paths starting from the beginning of a program.

```
predicate staticInclusive: static_path -> static_path -> bool where
  first: path1 path2 .
    prefix path1 path2
  ⊢ staticInclusive path1 path2
  * second path2 path1 .
    prefix path2 path1
  ⊢ staticInclusive path1 path2
  * spawnFirst: path n path1 path2 .
    \vdash staticInclusive
      (path @ [(NBnd n, MSpwn)] @ path1)
      (path @ [(NBnd n, MNxt)] @ path2)
  * spawnSecond: path n path1 path2 .
    ⊢ staticInclusive
      (path @ [(NBnd n, MNxt)] @ path1)
      (path @ [(NBnd n, MSpwn)] @ path2)
```

The noncompetitiveness means states that two paths can't compete during a run of a program,



since they are ordered or cannot occur in the same run of a program. The singularness means that two paths are the same or only of them can occur in a given run of a program.

```
predicate noncompetitive: static_path -> static_path -> bool where
  ordered: path1 path2 .
  ordered path1 path2
  ⊢ noncompetitive path1 path2
* notInclus: path1 path2 .
  ¬ (staticInclusive path1 path2)
  ⊢ noncompetitive path1 path2
```

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

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

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

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

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

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

```
predicate staticOneToOne: static_value_map -> term -> name -> bool where
  intro: staticEnv staticComm t graph nc .
    staticEval staticEnv staticComm t,
```

```

staticFlowsAccept staticEnv graph t,
forEveryTwo (staticTraceable graph (termId t)
  (staticSendId staticEnv t nc)) noncompetitive,
forEveryTwo (staticTraceable graph (termId t)
  (staticRecvId staticEnv t nc)) noncompetitive
⊢ staticOneToOne staticEnv t nc

```

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

```

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

```

The static one-sync classification means that there is at most one completed synchronization on a static channel in any run of a given program.

```

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

```

### 3.6 Formal Reasoning

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

The semantics and analysis must contain many details. To ensure the correctness of proofs, it is necessary to check that there are no subtle errors in either the definitions or proofs. Proofs in general require many subtle manipulations of symbols. The difference between a false statement and a true statement can often be difficult to spot, since the two may be very similar lexically. However, a mechanical proof checker, such as that of Isabelle, has no difficulty discerning between

valid and invalid derivations. Mechanical checking of proofs can notify users of errors in the proofs or definitions far better and faster than manual checking. This work has greatly benefited from Isabelle's proof checker in order to correctly define the language semantics, control flow analysis, communication analysis, and other helpful definitions. For instance, some bugs in the definitions were found trying to prove soundness. The proof checker would not accept the proof unless I provided facts that should be false, indicating that the definitions did not state my intentions. After correcting the errors in the definitions, the proof was completed such that the proof checker was satisfied.

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

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

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

```
predicate sortedLeft: nat list -> bool where
  empty:
    ⊢ sortedRight []
  * single: x .
    ⊢ sortedLeft [x]
  * cons: x y zs .
    n ≤ y,
    sortedLeft (y # zs)
    ⊢ sortedLeft (x # y # zs)

predicate sortedRight: nat list -> bool where
  empty:
    ⊢ sortedRight []
  * single: z .
    ⊢ sortedRight [z]
  * snoc: xs y z .
```

```

    sortedRight (xs @ [y]),
    y ≤ z
  ⊢ sortedRight (xs @ [y] @ [z])

```

```

lemma sortedEquiv: xs .
  ⊢ sortedLeft xs ≡ sortedRight xs

```

### 3.7 Soundness

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

```

theorem staticOneToManySound: t0 nc pathc .
  staticOneToMany t0 nc
  ⊢ oneToMany t0 (Chan pathc nc)

```

```

theorem staticManyToOneSound: t0 nc pathc .
  staticManyToOne t0 nc
  ⊢ manyToOne t0 (Chan pathc nc)

```

```

theorem staticOneToOneSound: t0 nc pathc .
  staticOneToOne staticEnv t0 nc
  ⊢ oneToOne t0 (Chan pathc nc)

```

```

theorem staticOneShotSound: t0 nc pathc .
  staticOneShot t0 nc
  ⊢ oneShot t0 (Chan pathc nc)

```

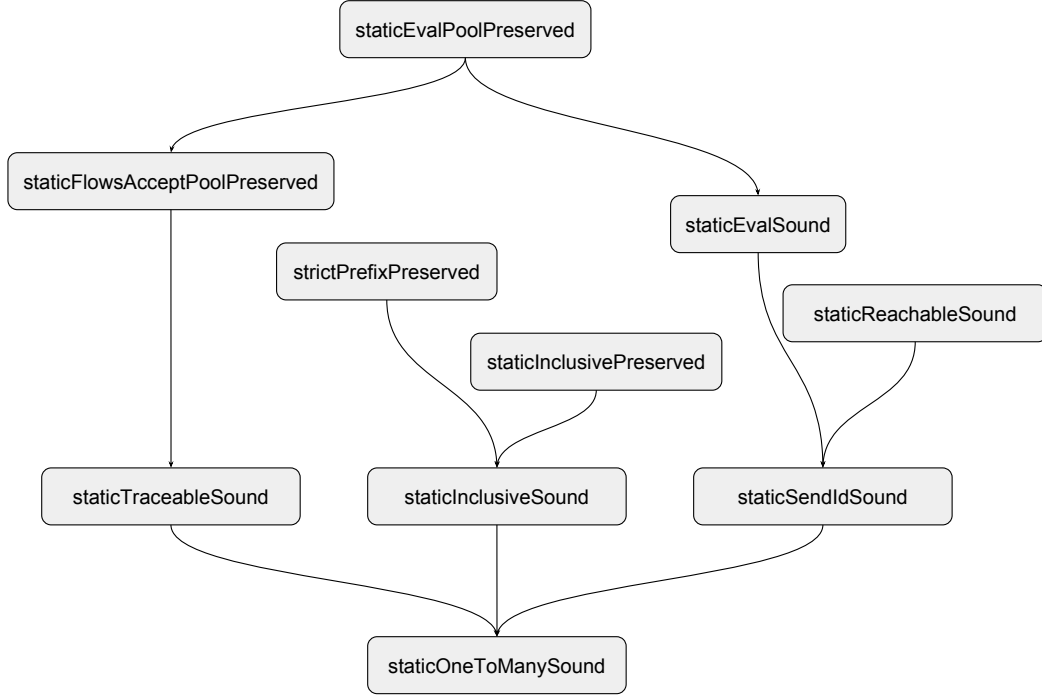
```

theorem staticOneShotSound: t0 nc pathc .
  staticOneSync t0 nc
  ⊢ oneSync t0 (Chan pathc nc)

```

The formal proofs of soundness of each static classification follow a similar structure. Let's examine in some detail the formal proof of soundness of static one-to-many classification, by unwinding the theorem into the lemmas that it follows from.

The following diagram illustrates the key dependencies of the theorems and lemmas in used in the derivations.



The soundness of static one-to-many classification is proved by a few simpler lemmas and the definitions of static and dynamic one-to-many classification. There is an isomorphic correspondence between the paths of dynamic evaluation and the paths of static evaluation, by definition. The static paths are derived from the static flow graph. Although it is possible to directly derive the dynamic paths from the static flow graph, deriving an isomorphic path structure keeps the the paths relation to the static flow graph clear. Additionally, for the higher precision analysis, static paths are not isomorphic to dynamic paths.

The three main lemmas state the soundness of the static traceability, the soundness of the static inclusiveness, and the soundness of a program step not being a static sending identifier. These lemmas depend on a correspondence between static paths and dynamic paths, which is bijective for the lower precision analysis. The lemma for soundness of static inclusiveness states that any two dynamic paths traced by running a program correspond to statically inclusive static paths. It follows from a straightforward case analysis of static inclusivity. The lemma for soundness of static traceability states that for any dynamic path traced by running a program, there is a corresponding static path that is statically traceable. The lemma for soundness of a program step not being a static sending identifier states that running a program reaches a synchronization on a sending event, then that synchronization is statically identified as a sending identifier by its term

identifier.

```

predicate pathsCorrespond: dynamic_path -> static_path -> bool where
  empty:
    ⊢ pathsCorrespond [] []
  * seq: path staticPath n .
    pathsCorrespond path staticPath
    ⊢ pathsCorrespond
      (path @ [DNxt n])
      (staticPath @ [(NBnd n, MNxt)])
  * spawn: path staticPath n .
    pathsCorrespond path staticPath
    ⊢ pathsCorrespond
      (path @ [DSpwn n])
      (staticPath @ [(NBnd n, MSpwn)])
  * call: path staticPath n .
    pathsCorrespond path staticPath
    ⊢ pathsCorrespond
      (path @ [DCll n])
      (staticPath @ [(NBnd n, MCll)])
  * return: path staticPath n .
    pathsCorrespond path staticPath
    ⊢ pathsCorrespond
      (path @ [DRtn n])
      (staticPath @ [(NRslt n, MRtn)])

```

```

Lemma staticTraceableSound: t0 pool comm path n c t' env stack staticEnv
  staticComm graph isEnd .
  star dynamicEval ([[] -> (Stt t0 [-> []]), {}) (pool, comm),
  pool path = Some (Stt (Bind n c t') env stack),
  staticEval staticEnv staticComm t0,
  staticFlowsAccept staticEnv graph t0,
  isEnd (NBnd n)
  ⊢ exists staticPath .
    pathsCorrespond path staticPath
  ∧ staticTraceable graph (termId t0) isEnd staticPath

```

```

Lemma staticInclusiveSound: t0 pool comm path1 stt1 path2 stt2 staticPath1
  staticPath2 .
  star dynamicEval ([[] -> (Stt t0 [-> []]) {} pool comm
  pool path1 = Some stt1,
  pool path2 = Some stt2,
  pathsCorrespond path1 staticPath1,
  pathsCorrespond path2 staticPath2
  ⊢ staticInclusive staticPath1 staticPath2

