Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

August 4, 2018

1 Summary

The goal of this master's thesis is to the develop formal and mechanically verified proofs of useful properties about communication in Concurrent ML[8, 9]. This work will build on Reppy and Xiao's static analysis for computing sound approximations of communication topologies[11, 16]. I will define a small-step operational semantics for Concurrent ML and a constraint-based static analysis[5] that describes all possible communications with varying precision. I will prove that the analysis is sound with respect to the semantics. The semantics, analysis, propositions, proofs, and theorems will rely on Isabelle/HOL[6, 7, 14, 15] as the formal language of reasoning. The proofs will be mechanically checked by Isabelle[17].

2 Overview

Concurrent programming languages provide features to specify a range of evaluation orders between steps of distinct expressions. The freedom to choose from a number of possible evaluation orders has certain advantages. Conceptually distinct tasks may need to overlap in time, but are easier to understand if they are written as distinct expressions. Concurrent languages may also allow the evaluation order between steps of expressions to be nondeterministic or unrestricted. If it's not necessary for tasks to be ordered in a precise way, then it may be better that the program allow arbitrary ordering and let a scheduler find an execution order based on runtime conditions and policies of fairness. A common use case for concurrent languages is GUI programming, in which a program has to process various requests while remaining responsive to subsequent user inputs and continually providing the user with the latest information it has processed. Concurrent ML is a concurrent programming language. It offers a thread abstraction, which is a piece of code allowed to have a wide range of evaluation orders relative to code encapsulated in other threads. The language provides a synchronization mechanism that can specify the execution order between parts of expressions in separate threads. It is often the case that synchronization is necessary when data is shared. Thus, in Concurrent ML, synchronization and data sharing mechanisms are actually subsumed by a uniform communication mechanism. Additional thread abstractions can be used for sharing data asynchronously, which can provide better usability or performance in some instances. A thread in Concurrent ML, is created using the spawn primitive.

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.

A given channel can have any arbitrary number of threads sending or receiving data on it over the course of the program's execution. Listing 1 and Listing 2 give a simple example derived from Reppy's book Concurrent Programming in ML[8] that illustrates these essential features of Concurrent ML.

The server implementation, given in Listing 2, defines a server that holds a number state. When a client gives the server a number v, the server gives back state, and holds onto v to as its new state, which it gives to the next client and so on. A request and reply is equivalent to reading and writing a mutable cell in isolation. The function make makes a new server. It creates a new channel reqCh, from which the server will receive requests. The sever behavior is defined by the infinite loop loop, which takes a number as the state of each iteration. Each iteration, the server tries to receive requests on reqCh. It expects the request to be composed of a number v and a channel replCh, through which to reply. Once a request has been received, it sends the current state back to the client through replCh by calling send (replCh, state). It initiates the next iteration of the loop by calling loop with a new state from the client. The server is created with a new thread by calling spawn (fn () =¿ loop 0). A handle to the new server is returned as reqCh wrapped in the constructor S. The function call makes a request to a server server with a number v and returns the number from the server's reply. It extracts the request channel reqCh from the server handle and creates a new channel replCh, from which the client will receive replies. 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. Reppy's original design of Concurrent ML allows for events other than sending and receiving to be triggered by synchronization. One such event chooses between one of many events to synchronize on. Only one of the events is chosen for synchronization, but all choices must be represented. Thus, event synchronization must be separated from event values, similar to the way function application is separated from function abstraction. send and recv are just shorthand for synchronization on send and receive events, respectively.

choose is an example of an event combinator; a way to construct an event from other events. Reppy's book on Concurrent ML offers explanations of many other useful combinators, such as the wrap and guard combinators[8]. Donnelly and Fluet extended Concurrent ML with the transactional event combinator thenEvt[1]. Transactional events provide a technique for describing tasks that sometimes execute in isolation and sometimes don't. Achieving similar results without transactional events would require duplication of code in multiple threads, resulting in code that is brittle under modification.

When thenEvt is synchronized on, either all of its constituent events and abstractions evaluate in isolation, or none evaluate.

A uniprocessor implementation of synchronous communication is inexpensive. Using a fairly course-grain interleaving, the communication on a channel can proceed by checking if the channel is in one of two possible states: either a corresponding thread is waiting or there's nothing waiting. The implementation doesn't need to consider states where competing threads are also trying to communicate on the same channel, since the course-grain interleaving ensures that competing threads have made no partial communication progress. In a multiprocessor setting, threads can run in parallel and multiple threads can simultaneously make partial progress on the same channel. The multiprocessor implementation of communication is more expensive than that of the uniprocessor, since it must consider additional states related to competing threads making partial communication progress.[10] Channels known to have only one sender or one receiver can have lower communication costs than those with arbitrary number of senders and arbitrary number of receivers, since some of the cost of handling competing threads can be avoided. Concurrent ML does not provide language features for multiple types of channels distinguished by their communication topologies, or the number of threads that may end up sending or receiving on it. However, channels can be classified into various topologies based on their potential communication. A many-to-many channel has any number of senders and receivers; a fan-out channel has one sender and any number of receivers; a fan-in channel has any number of senders and exactly one receiver; a one-to-one channel has exactly one of each; a one-shot channel has exactly one sender, one receiver, and sends data only once. The server implementation in Listing 2 with the following calling code exhibits these topologies.