```

```

Lemma staticSendIdSound: t0 pool comm path n ne t' env stack nsc nm env' pathc nc

```

```

.
star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
pool path = Some (Stt (Bind n (Sync ne) t') env stack),
env ne = Some (VAtm (SendEvt nsc nm) env'),
env' nsc = Some (VChn (Chan pathc nc)),
staticEval staticEnv staticComm t0
⊢ staticSendId staticEnv t0 nc (NBnd n)

```

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

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

```

predicate staticFlowsAcceptValue:
  static_value_map -> graph -> dynamic_value -> bool
where
  unit: staticEnv graph .
  ⊢ staticFlowsAcceptValue staticEnv graph VUnt
  * chan: staticEnv graph nc .
  ⊢ staticFlowsAcceptValue staticEnv graph (VChn nc)
  * sendEvt: staticEnv graph env nc nm .
    staticFlowsAcceptEnv staticEnv graph env
  ⊢ staticFlowsAcceptVal
    staticEnv graph (VAtm (SendEvt nc nm) env)
  * recvEvt: staticEnv graph env nc .
    staticFlowsAcceptEnv staticEnv graph env
  ⊢ staticFlowsAcceptVal
    staticEnv graph (VAtm (RecvEvt nc) env)
  * left: staticEnv graph env np .
    staticFlowsAcceptEnv staticEnv graph env
  ⊢ staticFlowsAcceptVal
    staticEnv graph (VAtm (Lft np) env)
  * right: staticEnv graph env np .
    staticFlowsAcceptEnv staticEnv graph env
  ⊢ staticFlowsAcceptVal

```

```

    staticEnv graph (VAtm (Rht  $n_p$ ) env)
* function: staticEnv graph  $t_b$  env  $n_f$   $n_p$  .
    staticFlowsAccept staticEnv graph  $t_b$ ,
    staticFlowsAcceptEnv staticEnv graph env
  ⊢ staticFlowsAcceptVal
    staticEnv graph (VAtm (Fun  $n_f$   $n_p$   $t_b$ ) env)
* pair: staticEnv graph env .
    staticFlowsAcceptEnv staticEnv graph env
  ⊢ staticFlowsAcceptVal
    staticEnv graph (VAtm (Pair  $n_1$   $n_2$ ) env)

predicate staticFlowsAcceptEnv:
  static_value_map -> graph -> env -> bool
where
  intro: staticEnv graph env .
    ∀  $n$   $v$  .
      env  $n$  = Some  $v$ 
    → staticFlowsAcceptValue staticEnv graph  $v$ 
  ⊢ staticFlowsAcceptEnv staticEnv graph env

predicate staticFlowsAcceptStack:
  static_value_map -> graph -> name -> stack -> bool
where
  empty: staticEnv graph  $n_r$  .
  ⊢ staticFlowsAcceptStack staticEnv graph  $n_r$  []
* cons:  $n_r$   $t$  graph staticEnv graph env stack  $n$  env .
    {(NRslt  $y$ , MRtn, termId  $t$ )} ⊆ graph,
    staticFlowsAccept staticEnv graph  $t$ ,
    staticFlowsAcceptEnv staticEnv graph env,
    staticFlowsAcceptStack staticEnv graph (resultName  $t$ ) stack
  ⊢ staticFlowsAcceptStack staticEnv graph  $n_r$  ((Ctn  $n$   $t$  env) # stack)

predicate staticFlowsAcceptPool of
  static_value_map -> graph -> pool -> bool
where
  intro: staticEnv graph pool .
    ∀ path  $t$  env stack .
      env path = Some (Stt  $t$  env stack)
    →
      staticFlowsAccept staticEnv graph  $t$ 
      ∧ staticFlowsAcceptEnv staticEnv graph env
      ∧ staticFlowsAcceptStack staticEnv graph (resultName  $t$ ) stack
  ⊢ staticFlowsAcceptPool staticEnv graph pool

```

The flows described by the various versions of static acceptance by flows depend on static environments in order to look up the control flow in the case where the term is a function. The



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

```

fun abstract: dynamic_value -> static_value where
  ⊢ abstract VUnt = SUnt
* path n .
  ⊢ abstract (VChn (Chan path x)) = SChn x
* atom env .
  ⊢ abstract (VAtm atom env) = SAtm atom

predicate staticEvalValue:
  static_value_map -> static_value_map -> dynamic_value -> bool
where
  unit: staticEnv staticComm .
    ⊢ staticEvalValue staticEnv staticComm VUnit)
* chan: staticEnv staticComm c .
  ⊢ staticEvalValue staticEnv staticComm (VChn c))
* sendEvt: staticEnv staticComm env nc nm .
  staticEvalEnv staticEnv staticComm env
  ⊢ staticEvalValue staticEnv staticComm
    (VAtm (SendEvt nc nm) env))
* recvEvt: staticEnv staticComm env nc .
  staticEvalEnv staticEnv staticComm env
  ⊢ staticEvalValue staticEnv staticComm
    (VAtm (RecvEvt nc) env))
* left: staticEnv staticComm env np .
  staticEvalEnv staticEnv staticComm env
  ⊢ staticEvalValue staticEnv staticComm
    (VAtm (Lft np) env))
* right: staticEnv staticComm env np .
  staticEvalEnv staticEnv staticComm env
  ⊢ staticEvalValue staticEnv staticComm
    (VAtm (Rht np) env))
* function: nf np tb staticEnv staticComm env .
  {SAtm (Fun nf np tb)} ⊆ staticEnv f,
  staticEval staticEnv staticComm tb,
  staticEvalEnv staticEnv staticComm env
  ⊢ staticEvalValue staticEnv staticComm
    (VAtm (Fun nf np tb) env))

```

```

* pair: staticEnv staticComm env n1 n2 .
  staticEvalEnv staticEnv staticComm env
  ⊢ staticEvalValue staticEnv staticComm
    (VAtm (Pair n1 n2) env))

predicate staticEvalEnv:
  static_value_map -> static_value_map -> env -> bool
where
  intro: staticEnv staticComm env .
    ∀ n v .
      env n = Some v
      → {abstract v} ⊆ staticEnv n
      ∧ staticEvalValue staticEnv staticComm v
      ⊢ staticEvalEnv staticEnv staticComm env

predicate staticEvalStack:
  static_value_map -> static_value_map
  -> static_value set -> stack -> bool
where
  empty: staticEnv staticComm staticVals .
    ⊢ staticEvalStack staticEnv staticComm staticVals []
  * cons: staticVals staticEnv staticComm .
    staticVals ⊆ staticEnv n,
    staticEval staticEnv staticComm t,
    staticEvalEnv staticEnv staticComm env,
    staticEvalStack staticEnv staticComm (staticEnv (resultName t)) stack
    ⊢ staticEvalStack staticEnv staticComm staticVals ((Ctn n t env) # stack))

predicate staticEvalPool:
  static_value_map -> static_value_map -> pool -> bool
where
  intro: staticEnv staticComm pool .
    ∀ path t env stack .
      pool path = Some (Stt t env stack)
      →
        staticEval staticEnv staticComm t
        ∧ staticEvalEnv staticEnv staticComm env
        ∧ staticEvalStack staticEnv staticComm (staticEnv (resultName t)) stack
      ⊢ staticEvalPool staticEnv staticComm pool)

```

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

```

predicate starLeft: ('a -> 'a -> bool) -> 'a -> 'a -> bool where
  refl: r z z .

```

```

  ⊢ starLeft r z z
* trans: r x y z .
  starLeft r x y,
  r y z
  ⊢ starLeft r x z

```

```

Lemma starImpliesStarLeft: r x z .
  star r x z
  ⊢ starLeft r x z

```

```

Lemma starLeftTrans: r x y z .
  starLeft r x y,
  starLeft r y z
  ⊢ starLeft r x z

```

The lemma for the completeness of static traceability follows from the generalized definitions of static acceptance by flows, the definition of static traceability, and the preservation of static acceptance by flows across multiples steps of evaluation.

```

Lemma staticFlowsAcceptPoolPreserved: t0 pool comm staticEnv staticComm graph .
  star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
  staticEval staticEnv staticComm t0,
  staticFlowsAcceptPool staticEnv graph [[] -> (Stt t0 [->] [])]
  ⊢ staticFlowsAcceptPool staticEnv graph pool

```

The preservation of static acceptance by flows over pools is proved by the equivalence between star and its leftward variant, and induction on the leftward variant. The preservation of static evaluation of pools over multiple steps is also relied upon.

```

Lemma staticEvalPoolPreserved: pool comm pool' comm' staticEnv staticComm .
  star dynamicEval pool comm pool' comm'
  staticEvalPool staticEnv staticComm pool
  ⊢ staticEvalPool staticEnv staticComm pool'

```

The completeness of static inclusiveness is derived from various lemmas that preserve relations from pairs of dynamic paths to pairs of corresponding static paths. Some of these, among many others, are the preservation of the strict prefix relation from static to dynamic paths, and the preservation of static inclusiveness over extension of static paths.

```

Lemma strictPrefixPreservedCorresp: staticPathath1 staticPath2 dynamicPath1
dynamicPath2 .
  strictPrefix staticPathath1 staticPath2,
  pathsCorrespond dynamicPath1 staticPath1,
  pathsCorrespond dynamicPath2 staticPath2
  ⊢ strictPrefix dynamicPath1 dynamicPath2

```

```

Lemma staticInclusivePreservedUnorderedExtension: path1 path2 l1 l2 .
  staticInclusive path1 path2,

```

$\neg (\text{prefix path1 path2}),$   
 $\neg (\text{prefix path2 path1}),$   
 $\vdash \text{staticInclusive } (\text{path1 @ [l1]}) (\text{path2 @ [l2]})$

These various preservation lemmas are derived from the basic properties of lists and straight forward properties of path correspondence, such as commutativity, as well as foundational principles like induction of corresponding paths.