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

A program analysis that describes communication topologies of channels has practical benefits in at least two ways. It can highlight which channels are candidates for optimized implementations of communication; or in a language extension allowing the specification of restricted channels, it can conservatively verify the correct usage of restricted channels. Listing 2 demonstrates the language extension based on an example from Reppy and Xiao[11].

Without a static analysis to check the usage of the special channels, one could inadvertently use a one-shot channel for a channel that has multiple senders, resulting in runtime behavior inconsistent with the general semantics of channel synchronization. The utility of the program analysis additionally depends on it being informative, sound, and computable. The analysis is informative iff there exist programs about which the analysis describes information that is not directly observable. The analysis is sound iff the information it describes about a program is the same or less precise than the operational semantics of the program. The analysis is computable iff there exists an algorithm that determines all the values described by the analysis on any input program.

Program analyses, like operational semantics, describe information about the execution or behavior of programs. Yet, while an operational semantics may be viewed as ground truth, the correctness of an analysis is derived from its relation to an operational semantics. In practice, program analyses often describe computable information with respect to operational semantics that are universal and capable of describing uncomputable information. To allow for computability, program analyses often describe approximate information. There are a large number of program analyses with a variety of practical uses. Some constructions of programs might be considered bad, by describing operations that don't make sense, like True * 5 / "hello", or accessing the 7th element of an array with 6 elements. A type systems, or static semantics, is an analysis that can help ensure programs are well constructed. It describes how programs and expressions can be composed, such that the programs won't get stuck or result in certain kinds of undesired behavior. Type systems can improve debugging by pointing out errors that may be infrequently executed. They can also improve execution speeds of safe languages by rendering some runtime checks unnecessary. Other analyses are useful for describing opportunities for program optimizations. Many analyses used for optimizations describe how data flows with information related to every point in the program. Each point refers to a term, from which the small-step semantics may take a step. Some programs may mention the same expression multiple times, possibly resulting in redundant computations. These redundant computations can be detected by available expressions analysis, one of many data flow analyses. An available expressions analysis describes which expressions must have been computed by each program point.

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

Since the variables x and z and the dereference !z are used in line 8, they are live at line 7. Since z is reassigned at line 7, !z is no longer live at line 6. Since x is bound at line 5 and not used above, it is not live at line 4 and above. Since z is bound at 4 and not used above, it is not live at line 3 and above. The liveness information demonstrates that the expression (4 * 73) doesn't need to be computed, and lines 2 and 3 can simply be removed. The information at each program point is derived from control structures in the program that dictate how information may flow between program points. Some uses of control structures are represented as literals in the syntax, while other uses are expressions that may evaluate to control structures, or function parameters that may bind to control structures. Function abstraction is a control structure allowing multiple parts of a program to flow into a section of code via a binding. In ML, function abstractions are higher order, and may be unknown without some form of evaluation. These control structures may be revealed by an abstract value flow analysis, which associates each program point with a set of abstract values that the point's expression may evaluate to.

The abstract values of f, g, h are simply their let bound expressions fn x =¿ x 1, fn y =¿ y + 2, fn z =¿ z + 3, respectively. x has the abstract values of fn y =¿ y + 2, fn z =¿ z + 3, so x 1 has the abstract values of 3, 4; (f g) has abstract values of 3, 4. Since the abstract values depend on the flow of information, which depends on the abstract values, the description of abstract values is inductive or recursive. The historical motivation for describing the abstract value information was really for its the control information, so the original approaches to these analyses are known as control flow analyses or

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

For a subset of Concurrent ML without event combinators, Reppy and Xiao developed an efficient algorithmic analysis that determines for each channel all abstract threads that send and receive on it. The algorithm depends on each primitive operation in the program being labeled with a program point. A sequence of program points ordered in a valid execution sequence forms a control path. Distinction between threads in a program can be inferred from whether or not their control paths diverge. The algorithm proceeds in multiple steps that produce intermediate data structures, used for efficient lookup in the subsequent steps. It starts with a control-flow analysis[12, 13] that results in multiple mappings. One mapping is from variables to abstract values that may bind to the variables. Another mapping is from channel-bound variables to abstract values that are sent on the respective channels. Another is from function-bound variables to abstract values that are the result of respective function applications. It constructs a control-flow graph with possible paths for pattern matching and thread spawning determined directly from the primitives used in the program. Relying on information from the mappings to abstract values, it constructs the possible paths of execution via function application and channel communication. It uses the graph for live variable analysis of channels, which limits the scope for the remaining analysis. Using the spawn and application edges of the control-flow graph, the algorithm then performs a data-flow analysis to determine a mapping from program points to all possible control paths leading into the respective program points. Using the CFA's mappings to abstract values, the algorithm determines the program points for sends and receives per channel variable. Then it uses the mapping to control paths to determine all control paths that send or receive on each channel, from which it classifies channels as one-shot, one-to-one, fan-in, fan-out, or many-to-many. Reppy and Xiao informally prove soundness of their analysis by showing that their analysis claims that more than one thread sends (or receives) on a channel if the execution allows more than one to send (or receive) on a that channel. The proof of soundness depends on the ability to relate the execution of a program to the static analysis of a program. The static analysis describes threads in terms of control paths, since it can only describe threads in terms of statically available information. Thus, in order to describe the relationship between the threads of the static analysis and the operational semantics, the operational semantics is defined as stepping between sets of control paths paired with terms. Divergent control paths are added whenever a new thread is spawned.

The syntax, semantics, and analysis need to describe many details. Proving propositions relating all of these definitions requires manipulation of all those details. To ensure the correctness of proofs, it is necessary to check that there are no subtle errors in the definitions or proofs. Proofs in general require many subtle manipulations of symbols. The difference between a false statement and a true statement can often be difficult to spot, since the two may be very similar lexically. However, a mechanical proof checker, such as the one in Isabelle, has no difficulty discerning between valid and invalid derivations of statements. Mechanical checking of proofs can notify us of errors in the proofs or definitions far better and faster than manual checking. I have already benefitted from Isabelle's proof checker in order to correctly define the language semantics and abstract value flow analysis for this work. While trying to prove soundness of the analysis, the proof assistant would not accept my proof unless I provided derivation of facts that I believed to be false. I determined that my intuition was correct but my definitions had errors. After correcting the errors, I was able to complete the proof, such that the proof checker was satisfied. Although Isabelle is described as a proof assistant[17], it is really a generic system for processing any kind of code. The code could be proofs, propositions, programs, or types. The processing could be checking proofs, interpreting programs, or translating code. Code and logics for processing code are defined by users using its meta-language Standard ML, and other user-defined languages. Isabelle/HOL is a higher-order logic built from Isabelle's primitives and other logics. It is useful for both programming and proving. Its ability to check that proofs satisfy propositions is simply one instance of its verification capabilities. It can also check that program terms satisfy types, similar to other programming systems for ML. Proofs and propositions are analogous to terms and types, respectively, yet Isabelle/HOL treats the two concepts distinctly. The practical uses for terms are quite different from that of proofs. If a term satisfies a type, then the term has utility for the data or computation it represents. The type is only valuable for confirming or denying the usage of a term. In contrast, once a proof satisfies a proposition, the proof becomes irrelevant, while the proposition is elevated to a theorem. The theorem is useful on its own without regard to any particular proof. Similar to other programming languages, type bool can be satisfied by values True or False. In contrast to other programming languages, additional syntax, or data constructors, can be defined to satisfy the type bool. A constructor can take any number of terms of any types as input in order to create a boolean term. Although these new terms could be used in programs, just as True and False are, their main utility is in theorem proving. In Isabelle/HOL, propositions are isomorphic to terms of type bool. The constructors are defined with a set of inference rules, where each inference rule defines the conditions sufficient for a construction to be valid, and at least one of the enumerated conditions is necessary for a valid construction . In other words, the constructor is equivalent to the boolean sum of all the conditions. The proposition definitions in terms of inference rules, or inductive definitions, are analogous to datatype definitions, just as propositions are analogous to types.

The definitions of list and sorted can be combined with definitions of natural numbers to form propositions. Note that Isabelle/HOL's list is defined with syntactic sugar. hd # tl can be used instead of Cons hd tl, and [a, b, c] is a # b # c # Nil. In Isabelle/HOL, propositions may be proved by applying the inference rules. The method rule is used to work backwards from the goal until no further conditions need to be satisfied. Theorems may also be proved forwards from axioms, theorems or assumptions to the goal, using other methods like drule or erule.

In truth, True and False are not primitive values, but actually just named instances of other propositions converted to boolean terms. False is defined to be the absurd statement that all propositions are valid.

3 Hypothesis

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

4 Evaluation

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

5 Architecture

To enable mechanical verification of the correctness of the proofs, I will construct the semantics, analysis and theorems in the formal language of Isabelle/HOL. To aid the development of formal proofs, I will design the analysis as a declarative specification as opposed to an algorithm. However, the declarative analysis will make the proof of computability less direct. To aid the scrutiny of the theorems' adequacy, I will express the definitions and propositions with the fewest number of structures, judgements, inferences rules, and axioms necessary. Efficiency of computation will be ignored in favor of verification. I will not rely on intermediate map or graph data structures, which Reppy and Xiao used for efficient computation. In order to relate the analysis to the operational semantics, I will borrow Reppy and Xiao's strategy of stepping between sets of control paths tied to terms. In this thesis work, I'm interested in communication topology soundness, rather than flow soundness. Nevertheless, I will need to prove additional flow soundness theorems en route to proving communication topology soundness. Restricting the grammar to a form that requires every abstraction and application to be bound to a variable would allow the operational semantics to maintain static term information necessary for proofs of flow soundness[3, 5]. The semantics would be defined as an environment based operational semantics, rather than a substitution based operational semantics. By avoiding simplification of terms in the operational semantics, it will be possible to relate the abstract values of the analysis to the values produced by the operational semantics, which in turn is relied on to prove flow soundness. I will incorporate the restricted grammar and the environment based semantics into this work. The restricted grammar is impractical for a programmer to write, yet it is still practical for a language under automated analysis since there is a straight forward procedure to transform more flexible grammars into the restricted form as demonstrated by Flanagan et al [2]. Additionally, the restricted grammar melds nicely with the control path semantics. Instead of defining additional meta-syntax for program points of primitive operations, I can simply use the required variables of the restricted grammar to identify program points, and control paths will simply be sequences of let bound variables. A modification of Listing 2 illustrates the restrictive grammar applied to Concurrent ML.