The lemma for completeness of sending identifier classification `staticSendIdSound` is proved using the lemma for soundness of static evaluation for synchronization of a send event, and the lemma for soundness of static evaluation. Since only sending identifiers are relevant the completeness of static reachability is used to ensure that the static step is indeed a sending identifier.

**Lemma** `sendChanStaticEvalSound`:  $t_0$  pool comm staticEnv staticComm path  
 $n \ n_e \ t' \ \text{env} \ \text{stack} \ n_{sc} \ n_m \ \text{env}_e \ \text{path}_c \ n_c \ .$   
 $\text{star dynamicEval } [[] \rightarrow (\text{Stt } t_0 \ [->] \ [])] \ \{\}$  pool comm,  
 $\text{staticEval staticEnv staticComm } t_0,$   
 $\text{pool path} = \text{Some } (\text{Stt } (\text{Bind } n \ (\text{Sync } n_e) \ t') \ \text{env} \ \text{stack}),$   
 $\text{env } n_e = \text{Some } (\text{VAtm } (\text{SendEvt } n_{sc} \ n_m) \ \text{env}_e),$   
 $\text{env}_e \ n_{sc} = \text{Some } (\text{VChn } (\text{Chan } \text{path}_c \ n_c))$   
 $\vdash \text{SChn } n_c \in \text{staticEnv } n_{sc}$

**Lemma** `staticEvalSound`:  $t_0$  pool comm staticEnv staticComm path  $t \ \text{env} \ \text{stack} \ n \ v \ .$   
 $\text{staticEval staticEnv staticComm } t_0,$   
 $\text{star dynamicEval } [[] \rightarrow (\text{Stt } t_0 \ [->] \ [])] \ \{\}$  pool comm,  
 $\text{pool path} = \text{Some } (\text{Stt } t \ \text{env} \ \text{stack}),$   
 $\text{env } n = \text{Some } v$   
 $\vdash \text{abstract } v \in \text{staticEnv } n$

**Lemma** `staticReachableSound`:  $t_0$  pool comm staticEnv staticComm path  $t \ \text{env} \ \text{stack} \ .$   
 $\text{star dynamicEval } [[] \rightarrow (\text{Stt } t_0 \ [->] \ [])] \ \{\}$  pool comm,  
 $\text{pool path} = \text{Some } (\text{Stt } t \ \text{env} \ \text{stack})$   
 $\vdash \text{staticReachable } t_0 \ t$

Both the soundness of static evaluation on the synchronization of a send event, and the soundness of static evaluation follow from the preservation of static evaluation over multiple steps of dynamic evaluation.

The lemma for completeness of static reachability relies on a reformulation of static reachability defined by proofs that induct on a larger term containing the reachable term. This definition is useful for forward derivations of reachability relations, however it doesn't offer much guidance for deciding reachability. In contrast, the definition of the original static reachability relation is syntax-directed in order to portray a clear connection to a computable algorithm that can determine the reachable term from an initial program. To show that a term is reachable from the initial program, it is necessary to show that each intermediate term is reachable from the initial term. Thus, the induction needs to enable unraveling the goals from the end of the program to the beginning, maintaining the initial program state in context for each subgoal. Because the static

reachable relation is a "may", the generalized relations should hold unless there is a clear reason that it could never hold. For instance, in the relation over atoms, it holds for all atoms other than functions, it holds without any additional demands.

```
predicate staticReachableForward: term -> term -> bool where
  refl: t0 .
  ⊢ staticReachableForward t0 t0
* spawn: t0 n tc t' .
  staticReachableForward t0 (Bind n (Spwn tc) t')
  ⊢ staticReachableForward t0 tc
* caseLeft: t0 n ns nl tl nr tr t' .
  staticReachableForward t0 (Bind n (Case ns nl tl nr tr) t')
  ⊢ staticReachableForward t0 tl
* caseRight: t0 n ns nl tl nr tr t' .
  staticReachableForward t0 (Bind n (Case ns nl tl nr tr) t')
  ⊢ staticReachableForward t0 tr
* function: t0 n nf np tb t' .
  staticReachableForward t0 (Bind n (Atom (Fun nf np tb)) t')
  ⊢ staticReachableForward t0 tb
* seq: t0 n nf np tb t' .
  staticReachableForward t0 (Bind n c t')
  ⊢ staticReachableForward t0 t'
```

```
predicate staticReachableAtom: term -> atom -> bool where
  sendEvt: t0 nc nm .
  ⊢ staticReachableAtom t0 (SendEvt nc nm)
* recvEvt: t0 nc .
  ⊢ staticReachableAtom t0 (RecvEvt nc)
* pair: t0 n1 n2 .
  ⊢ staticReachableAtom t0 (Pair n1 n2)
* left: t0 nl .
  ⊢ staticReachableAtom t0 (Lft nl)
* right: t0 nr .
  ⊢ staticReachableAtom t0 (Rht nr)
* function: t0 tb nf np tb .
  staticReachableForward t0 tb
  ⊢ staticReachableAtom t0 (Fun nf np tb)
```

```
predicate staticReachableVal: term -> dynamic_value -> bool where
  unit: t0 .
  ⊢ staticReachableValue t0 VUnit
* chan: t0 c .
  ⊢ staticReachableValue t0 (VChn c)
* atom: t0 t env .
  staticReachableAtom t0 t,
  staticReachableEnv t0 env
```

$\vdash \text{staticReachableValue } t_0 \text{ (VAtm } t \text{ env)}$

**predicate** staticReachableEnv: term  $\rightarrow$  env  $\rightarrow$  bool **where**

intro:  $t_0$  env  
 $\forall n \ v \ .$   
 env  $n = \text{Some } v$   
 $\rightarrow \text{staticReachableValue } t_0 \ v$   
 $\vdash \text{staticReachableEnv } t_0 \text{ env}$

**predicate** staticReachableStack: term  $\rightarrow$  stack  $\rightarrow$  bool **where**

empty:  $t_0 \ .$   
 $\vdash \text{staticReachableStack } t_0 \ []$   
 \* cons:  $t_0 \ t_k \ \text{env}_k \ \text{stack}' \ .$   
 staticReachableForward  $t_0 \ t_k$ ,  
 staticReachableEnv  $t_0 \ \text{env}_k$ ,  
 staticReachableStack  $t_0 \ \text{stack}'$   
 $\vdash \text{staticReachableStack } t_0 \ ((\text{Ctn } n_k \ t_k \ \text{env}_k) \# \text{stack}')$

**predicate** staticReachablePool: term  $\rightarrow$  pool  $\rightarrow$  bool **where**

intro:  $t_0 \ \text{pool} \ .$   
 $\forall \text{path } t \ \text{env} \ \text{stack} \ .$   
 pool path = Some (Stt  $t \ \text{env} \ \text{stack}$ )  
 $\rightarrow \text{staticReachableForward } t_0 \ t$   
 $\wedge \text{staticReachableEnv } t_0 \ \text{env}$   
 $\wedge \text{staticReachableStack } t_0 \ \text{stack}$   
 $\vdash \text{staticReachablePool } t_0 \ \text{pool}$

The completeness of static reachability follows the definitions a generalized form of completeness over pools.

**Lemma** staticReachablePoolSound:  $t_0 \ \text{pool} \ .$   
 star dynamicEval  $[[[] \rightarrow (\text{Stt } t_0 \ [->] [])], \{\} \ \text{pool} \ \text{comm}$   
 $\vdash \text{staticReachablePool } t_0 \ \text{pool}$

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

**Lemma** staticReachableForwardImpliesStaticReachable:  $t_0 \ t \ .$   
 staticReachableForward  $t_0 \ t$   
 $\vdash \text{staticReachable } t_0 \ t$

**Lemma** staticReachableTrans:  $t_1 \ t_2 \ t_3 \ .$   
 staticReachable  $t_1 \ t_2$ ,  
 staticReachable  $t_2 \ t_3$   
 $\vdash \text{staticReachable } t_1 \ t_3$

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

## 4 Higher Precision Communication

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

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

The higher precision analysis uses a trimmed down graph to better differentiate between distinct channels. A trimmed graph is specialized for a particular dynamic channel. From the creation step, it must contain transitive flows to all the program steps where the channel is live. It should also be as small as possible, for higher precision.

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

```
datatype mode =
  MNxt
| MSpwn
| ESend name
| MCl1
| MRtn

type flow = termId * mode * termId

type static_step = termId * mode

type staticPath = static_step list
```

We use a slightly modified version of the server implementation to demonstrate some key concepts of the higher precision analysis. An additional loop function `lp` has been added to the server implementation. The loop basically just wastes steps, but it is used to demonstrate how liveness analysis treats functions that don't contain any channel of interest.

```

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

bind lp = fun lp' x1 =>
(
  bind z1 = case x1 of
    lft y1 =>
      (
        bind z2 = app lp' y1
        rslt z2
      )
    | rht y2 =>
      (
        bind u2 = unt
        rslt u2
      )
  bind u3 = unt
  rslt u3
)

bind mksr = fun _ x2 =>
(
  bind k1 = mkChn
  bind z4 = lp l2
  bind srv = fun srv' x3 =>
    (
      bind e1 = recvEvt k1
      bind p1 = sync e1
      bind v1 = fst p1
      bind k2 = snd p1
      bind e2 = sendEvt k2 x3
      bind z5 = sync e2
      bind z6 = app srv' v1
      rslt z6
    )
  bind z7 = spawn
  (
    bind z8 = srv r1
    rslt z8
  )
)

```



```

    )
    rslt k1
  )

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

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

```

The static acceptance by flows for higher precision is similar to that of the lower precision analysis. However, it must additionally consider flows with the sending mode.

```

predicate staticFlowsAcceptTm:
  static_value_map -> graph -> term -> term -> bool
where
  result: staticEnv graph n .
  ⊢ staticFlowsAcceptTm staticEnv graph t0 (Rslt n)
* unit: n t' graph staticEnv .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAcceptTm staticEnv graph t0 t'
  ⊢ staticFlowsAcceptTm staticEnv graph t0 (Bind n Unt t')
* makeChan: n t' graph staticEnv .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAcceptTm staticEnv graph t0 t'
  ⊢ staticFlowsAcceptTm staticEnv graph t0 (Bind n MkChn t')
* sendEvt: n t' graph staticEnv nc nm .
  (NBnd n , MNxt, termId t') ∈ graph,
  staticFlowsAcceptTm staticEnv graph t0 t'

```

$\vdash \text{staticFlowsAccept}$   
 $\text{staticEnv graph}$   
 $(\text{Bind } n \text{ (Atom (SendEvt } n_c \text{ } n_m)) } t') \text{ t'}$   
 $* \text{recvEvt: } n \text{ t' graph staticEnv } n_c .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'}$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Atom (RecvEvt } n_c)) } t')$   
 $* \text{pair: } n \text{ t' graph staticEnv } n_1 \text{ } n_2 .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'}$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Atom (Pair } n_1 \text{ } n_2)) } t')$   
 $* \text{left: } n \text{ t' graph staticEnv } n_s .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'}$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Atom (Lft } n_s)) } t')$   
 $* \text{right: } n \text{ t' graph staticEnv } n_s .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'}$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Atom (Rht } n_s)) } t')$   
 $* \text{function: } n \text{ t' graph staticEnv } t_b \text{ } n_f \text{ } n_p .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ } t_b$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Atom (Fun } n_f \text{ } n_p \text{ } t_b)) } t')$   
 $* \text{spawn: } n \text{ t' } t_c \text{ graph staticEnv.}$   
 $\{(\text{NBnd } n , \text{MNxt, termId } t'),$   
 $(\text{NBnd } n , \text{MSpwn, termId } t_c)\} \subseteq \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ } t_c,$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'}$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Spwn } t_c) } t')$   
 $* \text{sync: } n \text{ t' graph staticEnv } n_{se} .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\forall n_{sc} \text{ } n_m \text{ } n_c \text{ } n_y .$   
 $(\text{SATm (SendEvt } n_{sc} \text{ } n_m)) \in \text{staticEnv } n_{se},$   
 $\rightarrow (\text{SChn } n_c) \in \text{staticEnv } n_{sc}$   
 $\rightarrow \text{staticRecvId staticEnv } t_0 \text{ } n_c (\text{NBnd } n_y)$   
 $\rightarrow (\text{NBnd } n , \text{ESend } n_{se} , \text{NBnd } n_y) \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'}$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Sync } n_{se}) } t')$   
 $* \text{first: } n \text{ t' graph staticEnv } n_p .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'},$   
 $\vdash \text{staticFlowsAcceptTm staticEnv graph } t_0 (\text{Bind } n \text{ (Fst } n_p) } t')$   
 $* \text{second: } n \text{ t' graph staticEnv } n_p .$   
 $(\text{NBnd } n , \text{MNxt, termId } t') \in \text{graph},$   
 $\text{staticFlowsAcceptTm staticEnv graph } t_0 \text{ t'},$

```

  ⊢ staticFlowsAcceptTm staticEnv graph t0 (Bind n (Snd np) t')
* caseEither: n tl tr t' graph staticEnv ns .
  {
    (NBnd n, MCll, termId tl),
    (NBnd n, MCll, termId tr),
    (NRslt (resultName tl), MRtn, termId t'),
    (NRslt (resultName tr), MRtn, termId t')
  } ⊆ graph,
  staticFlowsAcceptTm staticEnv graph t0 tl,
  staticFlowsAcceptTm staticEnv graph t0 tr,
  staticFlowsAcceptTm staticEnv graph t0 t'
  ⊢ staticFlowsAcceptTm staticEnv graph t0 (Bind n (Case ns nl tl nr tr) t')
* app: staticEnv nf n t' na .
  ∀ nf' np tb .
    (SAtm (Fun nf' np tb)) ∈ staticEnv nf
  →
  {
    (NBnd n, MCll, termId tb),
    (NRslt (resultName tb), MRtn, termId t')
  } ⊆ graph,
  staticFlowsAcceptTm staticEnv graph t0 t'
  ⊢ staticFlowsAcceptTm staticEnv graph t0 (Bind n (App nf na) t')

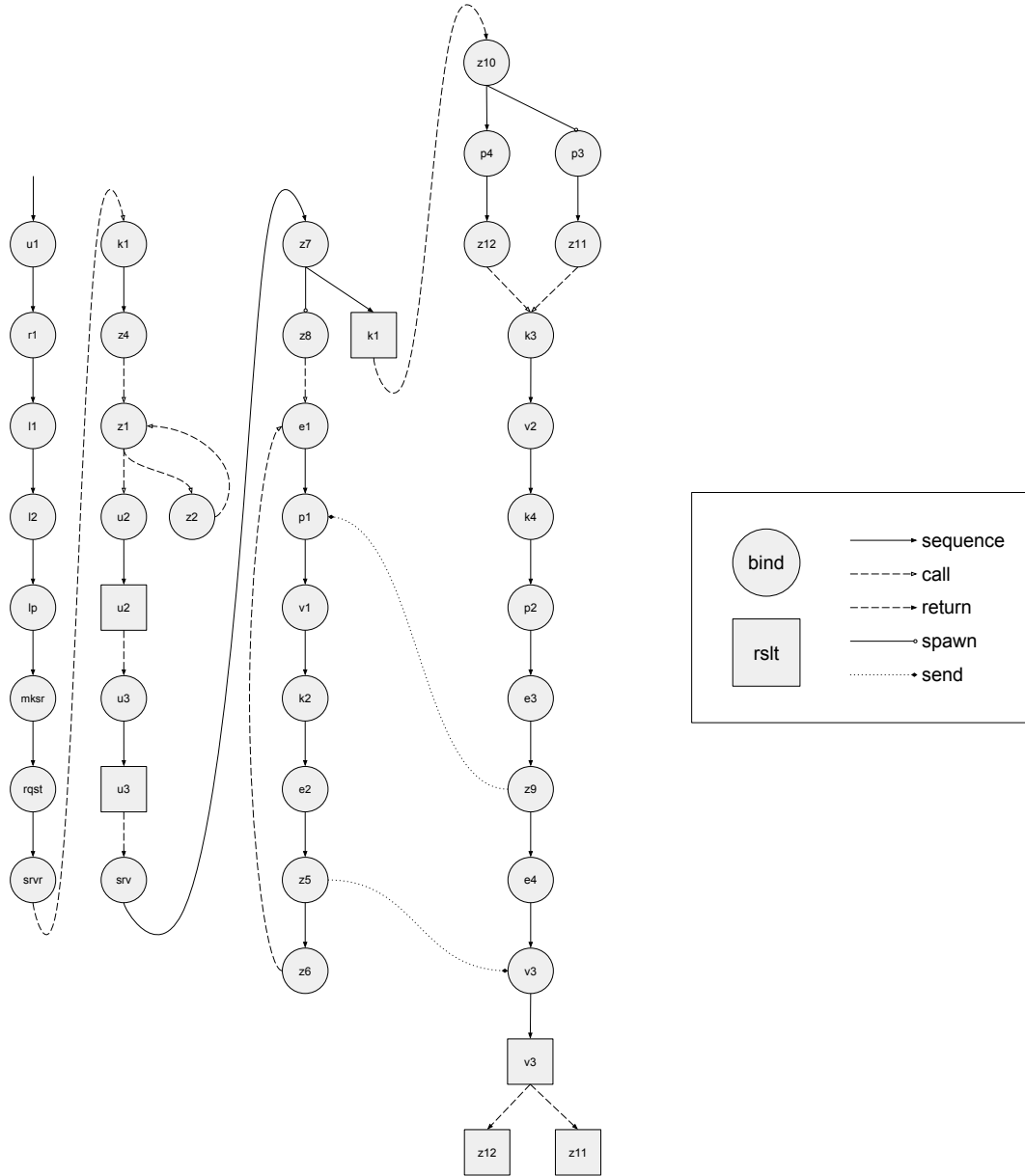
```

```

predicate staticFlowsAccept:
  static_value_map -> graph -> term -> bool
where
  intro: staticEnv graph n .
    staticFlowsAcceptTm staticEnv graph t0 t0
  ⊢ staticFlowsAccept staticEnv graph t0

```

The server implementation represented as a graph illustrates how static acceptance by flows can interpret a program as a flow graph.