6 Implementation

We describe possible implementations of specialized and unspecialized Concurrent ML using feasible low-level thread-centric features such as wait and poll. The thread-centric approach allows us to focus on optimizations common to many implementations by decoupling the implementation of communication features from thread scheduling and management. Depending on the low level features provided by existing language implementations, Concurrent ML could be implemented in terms of lower level features, as is the case in SML/NJ and MLton. It could also be implemented as primitive features within a compiler and runtime or interpreter. Analyzing and optimizing Concurrent ML would require treating the language as an object, so implementing its features as primitives would make the most sense. Thus, one can think of the implementation shown here as an intermediate representation presented with concrete syntax.

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

A channel can be in one of three states. Either some threads are trying to send through it, some threads are trying to receive from it, or no threads are trying to send or receive. Additionally a channel is composed of a mutex lock, so that send and recv operations can yield to each other when updating the channel state. When multiple threads are trying to send on a channel, the channel is associated with a queue consisting of messages to be sent, along with conditions waited on by sending threads. When multiple threads are trying to receive on a channel, the channel is associated with a queue consisting of initially empty cells accessible by receiving threads and conditions waited on by the receiving threads. The three states are represented by the datatype chan_content. The channel is represented by the chan datatype, which is composed of a reference to chan_content and a mutex lock. The send operation acquires the channel's lock to ensure that it updates the channel based on any one of its latest state. If there are threads trying to receive from the channel, the send operation dequeues an item from the state's associated queue. The item consists of a condition waited on by a receiving thread and an empty cell that can be accessed by the receiving thread. It deposits the message in the cell and signals on the condition, updates the channel state to inactive if there are no further receiving threads waiting, then releases the lock, signals on the condition and returns the unit value. If there are no threads trying to receive from the channel, the send operation updates the channel state to that of trying to send with an additional condition and message in the associated queue. It releases the lock and waits on the enqueued condition. Once a receiving thread signals on the same condition, the send operation returns with the unit value. The recv operation acquires the channel's lock to ensure that it updates the channel based on any one of its latest state. If there are threads trying to send on the channel, the recv operation dequeues an item from the state's associated queue. The item consists of a condition waited on by a sending thread along with a message to be sent. It signals the condition and updates the channel state to inactive if there are no further sending threads waiting, then releases the lock and returns the sent message. If there are no threads trying to send on the channel, the recv operation updates the channel state to that of trying to receive with an additional condition and empty cell in the associated queue. It releases the lock and waits on the enqueued condition. Once a sending thread signals on the same condition, the recv operation returns with the value deposited in the cell by a sending thread.

Implementation of fan-out channels, compared to that of many-to-many channels, requires fewer steps to synchronize and can execute more steps outside of critical regions, which reduces contention for locks. A channel is composed of a lock and one of three possible states, as is the case for many-to-many channels. However, the state of a thread trying to send need only be associated with one condition and one message. The send operation checks if the channel's state is inactive and tries to use the compareAndSwap operator to transactionally update the state of the channel to that of trying to send. If successful, it simply waits on sendCond, the condition that a receiving thread will signal on, and then returns the unit value. If the transactional update fails and the state is that of threads trying to receive on the channel, then the send operation acquires the lock, then dequeues an item from the associated queue where the item consists of recvCond, a condition waited on by a receiving thread, and a cell for depositing the message to that receiving thread. It deposits the message in the cell, updates the state to inactive there are no further items on the queue, then releases the lock. Then it signals on the condition and returns the unit value. The lock is acquired after the state is determined to be that of threads trying to receive, since the expectation is that the current thread is the only one that tries to update the channel from that state. If the communication topology analysis were incorrect and and there were actually multiple threads that could call the send operation, then there might be data races. Likewise, due to the expectation of a single thread sending on the channel, the send operation should never witness the state of threads already trying to send. The recv operation acquires the lock and checks the state of the channel, just as the recv operation for many-to-many channels. If the channel is in a state where there is no already trying to send, then it behaves the same as the recv operation of many-to-many channels. If there is already a thread trying to receive, then it updates the state to inactive and releases the lock. Then it signals on the state's associated condition, which is waited on by a sending thread, and returns the states' associated message.

The implementation of fan-in channels is very similar to that of fan-out channels.

a one-to-one channel can also be in one of three possible states, but there is no associated lock. Additional, none of the states are associated with queues. Instead, there is a possible state of a thread trying to send, with a condition and a message, or a possible state of a thread trying to receive with a condition and an empty cell, or a possible inactive state. The send operation checks if the channel's state is inactive and tries to use the compareAndSwap operator to transactionally update the state of the channel to that of trying to send. If successful, it simply waits on sendCond, the condition that a receiving thread will signal on, and then returns the unit value. If the transactional update fails and the state is that of a thread trying to receive on the channel, then it deposits the message in the state's associated cell, updates the channel state to inactive, then signals on the state's associated condition and returns the unit value. If the communication analysis for the channel is correctly one-to-one, then there should be no other thread trying update the state from the state of a thread trying to receive, and no thread modifies that particular state, so no locks are necessary. Likewise, the send operation should never witness the state of another thread already trying to send, if it is truly one-to-one. The recv operation checks if the channel's state is inactive and tries to use the compareAndSwap operator to transactionally update the state of the channel to that of trying to receive. If successful, it simply waits on recvCond, the condition that a sending thread will signal on after it deposits a message, and then returns the deposited message. If the transactional update fails and the state is that of a thread trying to send on the channel, then it updates the channel state to inactive, then signals on the state's associated condition and returns the message associated with the sending thread. If the communication analysis for the channel is correctly one-to-one, then there should be no other thread trying update the state from the state of a thread trying to send, and no thread modifies that particular state, so no locks are necessary. Likewise, the recv operation should never witness the state of another thread already trying to receive, if it is truly one-to-one.

A one-shot channel consists of the same possible states as a one-to-one channel, but is additionally associated with a mutex lock, to account for the fact that multiple threads may try to receive on the channel, even though only at most one message is ever sent. The send operation is like that of one-to-one channels, except that if the state is that of a thread trying to receive, it simply deposits the message and signals on the associated condition, without updating the channel's state to inactive, which would be unnecessary, since no further attempts to send are expected. The recv operation checks if the channel's state is inactive and tries to use the compareAndSwap operator to transactionally update the state of the channel to that of trying to receive. If successful, it simply waits on recvCond, the condition that a sending thread will signal on after it deposits a message, and then returns the deposited message. If the transactional update fails and the state is that of a thread trying to send on the channel, then it acquires the lock, signals on the state's associated condition and returns the message associated with the sending thread, without ever releasing the lock, so that competing receiving threads will know to not progress. If the state is that of a thread trying to receive on the channel, the it acquires the lock, which should block the current thread forever, if there truly is only one send ever.

An even more restrictive version of a channel with at most one send could be used if it's determined that the number of receiving threads is at most one. The one-shot-to-one channel is composed of a possibly empty cell, a condition for a sending thread to wait on, and a condition for a receiving thread to wait on. The send operation deposits the message in the cell, signals on the recvCond, waits on the sendCond, and then returns the unit value. The recv operation waits on the recvCond, signals on the sendCond and then returns the deposited message.

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

7 Objectives

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

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

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

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

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

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

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

Bound expressions of the unit literal, primitive constructs, and first and second constructs can be evalued to values in one step. The seq step relation describes the evaluation of bound expressions with environments to values. The unit construct simpl evalues to the unit value. A prim constructs is simply wrapped up with the environment. The first construct is evaluated by retrieving the pair associated with the construct's named argument and looking up the value of that pair's first named argument. The second construct is evaluated by retrieving the pair associated with the construct's named argument and looking up the value of that pair's second named argument. Other constructs for spawn, application, and case matching cannot be described with the seq step because they do not evaluate to values in one step and require updating additional information about control flow.

The seq step up relation handles the evaluation of function application and case pattern matching. These bound expressions with associated environments are evalued to full expressions along with a new environment. Pattern matching is evaluated looking up the value of first named argument. If the pattern is a Left construct, then the it's evauluated to the left expression along with an environment updated with a name specified for use in that expression. The value associated with the left case pattern, is bound to the specified name in the new environment. If the pattern is a Right construct, then the it's evauluated to the right expression along with an environment updated with a name specified for use in that expression. The value associated with the right case pattern, is bound to the specified name in the new environment. Function application evaluates to the expression within the applied function abstraction, along with the abstraction's environment with a couple of modifications. The environenment is updated with the recursion paramenter name bound to the abstraction, and the variable parameter bound to the argument specified in the application expression.