For the liveness of channel analysis, it is necessary to track any name built on a channel. A name is build on a channel if the name binds to a static value containing a channel of interest, or a static value that contains names built on the channel. In the case where the tracked name possibly binds to a function, for the name to be considered built on a channel, the channel simply needs to be live in the body of the function. This condition is represented formally as the requirement

that there is a name, such that it is a free variable in the function, and it's built on the channel.

```

fun freeVarsAtom: atom -> name set where
  nc nm .
  ⊢ freeVarsAtom (SendEvt nc nm) = {nc, nm}
* nc .
  freeVarsAtom (RecvEvt nc) = {nc}
* n1 n2 .
  ⊢ freeVarsAtom (Pair n1 n2) = {n1, n2}
* n .
  ⊢ freeVarsAtom (Lft n) = {n}
* n .
  ⊢ freeVarsAtom (Rht n) = {n}
* nf np tb .
  ⊢ freeVarsAtom (Fun nf np tb) = freeVarsTerm tb \ {nf, np}

and freeVarsComplex: complex -> name set where
  ⊢ freeVarsComplex Unt = {}
* ⊢ freeVarsComplex MkChn = {}
* atom .
  ⊢ freeVarsComplex (Atom atom) = freeVarsAtom atom
* t .
  ⊢ freeVarsComplex (Spwn t) = freeVarsTerm t
* n .
  ⊢ freeVarsComplex (Sync n) = {n}
* n .
  ⊢ freeVarsComplex (Fst n) = {n}
* n .
  ⊢ freeVarsComplex (Snd n) = {n},

* ns nl tl nr tr .
  ⊢ freeVarsComplex (Case ns nl tl nr tr) =
    {ns} ∪ freeVarsTerm tl ∪ freeVarsTerm tr \ {nl, nr}
* nf na .
  ⊢ freeVarsComplex (App nf na) = {nf, na},

and freeVarsTerm: term -> name set where
  n c t .
  ⊢ freeVarsTerm (Bind n c t) = freeVarsComplex c ∪ freeVarsTerm t \ {n}
* n .
  ⊢ freeVarsTerm (Rslt n) = {n}

predicate staticBuiltOnChan: static_value_map -> name -> name -> bool where
  chan: nc staticEnv n .
    SChn nc ∈ staticEnv n
  ⊢ staticBuiltOnChan staticEnv nc n

```

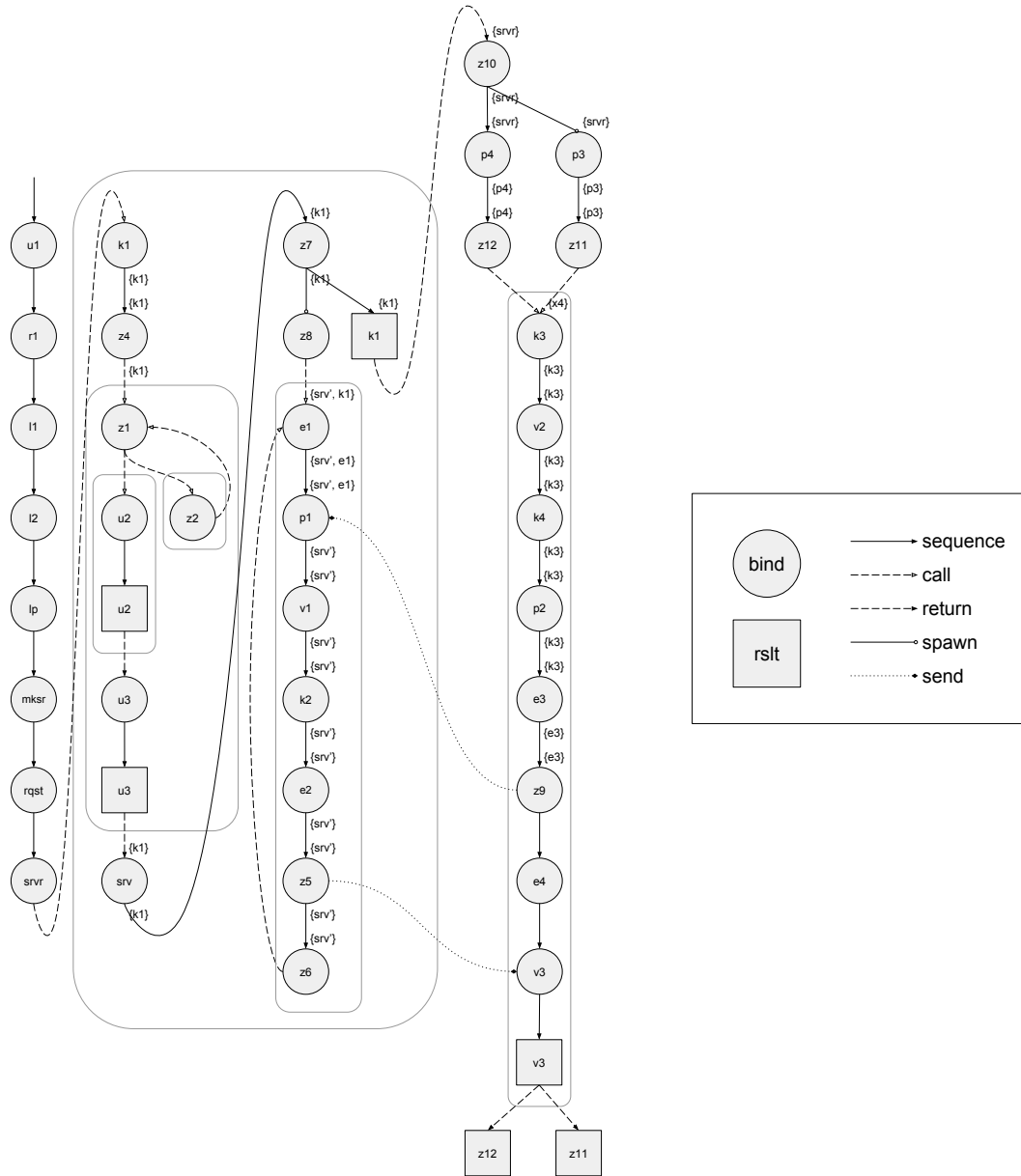
```

* sendEvt:  $n_{sc} \ n_m \ \text{staticEnv} \ n \ n_c \ .$ 
  ( $\text{SAtm} \ (\text{SendEvt} \ n_{sc} \ n_m) \in \text{staticEnv} \ n,$ 
  (
     $\text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n_{sc}$ 
 $\vee \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n_m$ 
  )
 $\vdash \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n$ 
* recvEvt:  $n_{rc} \ \text{staticEnv} \ n \ n_c \ .$ 
  ( $\text{SAtm} \ (\text{RecvEvt} \ n_{rc}) \in \text{staticEnv} \ n,$ 
   $\text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n_{rc}$ 
 $\vdash \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n$ 
* pair:  $n_1 \ n_2 \ \text{staticEnv} \ n \ n_c \ .$ 
  ( $\text{SAtm} \ (\text{Pair} \ n_1 \ n_2) \in \text{staticEnv} \ n,$ 
  (
     $\text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n_1$ 
 $\vee \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n_2$ 
  )
 $\vdash \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n$ 
* left:  $n_a \ \text{staticEnv} \ n \ n_c \ .$ 
  ( $\text{SAtm} \ (\text{Lft} \ n_a) \in \text{staticEnv} \ n,$ 
   $\text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n_a$ 
 $\vdash \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n$ 
* right:  $n_a \ \text{staticEnv} \ n \ n_c \ .$ 
  ( $\text{SAtm} \ (\text{Rht} \ n_a) \in \text{staticEnv} \ n,$ 
   $\text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n_a$ 
 $\vdash \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n$ 
* function:  $n_f \ n_p \ t_b \ n_{fv} \ .$ 
  ( $\text{SAtm} \ (\text{Fun} \ n_f \ n_p \ t_b) \in \text{staticEnv} \ n,$ 
 $n_{fv} \in \text{freeVarsAtom} \ (\text{Fun} \ n_f \ n_p \ t_b),$ 
 $\text{staticBuiltOnChan} \ \text{staticEnv} \ n_{fv} \ n$ 
 $\vdash \text{staticBuiltOnChan} \ \text{staticEnv} \ n_c \ n$ 

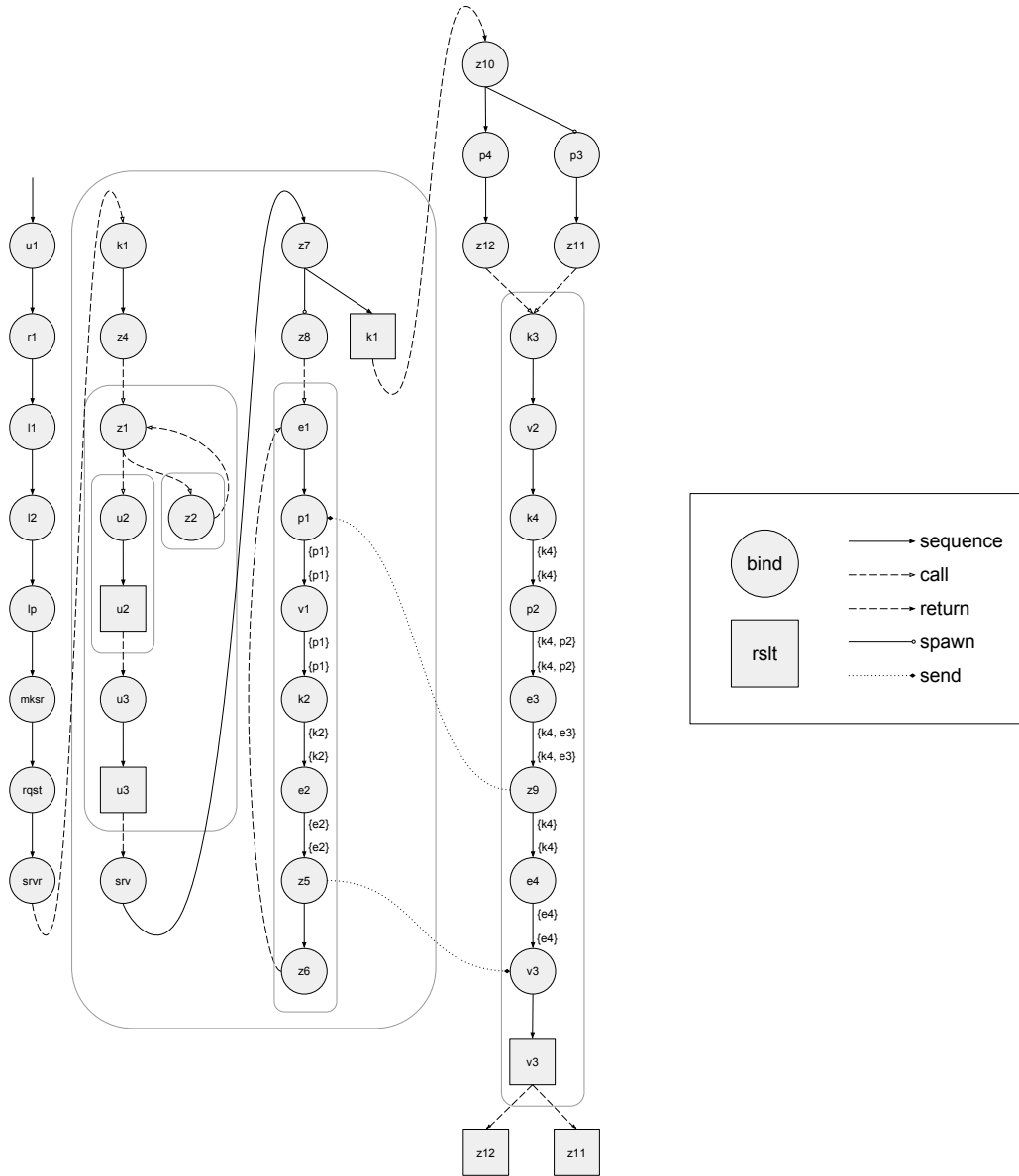
```

The static liveness of a channel describes entry functions and exit functions. The entry function maps a term identifier to a set of names built on the given channel, if those names are live at the entry of that term identifier. The exit function maps a term identifier to a set of names built on the given channel, if those names are live at the exit of that term identifier.

The following diagram illustrates the entry and exit sets of each term id for channel k1 in the server example. Each entry set appears right above its related term identifier, and each exit set appears right below its related term identifier.



The following diagram illustrates the entry and exit sets of each term id for channel  $k_4$  in the server example. Each entry set appears right above its related term identifier, and each exit set appears right below its related term identifier.



```

predicate staticLiveChan:
  static_value_map -> term_id_map
-> term_id_map -> name -> term -> bool
where
  result: staticEnv entr nc ny exit .
  (

```



```

    (staticBuiltOnChan staticEnv nc ny)
  → {ny} ⊆ entr (NRslt ny)
)
⊢ staticLiveChan staticEnv entr exit nc (Rslt ny)
* unit: exit n entr t' staticEnv nc .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  entr (termId t') ⊆ exit (NBnd n),
  staticLiveChan staticEnv entr exit nc t'
⊢ staticLiveChan staticEnv entr exit nc (Bind n Unt t')
* makeChan: exit n entr t' staticEnv nc .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  entr (termId t') ⊆ exit (NBnd n),
  staticLiveChan staticEnv entr exit nc t'
⊢ staticLiveChan staticEnv entr exit nc (Bind n MkChn t')
* sendEvt: exit n entr staticEnv nc nsc nm t' nc .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  (staticBuiltOnChan staticEnv nc nsc
  → {nsc} ⊆ entr (NBnd n)),
  (staticBuiltOnChan staticEnv nc nm
  → {nm} ⊆ entr (NBnd n)),
  entr (termId t') ⊆ exit (NBnd n),
  staticLiveChan staticEnv entr exit nc t'
⊢ staticLiveChan staticEnv entr exit nc
  (Bind n (Atom (SendEvt nsc nm)) t')
* recvEvt: exit n entr staticEnv nc nr nrc .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  (staticBuiltOnChan staticEnv nc nr then
  → {nr} ⊆ entr (NBnd n)),
  entr (termId t') ⊆ exit (NBnd n),
  staticLiveChan staticEnv entr exit nc t'
⊢ staticLiveChan staticEnv entr exit nc
  (Bind n (Atom (RecvEvt nrc)) t')
* pair: exit n entr staticEnv tc n1 n2 t' .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  (staticBuiltOnChan staticEnv nc n1
  → {n1} ⊆ entr (NBnd n)),
  (staticBuiltOnChan staticEnv nc n2
  → {n2} ⊆ entr (NBnd n)),
  entr (termId t') ⊆ exit (NBnd n),
  staticLiveChan staticEnv entr exit nc t'
⊢ staticLiveChan staticEnv entr exit nc (Bind n (Atom (Pair n1 n2)) t'))
* left: exit n entr staticEnv nc na t' .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  (staticBuiltOnChan staticEnv nc na
  → {na} ⊆ entr (NBnd n)),
  entr (termId t') ⊆ exit (NBnd n),

```

```

    staticLiveChan staticEnv entr exit  $n_c$   $t'$ 
   $\vdash$  staticLiveChan staticEnv entr exit  $n_c$  (Bind  $n$  (Atom (Lft  $n_a$ ))  $t'$ ))
* right: exit  $n$  entr staticEnv  $n_c$   $n_a$   $t'$  .
  (exit (NBnd  $n$ )  $\setminus$  { $n$ })  $\subseteq$  entr (NBnd  $n$ ),
  (staticBuiltOnChan staticEnv  $n_c$   $n_a$ 
 $\rightarrow$  { $n_a$ }  $\subseteq$  entr (NBnd  $n$ ))
  entr (termId  $e$ )  $\subseteq$  exit (NBnd  $n$ ),
  staticLiveChan staticEnv entr exit  $n_c$   $t$ 
   $\vdash$  staticLiveChan staticEnv entr exit  $n_c$  (Bind  $n$  (Atom (Rht  $n_a$ ))  $e$ )
* function: exit  $n$  entr  $t_b$   $n_p$   $n$  staticEnv  $n_c$   $t'$   $n_f$  .
  (exit (NBnd  $n$ )  $\setminus$  { $n$ })  $\subseteq$  entr (NBnd  $n$ ),
  (entr (termId  $t_b$ )  $\setminus$  { $n_f$ ,  $n_p$ })  $\subseteq$  entr (NBnd  $n$ ),
  staticLiveChan staticEnv entr exit  $n_c$   $t_b$ ,
  entr (termId  $t'$ )  $\subseteq$  exit (NBnd  $n$ ),
  staticLiveChan staticEnv entr exit  $n_c$   $t'$ 
   $\vdash$  staticLiveChan staticEnv entr exit  $n_c$ 
    (Bind  $n$  (Atom (Fun  $n_f$   $n_p$   $t_b$ ))  $t'$ )
* spawn: exit  $n$  entr  $t'$   $t_c$   $n_c$  staticEnv .
  (exit (NBnd  $n$ )  $\setminus$  { $n$ })  $\subseteq$  entr (NBnd  $n$ ),
  entr (termId  $t'$ )  $\subseteq$  exit (NBnd  $n$ ),
  entr (termId  $t_c$ )  $\subseteq$  exit (NBnd  $n$ ),
  staticLiveChan staticEnv entr exit  $n_c$   $t_c$ ,
  staticLiveChan staticEnv entr exit  $n_c$   $t'$ 
   $\vdash$  staticLiveChan staticEnv entr exit  $n_c$ 
    (Bind  $n$  (Spwn  $t_c$ )  $t'$ )
* sync: exit  $n$  entr staticEnv  $n_c$   $n_e$   $t'$  .
  (exit (NBnd  $n$ )  $\setminus$  { $n$ })  $\subseteq$  entr (NBnd  $n$ ),
  (staticBuiltOnChan staticEnv  $n_c$   $n_e$ 
 $\rightarrow$  { $n_e$ }  $\subseteq$  entr (NBnd  $n$ )),
  entr (termId  $t'$ )  $\subseteq$  exit (NBnd  $n$ ),
  staticLiveChan staticEnv entr exit  $n_c$   $t'$ ,
   $\vdash$  staticLiveChan staticEnv entr exit  $n_c$ 
    (Bind  $n$  (Sync  $n_e$ )  $t'$ )
* first: exit  $n$  entr staticEnv  $n_c$   $n_a$   $t'$  .
  (exit (NBnd  $n$ )  $\setminus$  { $n$ })  $\subseteq$  entr (NBnd  $n$ ),
  (staticBuiltOnChan staticEnv  $n_c$   $n_a$ 
 $\rightarrow$  { $n_a$ }  $\subseteq$  entr (NBnd  $n$ )),
  entr (termId  $t'$ )  $\subseteq$  exit (NBnd  $n$ ),
  staticLiveChan staticEnv entr exit  $n_c$   $t'$ 
   $\vdash$  staticLiveChan staticEnv entr exit  $n_c$  (Bind  $n$  (Fst  $n_a$ )  $t'$ )
*  $\forall$  exit  $n$  entr staticEnv  $n_c$   $n_a$   $t'$  .
  (exit (NBnd  $n$ )  $\setminus$  { $n$ })  $\subseteq$  entr (NBnd  $n$ ),
  (staticBuiltOnChan staticEnv  $n_c$   $n_a$  then
 $\rightarrow$  { $n_a$ }  $\subseteq$  entr (NBnd  $n$ )),
  entr (termId  $t'$ )  $\subseteq$  exit (NBnd  $n$ ),
  staticLiveChan staticEnv entr exit  $n_c$   $t'$ 

```

```

  ⊢ staticLiveChan staticEnv entr exit nc
    (Bind n (Snd na) t')
* caseEither: exit n entr tl nl tr nr staticEnv nc ns t' .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  (entr (termId tl) \ {nl}) ⊆ entr (NBnd n),
  (entr (termId tr) \ {nr}) ⊆ entr (NBnd n),
  (staticBuiltOnChan staticEnv nc ns
    → {ns} ⊆ entr (NBnd n)),
  staticLiveChan staticEnv entr exit nc tl,
  staticLiveChan staticEnv entr exit nc tr,
  entr (termId t') ⊆ exit (NBnd n),
  staticLiveChan staticEnv entr exit nc t'
  ⊢ staticLiveChan staticEnv entr exit nc (Bind n (Case ns nl tl nr tr) t')
* app: exit n entr staticEnv nc na nf t' .
  (exit (NBnd n) \ {n}) ⊆ entr (NBnd n),
  (staticBuiltOnChan staticEnv nc na
    → {na} ⊆ entr (NBnd n)),
  (staticBuiltOnChan staticEnv nc nf
    → {nf} ⊆ entr (NBnd n)),
  entr (termId t') ⊆ exit (NBnd n),
  staticLiveChan staticEnv entr exit nc t'
  → staticLiveChan staticEnv entr exit nc (Bind n (App nf na) t')