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

The concur step relation describes the evaluation of a trace pool to anthoer tracel pool. Trace pools grow montonicially with respect to evaluation, retaining a full history. A trace pool is evaluated to a new trace pool based only on information associated with leaf paths and their associated states. If a leaf path is associated with a bound expresion that can be evaluated to a value in one sequential step, then the trace pool is updated with the leaf path extended by a label indicating a sequential control flow, and the extended path is associated with the next expression to be evaluated and an environment updated the value. If the bound expression can be evalueated to an expression and new environment, the the leaf path is extended with a label indicating a calling control flow. The extended path is associated with the the new expression and environment resulting from the evaluation of the bound expression, and the name and expression of the let binding are pushed onto the stack to be evaluated in later steps. If a leaf path is associated with a result expression, then the trace pool is updated with the leaf path extended with a label indicating a return control flow. The new path is associated with a state whose expression is popped from the continuation stack, and whose environment is popped from the stack and updated with continuation's name bound to the result's associated value. If a leaf path is associated with a channel creation construct, then the trace pool is updated with the leaf path extended with a sequential control flow label, and its associated state contains the next expression, and an environment updated with a new channel value. If a leaf path is associated with a let expression binding to a spawn contstruct, then the trace pool is updated with two new paths extending the leaf path. For one, the leaf path is extended with a sequential label whose state has the next expression and the environment updated with the unit value bound to the let binding name, and the original continuation stack. For the other, the leaf path extended with a label indication spawning control flow. Its state has the spawned expression, the original environment, and an empty continuation stack. If two leaf in the trace pool correspond to synchronization constructs on the same channel, where once synchronizes on a send event and the other synchronizes on a receive event, then the trace pool is updated with two new paths, extended the two synchronizing leaf paths. The leaf path for the send event is extended by a label indicating sequential flow. For its state, the next expression of the let expression is used, and the environment is updated with the unit value bound to the let expression name. The leaf path for the receive event is also extended by a label indicatin sequential control flow. For its state, the next expression of the let expression is used, and the environment is updated with the value associated with the send event's second argument. Additionally, the communication set is updated with the send and receive paths, and the channel that is communicated on.

8 Mathematical Artifacts

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

type 'a chan
val channel : unit -> 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit

```

```

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

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

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

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


```

```

type 'a event
val sync : 'a event -> 'a
val recvEvt : 'a chan -> 'a event
val sendEvt : 'a chan * 'a -> unit event
val choose : 'a event * 'a event -> 'a event

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

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


val server = Serv.make ()
val _ = spawn (fn () => Serv.call (server, 35))
val _ = spawn (fn () =>
  Serv.call (server, 12);
  Serv.call (server, 13))
val _ = spawn (fn () => Serv.call (server, 81))
val _ = spawn (fn () => Serv.call (server, 44))

```

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

  fun make () = let

    val reqChn = FanIn.channel()

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

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

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

  end

```

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

```

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

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

```

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

inductive
  sorted ::
    ('a ⇒ 'a ⇒ bool) ⇒
    'a list ⇒ bool where
  Nil : sorted P Nil |
  Single : sorted P (Cons x Nil) |
  Cons :
    P x y ⟹
    sorted P (Cons y ys) ⟹
    sorted P (Cons x (Cons y ys))
```

```
datatype nat = Z | S nat

inductive
  lte ::
    nat ⇒ nat ⇒ bool where
  Eq : lte n n |
  Lt : lte n1 n2 ⟹ lte n1 (S n2)

theorem "
  sorted lte
    (Cons (Z) (Cons (S Z)
      (Cons (S Z) (Cons
        (S (S (S Z))) Nil))))"
  apply (rule Cons)
  apply (rule Lt)
  apply (rule Eq)
  apply (rule Cons)
  apply (rule Eq)
  apply (rule Cons)
  apply (rule Lt)
  apply (rule Lt)
  apply (rule Eq)
  apply (rule Single)
  done
```

```
definition True :: bool where
  True ≡ ((λx ::bool. x) = (λx. x))

definition False :: bool where
  False ≡ (∀P. P)


```

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

```

structure ManyToManyChan : CHAN = struct
  type message_queue = 'a option ref queue

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

  datatype 'a chan =
    Chn of 'a chan_content ref * mutex_lock

  fun channel () = Chn (ref Inac, mutexLock ())

  fun send (Chn (conRef, lock)) m =
    acquire lock;
    (case !conRef of
      Recv q => let
        val (recvCond, mopRef) = dequeue q in
        mopRef := Some m;
        if (isEmpty q) then conRef := Inac else ();
        release lock; signal recvCond; () end |
      Send q => let
        val sendCond = condition () in
        enqueue (q, (sendCond, m));
        release lock; wait sendCond; () end |
      Inac => let
        val sendCond = condition () in
        conRef := Send (queue [(sendCond, m)]);
        release lock; wait sendCond; () end)

  fun recv (Chn (conRef, lock)) =
    acquire lock;
    (case !conRef of
      Send q => let
```

```
        val (sendCond, m) = dequeue q in
        if (isEmpty q) then
          conRef := Inac
        else
          ();
        release lock; signal sendCond; m end |
      Recv q => let
        val recvCond = condition ()
        val mopRef = ref None in
        enqueue (q, (recvCond, mopRef));
        release lock; wait recvCond;
        valOf (!mopRef) end |
      Inac => let
        val recvCond = condition ()
        val mopRef = ref None in
        conRef := Recv (queue [(recvCond, mopRef)]);
        release lock; wait recvCond;
        valOf (!mopRef) end)

end


```

```

structure FanOutChan : CHAN = struct

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

datatype 'a chan =
  Chn of 'a chan_content ref * mutex_lock

fun channel () = Chn (ref Inac, mutexLock ())

fun send (Chn (conRef, lock)) m = let
  val sendCond = condition () in
  case cas (conRef, Inac, Send (sendCond, m)) of
    Inac => (* conRef already set *)
      wait sendCond; () |
    Recv q =>
    (* the current thread is
      * the only one that updates from this state *)
      acquire lock;
      (let
        val (recvCond, mopRef) = dequeue q in
        mopRef := Some m;
        if (isEmpty q) then conRef := Inac else ();
        release lock; signal (recvCond);
```