```

The static liveness of a flow checks if a flow exists in a whole graph, and if it meets certain criteria with respect to the entry and exit liveness functions. The channel liveness relation says if a channel is live within a function, but does not say anything whether or not a channel remains live across calling or returning flows. The liveness of flows can check if the liveness is maintained across function boundaries, in addition to simply lifting the static liveness the analysis for the intraprocedural flows. No channel is considered to be live in the body of the loop at **bind**  $\lambda p$ . However, a channel may be live before the loop is called and after the loop returns. In such a case, the live flow is retained from the caller to within the loop function's body, even though the steps in the loop may not be live according to the entry function in the static channel liveness.

```

predicate staticLiveFlow:
  graph -> term_id_map -> term_id_map -> flow -> bool
where
  seq: l l' graph exit entr .
    (l, MNxt, l') ∈ graph,
    ⊃ (exit l = {}),
    ⊃ (entr l' = {})
  ⊢ staticLiveFlow graph entr exit (l, MNxt, l')
* spawn: l l' graph exit entr .
  (l, MSpwn, l') ∈ graph,
  ⊃ (exit l = {}),
  ⊃ (entr l' = {})
  ⊢ staticLiveFlow graph entr exit (l, MSpwn, l')

```

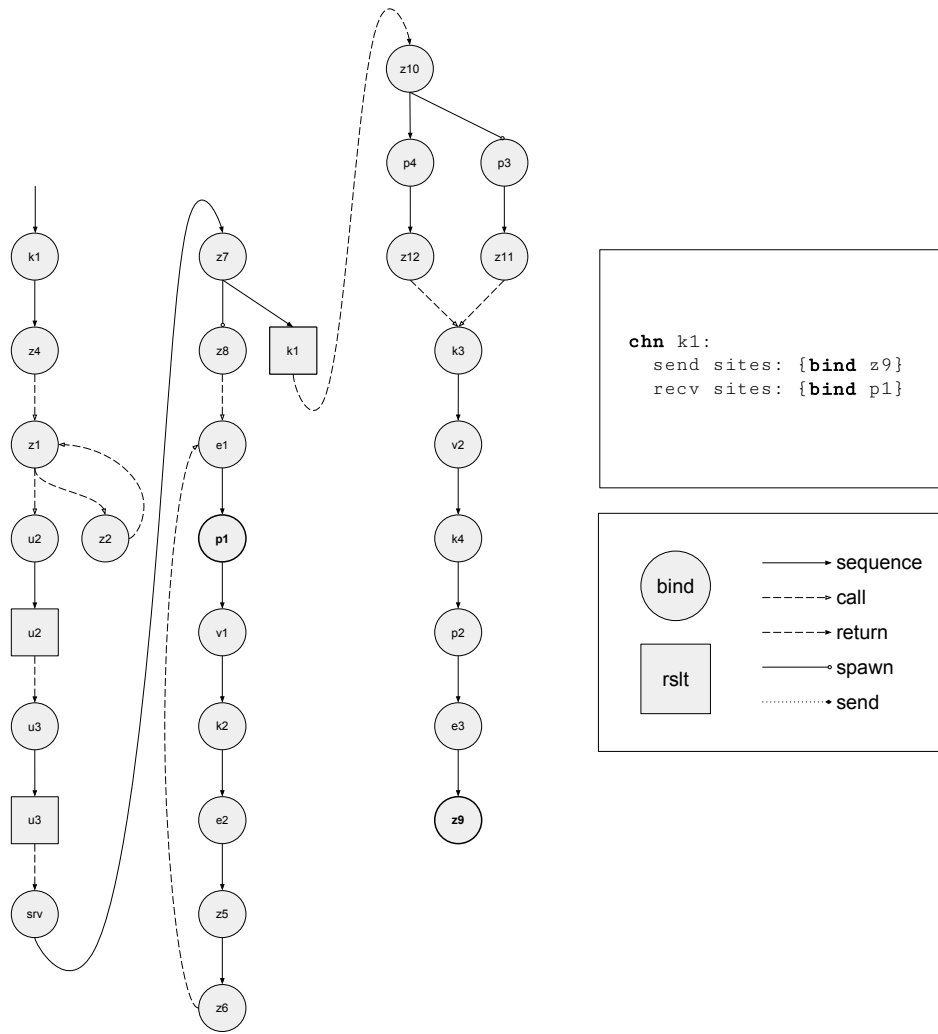
```

* call: l l' graph exit entr .
  (l, MCll, l') ∈ graph,
  (¬ (exit l = {})) ∨ (¬ (entr l' = {}))
  ⊢ staticLiveFlow graph entr exit (l, MCll, l')
* return: l l' graph entr exit .
  (l, MRtn, l') ∈ graph,
  ¬ (entr l' = {})
  ⊢ staticLiveFlow graph entr exit (l, MRtn, l')
* send: ns ne nr graph entr exit .
  ((NBnd ns), ESend ne, (NBnd nr)) ∈ graph,
  {ne} ⊆ (entr (NBnd ns))
  ⊢ staticLiveFlow graph entr exit
    ((NBnd ns), ESend ne, (NBnd nr))

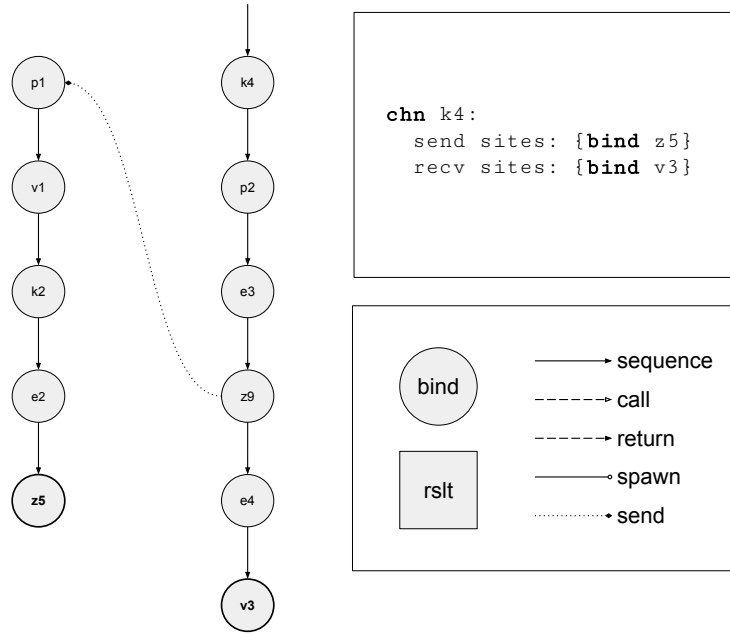
```

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

The following diagram illustrates the graph of the server example, containing only live flows for channel k1.



The following diagram illustrates the graph of the server example, containing only live flows for channel k4.



```

predicate staticTraceable:
  static_value_map -> graph -> term_id_map -> term_id_map
  -> termId -> (termId -> bool) -> static_path -> bool
where
  empty: isEnd start staticEnv graph entr exit .
  isEnd start
  ⊢ staticTraceable graph entr exit start isEnd []
  * snoc: graph entr exit start middle path isEnd mode.
    staticTraceable graph entr exit start (λ l . l = middle) path,
    (isEnd end),
    staticLiveFlow graph entr exit (middle, mode, end)
    ⊢ staticTraceable graph entr exit start isEnd (path @ [(middle, mode)])

```

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

```

predicate staticInclusive: static_path -> static_path -> bool where

```

```

ordered: path1 path2 .
  prefix path1 path2 ∨ path2 path1
⊢ staticInclusive path1 path2
* spawnLeft: path n path1 path2 .
  ⊢ staticInclusive
    (path @ (NBnd n, MSpwn) # path1)
    (path @ (NBnd n, MNxt) # path2)
* spawnRight: path n path1 path2 .
  ⊢ staticInclusive
    (path @ (NBnd n, MNxt) # path1)
    (path @ (NBnd n, MSpwn) # path2)
* sendLeft: path n path1 path2 .
  ⊢ staticInclusive
    (path @ (NBnd n, ESend xE) # path1)
    (path @ (NBnd n, MNxt) # path2)
* sendRight: path n path1 path2 .
  ⊢ staticInclusive
    (path @ (NBnd n, MNxt) # path1)
    (path @ (NBnd n, ESend xE) # path2)

predicate singular: static_path -> static_path -> bool where
  refl: path .
    singular path path
* notInclus: path1 path2 .
  ⊢ (staticInclusive path1 path2)
  ⊢ singular path1 path2

predicate noncompetitive: static_path -> static_path -> bool where
  ordered: path1 path2 .
    ordered path1 path2
  ⊢ noncompetitive path1 path2
* notInclus: path1 path2 .
  ⊢ (staticInclusive path1 path2)
  ⊢ noncompetitive path1 path2

```

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

```

predicate staticOneToMany: static_value_map -> term -> name -> bool where
  intro: staticEnv staticComm t graph entr exit nc
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    staticLiveChan staticEnv entr exit nc t,
    forEveryTwo (staticTraceable graph entr exit (NBnd nc)
      (staticSendId staticEnv t nc)) noncompetitive
  ⊢ staticOneToMany staticEnv t nc

```

```

predicate staticManyToOne: static_value_map -> term -> name -> bool where
  intro: staticEnv staticComm t graph entr exit nc
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    staticLiveChan staticEnv entr exit nc t,
    forEveryTwo (staticTraceable graph entr exit (NBnd nc)
      (staticRecvId staticEnv t nc)) noncompetitive
  ⊢ staticManyToOne staticEnv t nc

predicate staticOneToOne: static_value_map -> term -> name -> bool where
  intro: staticEnv staticComm t graph entr exit nc
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    staticLiveChan staticEnv entr exit nc t,
    forEveryTwo (staticTraceable graph entr exit (NBnd nc)
      (staticSendId staticEnv t nc)) noncompetitive,
    forEveryTwo (staticTraceable graph entr exit (NBnd nc)
      (staticRecvId staticEnv t nc)) noncompetitive
  ⊢ staticOneToOne staticEnv t nc

predicate staticOneShot: static_value_map -> term -> name -> bool where
  intro: staticEnv staticComm t graph entr exit nc .
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    staticLiveChan staticEnv entr exit nc t,
    forEveryTwo (staticTraceable graph entr exit (NBnd nc)
      (staticSendId staticEnv t nc)) singular
  ⊢ staticOneShot V t nc

predicate staticOneSync: static_value_map -> term -> name -> bool where
  intro: staticEnv staticComm t graph entr exit nc .
    staticEval staticEnv staticComm t,
    staticFlowsAccept staticEnv graph t,
    staticLiveChan staticEnv entr exit nc t,
    forEveryTwo (staticTraceable graph entr exit (NBnd nc)
      (staticSendId staticEnv t nc)) singular,
    forEveryTwo (staticTraceable graph entr exit (NBnd nc)
      (staticRecvId staticEnv t nc)) singular
  ⊢ staticOneSync V t nc

```



## 4.1 Higher Precision Soundness Proof Strategy

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

Essentially, it will be necessary to show that static properties that hold for some static path are preserved for corresponding dynamic paths. However, in the higher precision analysis these paths correspond modulo the channel of interest. Let's outline the derivation of soundness of one-shot classification.

```
theorem staticOneShotSound:  $t_0 \ n_c \ \text{path}_c$  .  
  staticOneShot  $t_0 \ n_c$   
   $\vdash$  oneShot  $t_0 \ (\text{Chan } \text{path}_c \ n_c)$ 
```

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

```
predicate pathsCorrespond: dynamic_path -> static_path -> bool where  
  empty: pathsCorrespond [] []  
  * seq: path staticPath n .  
    pathsCorrespond path staticPath  
     $\vdash$  pathsCorrespond  
      (path @ [DNxt n])  
      (staticPath @ [(NBnd n, MNxt)])  
  * spawn: path staticPath n .  
    pathsCorrespond path staticPath  
     $\vdash$  pathsCorrespond  
      (path @ [DSpwn n])  
      (staticPath @ [(NBnd n, MSpwn)])  
  * call: path staticPath n .  
    pathsCorrespond path staticPath  
     $\vdash$  pathsCorrespond  
      (path @ [DCll n])  
      (staticPath @ [(NBnd n, MCll)])  
  * return: path staticPath n .  
    pathsCorrespond path staticPath  
     $\vdash$  pathsCorrespond  
      (path @ [DRtn n])  
      (staticPath @ [(NRslt n, MRtn)])
```

```

predicate pathsCorrespondModChan:
  pool -> communication -> chan -> dynamic_path -> static_path -> bool
where
  seq: pool pathc nc pathsfx stt staticPath comm .
    pool (pathc @ (DNxt nc) # pathsfx) = Some stt,
    pathsCorrespond ((DNxt nc) # pathsfx) staticPath
  ⊢ pathsCorrespondModChan
    (pool, comm) (Chan pathc nc)
    (pathc @ (DNxt nc) # pathsfx) staticPath
  * send: pool pathr nr pathsfx stt paths ns nse tsy envsy stacksy
    nre try envry stackry cc comm c staticPathre staticPathsfx .
    pool (pathr @ (DNxt nr) # pathsfx) = Some stt,
    pool paths = Some (Stt (Bind ns (Sync nse) tsy) envsy stacksy),
    pool pathr = Some (Stt (Bind nr (Sync nre) try) envry stackry),
    {(paths, cc, pathr)} ⊆ comm,
    dynamicBuiltOnChanVar envry c nr,
    pathsCorrespondModChan pool comm c paths staticPathpf,
    pathsCorrespond pathsfx staticPathsfx
  ⊢ pathsCorrespondModChan pool comm c
    (pathr @ (DNxt nr) # pathsfx)
    (staticPathpf @ [(NBnd ns, ESend nse), (NBnd nr, MNxt)] @ staticPathsfx)

```

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

```

Lemma staticTraceableSound: t0 pool comm path n c t' env stack staticEnv
  staticComm
  entr exit nc graph isEnd pathc .
  star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
  pool path = Some (Stt (Bind n c t') env stack),
  staticEval staticEnv staticComm t0,
  staticLiveChan staticEnv entr exit nc t0,
  staticFlowsAccept staticEnv graph t0,
  isEnd (NBnd n)
  ⊢ exists staticPath .
  pathsCorrespondModChan pool comm (Chan pathc nc) path staticPath
  ∧ staticTraceable graph entr exit (NBnd nc) isEnd staticPath

```

```

lemma staticInclusiveSound: t0 pool comm staticEnv entr exit nc graph staticComm
path1 stt1 pathc staticPath1 path2 stt2 staticPath2 .
  star dynamicEval [[] -> (Stt t0 [->] [])] {} pool comm,
  staticLiveChan staticEnv entr exit nc t0,
  staticFlowsAccept staticEnv graph t0,
  staticEval staticEnv staticComm t0,
  pool path1 = Some stt1,
  pathsCorrespondModChan pool comm (Chan pathc nc) path1 staticPath1,
  staticTraceable graph entr exit
    (NBnd nc) (staticSendId staticEnv t0 nc) staticPath1,
  pool path2 = Some stt2,
  pathsCorrespondModChan pool comm (Chan pathc nc) path2 staticPath2,
  staticTraceable graph entr exit
    (NBnd nc) (staticSendId staticEnv t0 nc) staticPath2
  ⊢ staticInclusive staticPath1 staticPath2

```

## 5 Related Work

There has been much research on both dynamic and static analysis of concurrency languages. The formal communication classification analysis and soundness proofs in this work are based on the analysis and proofs of *Specialization of CML message-passing primitives* by Reppy and Xiao [17]. The mechanization of concurrency analyses is prevalent, and mechanization is typically the main goal when developing the analyses. Examples include type checkers in compilers, model checking tools for concurrency models, such as Lustre [9] and Kind [20], and also verification libraries in proof assistants, such as Affeldt et al’s Coq library [2]. These systems can verify certain properties of concurrency programs or models, but they don’t make any guarantees about the analysis itself. Rather than focus on mechanizing the analysis, this work has focused on mechanizing the theory of analyses for concurrency languages, i.e. the meta-theory of concurrency. There have been a number of works on the meta-theory of Concurrent ML, such as the work of Reppy and Xiao, Nielson et al [15], Kobayashi et al [11], and Gasser et al [7]. There has been relatively little work to mechanize theories of Concurrent ML; however, there has been much work in the mechanization of the theories of  $\pi$ -calculus [13], such as the work by Gay [8] and Melham [12].

## 6 Future Work

The formal syntax, semantics, and communication analysis of this work form the basis of a framework for studying concurrency functions, synchronization mechanisms, and their applications. These language features enable the construction of reactive programs, which have separation of parts that are conceptually distinct, yet still depend on each other.

This work has kicked off the framework with a formal communication analysis that has practical applications in aiding optimizations for parallel computation. In the future, additional analyses

could be built on the existing semantics, in order to verify the correctness of language extensions or optimizations. Extending the semantics to handle event combinators for choosing events, sequencing events, guarding events, among others, would be an important next step.

Concurrency is a double edged sword. Without specification of ordering, programs may describe their behavior more clearly or allow parallelism for faster execution. On the other hand, unspecified orderings may also lead to nondeterministic behavior, which may be not be wanted. To gain the benefits of concurrency without its hinderance, the language could be extended with syntax to identify blocks of code that are required to be deterministic, along with a corresponding static analysis that checks if such code is actually deterministic. The determinism analysis could rely on the static communication analysis to ensure that all synchronized receiving events receive from at most one channel, that channel is sent on by at most one thread, and that thread is also deterministic.

Other analyses could aid optimizations for incremental computation [1]. One possible optimization could transform a program into one that checks for altered dependencies and only recomputes the data that depends on altered dependencies.

## References

- [1] Umut A Acar, Guy E Blelloch, and Robert Harper. *Adaptive functional programming*, volume 37. ACM, 2002.
- [2] Reynald Affeldt and Naoki Kobayashi. A coq library for verification of concurrent programs. *Electronic Notes in Theoretical Computer Science*, 199:17–32, 2008.
- [3] Andrew Appel and John Reppy. Sml/nj. Accessed: 2018-12-04.
- [4] Kevin Donnelly and Matthew Fluet. Transactional events. *Journal of Functional Programming*, 18(5-6):649–706, 2008.
- [5] Matthias Felleisen and Daniel P Friedman. *Control Operators, the SECD-machine, and the [1]-calculus*. Indiana University, Computer Science Department, 1986.
- [6] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.
- [7] Kirsten L Solberg Gasser, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for cml. In *ACM SIGPLAN Notices*, volume 32, pages 38–51. ACM, 1997.
- [8] Simon J Gay. A framework for the formalisation of pi calculus type systems in isabelle/hol. In *International Conference on Theorem Proving in Higher Order Logics*, pages 217–232. Springer, 2001.
- [9] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
- [10] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [11] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *International Static Analysis Symposium*, pages 225–242. Springer, 1995.
- [12] Thomas F. Melham. A mechanized theory of the pi-calculus in hol. *Nord. J. Comput.*, 1(1):50–76, 1994.
- [13] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.
- [14] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

- [15] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 84–97. ACM, 1994.
- [16] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
- [17] John Reppy and Yingqi Xiao. Specialization of cml message-passing primitives. In *ACM SIGPLAN Notices*, volume 42, pages 315–326. ACM, 2007.
- [18] John H Reppy. *Concurrent programming in ML*. Cambridge University Press, 2007.
- [19] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.
- [20] Cesare Tinelli. Kind. Accessed: 2018-12-04.
- [21] Stephen Weeks, Matthew Fluet, Henry Cejtin, and Suresh Jagannathan. Mlton. Accessed: 2018-12-04.