```
          () end) |
      Send _ => raise NeverHappens end

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

  end

```

```
structure FanInChan : CHAN = struct

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

datatype 'a chan =
  Chn of 'a chan_content ref * mutex_lock

fun channel () = Chn (ref Inac, mutexLock ())

fun send (Chn (conRef, lock)) m =
  acquire lock;
  case !conRef of
  Recv (recvCond, mopRef) =>
    mopRef := Some m; conRef := Inac;
    release lock; signal recvCond;
    () |
  Send q => let
    val sendCond = condition () in
    enqueue (q, (sendCond, m));
    release lock; wait sendCond;
```

```
      () end |
    Inac => let
      val sendCond = condition () in
      conRef := Send (queue [(sendCond, m)])
      release lock; wait sendCond; () end

  fun recv (Chn (conRef, lock)) = let
    val recvCond = condition ()
    val mopRef = ref None in
    case cas (conRef, Inac, Recv (recvCond, mopRef)) of
      Inac => (* conRef already set *)
        wait recvCond; valOf (!mopRef) |
      Send q =>
        (* the current thread is the only one
        -* that updates the state from this state *)
        acquire lock;
        (let
          val (sendCond, m) = dequeue q in
          if (isEmpty q) then conRef := Inac else ();
          release lock; signal sendCond; m end) |
      Recv _ => raise NeverHappens end end


```

```

structure OneToOneChan : CHAN = struct

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

  datatype 'a chan = Chn of 'a chan_content ref

  fun channel () = Chn (ref Inac)

  fun send (Chn conRef) m = let
    val sendCond = condition () in
    case cas (conRef, Inac, Send (sendCond, m)) of
      Inac =>
        (* conRef already set to Send *)
        wait sendCond; () |
      Recv (recvCond, mopRef) =>
        (* the current thread is the only one
        -* that accesses conRef for this state *)
        mopRef := Some m; conRef := Inac;
        signal recvCond; () |
      Send _ => raise NeverHappens end end


```

```
fun recv (Chn conRef) = let
  val recvCond = condition ();
  val mopRef = ref None in
  case cas (conRef, Inac, Recv (recvCond, mopRef)) of
    Inac => (* conRef already set to Recv*)
      wait recvCond; valOf (!mopRef) |
    Send (sendCond, m) =>
      (* the current thread is the only one
      -* that accesses conRef for this state *)
      conRef := Inac; signal sendCond; m |
    Recv _ => raise NeverHappens end end

end

```

```
structure OneShotChan : CHAN = struct

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

datatype 'a chan = Chn of 'a chan_content ref * mutex_lock

fun channel () = Chn (ref Inac, lock ())

fun send (Chn (conRef, lock)) m = let
  val sendCond = condition () in
  case (conRef, Inac, Send (sendCond, m)) of
    Inac =>
      (* conRef already set to Send*)
      wait sendCond; () |
    Recv (recvCond, mopRef) =>
      mopRef := Some m; signal recvCond;
      () |
    Send _ => raise NeverHappens end end


fun recv (Chn (conRef, lock)) = let
  val recvCond = condition ()
  val mopRef = ref None in
  case (conRef, Inac, Recv (recvCond, mopRef)) of
    Inac =>
      (* conRef already set to Recv*)
      wait recvCond; valOf (!mopRef) |
    Send (sendCond, m) =>
      acquire lock; signal sendCond;
      (* never relases lock;
      -* blocks others forever *)
      m |
    Recv _ =>
      acquire lock;
      (* never able to acquire lock;
      -* blocked forever *)
      raise NeverHappens end end

end

```

```
structure OneShotToOneChan : CHAN = struct

  datatype 'a chan =
    Chn of condition * condition * 'a option ref

  fun channel () =
    Chn (condition (), condition (), ref None)

  fun send (Chn (sendCond, recvCond, mopRef)) m =
    mopRef := Some m; signal recvCond;
    wait sendCond; ()

  fun recv (Chn (sendCond, recvCond, mopRef)) =
    wait recvCond; signal sendCond;
    valOf (!mopRef)

  end

```

9 Syntax

```

  datatype var = Var string

  datatype exp =
    Let var boundexp exp |
    Rslt var

  boundexp =
    Unt |
    MkChn |
    Prim prim |
    Spwn exp |
    Sync var |
    Fst var |
    Snd var |
    Case var var exp var exp |
    App var var and

  prim =
    SendEvt var var |
    RecvEvt var |
    Pair var var |
    Lft var |
    Rht var |
    Abs var var ex
```

10 Dynamic Semantics

```
datatype ctrl_label =
  LNxt var | LSpwn var | LCall var | LRtn var

type_synonym ctrl_path = (ctrl_label list)

datatype chan = Chn ctrl_path var

datatype val =
  VUnt | VChn chan | VClsr prim (var ⇀ val)

datatype ctn = Ctn var exp (var ⇀ val)

datatype state = Stt exp (var ⇀ val) (ctn list)


```

```

inductive
  seq_step ::
    bind * (var ⇀ val)) ⇒ val ⇒ bool where
  LetUnt :
    seq_step (Unt, env) VUnt |
  LetPrim :
    seq_step (Prim p, env) (VClsr p env) |
  LetFst :
    env xp = Some (VClsr (Pair x1 x2) envp) ⟹
    envp x1 = Some v ⟹
    seq_step (Fst xp, env) v |
  LetSnd :
    env xp = Some (VClsr (Pair x1 x2) envp) ⟹
    envp x2 = Some v ⟹
    seq_step (Snd xp, env) v



```

```


inductive
  seq_step_up ::
    bind * (var ⇀ val)) ⇒
    exp * val_env ⇒ bool where
  LetCaseLft :
    env xs = Some (VClsr (Lft xl') envl) ⟹
    envl xl' = Some vl ⟹
    seq_step_up
      (Case xs xl el xr er, env)
      (el, env(xl ↦ vl)) |
  LetCaseRht :
    env xs = Some (VClsr (Rht xr') envr) ⟹
    envr xr' = Some vr ⟹
    seq_step_up
      (Case xs xl el xr er, env)
      (er, env(xr ↦ vr)) |
  LetApp :
    env f = Some (VClsr (Abs fp xp el) envl) ⟹
    env xa = Some va  ⟹
    seq_step_up
      (App f xa, env)
      (el, envl(
        fp ↦ (VClsr (Abs fp xp el) envl),
        xp ↦ va))
```

```


type_synonym cmmn_set = (ctrl_path * chan * ctrl_path) set

type_synonym trace_pool = ctrl_path ⇀ state

inductive
  leaf ::
    trace_pool ⇒ ctrl_path ⇒ bool where
  intro :
    trpl pi ≠ None ⟹
    (∄ pi' . trpl pi' ≠ None ∧ strict_prefix pi pi') ⟹
    leaf trpl pi
```

```

inductive
  concur_step ::
    trace_pool * cmmn_set ⇒
    trace_pool * cmmn_set ⇒
    bool where
  Seq_Sttep_Down :
    leaf trpl pi ⟹
    trpl pi = Some
```

```
        (Stt (Rslt x) env
          ((Ctn xk ek envk) # k)) ⟹
      env x = Some v ⟹
      concur_step
        (trpl, ys)
        (trpl(pi @ [LRtn xk] ↦
          (Stt ek (envk(xk ↦ v)) k)), ys) |
    Seq_Step :
      leaf trpl pi ⟹
      trpl pi = Some
        (Stt (Let x  b e) env k) ⟹
      seq_step (b, env) v⟹
      concur_step
        (trpl, ys)
        (trpl(pi @ [LNxt x] ↦
          (Stt e (env(x ↦ v)) k), ys) |
    Seq_Step_Up :
      leaf trpl pi ⟹
      trpl pi = Some
        (Stt (Let x b e) env k) ⟹
      seq_step_up (b, env) (e', env') ⟹
      concur_step
        (trpl, ys)
        (trpl(pi @ [LCall x] ↦
          (Stt e' env'
            ((Ctn x e env) # k))), ys) |
    LetMkCh :
      leaf trpl pi ⟹
      trpl pi = Some (Stt (Let x MkChn e) env k) ⟹
      concur_step
        (trpl, ys)
        (trpl(pi @ [LNxt x] ↦
          (Stt e (env(x ↦ (VChn (Chn pi x)))) k)), ys) |
    LetSpwn :
      leaf trpl pi ⟹
      trpl pi = Some
        (Stt (Let x (Spwn ec) e) env k) ⟹
      concur_step
        (trpl, ys)
        (trpl(
          pi @ [LNxt x] ↦
              (St e (env(x ↦ VUnt)) k),
          pi @ [LSpwn x] ↦
              (St ec env []), ys) |
    LetSync :
      leaf trpl pis ⟹
      trpl pis = Some
        (Stt (Let xs (Sync xse) es) envs ks) ⟹
      envs xse = Some
        (VClsr (SendEvt xsc xm) envse) ⟹
      leaf trpl pir  ⟹
      trpl pir = Some
        (Stt (Let xr (Sync xre) er) envr kr) ⟹
      envr xre = Some
        (VClsr (RecvEvt xrc) envre) ⟹
      envse xsc = Some (VChn c) ⟹
      envre xrc = Some (VChn c) ⟹
      envse xm = Some vm ⟹
      concur_step
        (trpl, ys)
        (trpl(
          pis @ [LNxt xs] ↦
            (Stt es (envs(xs ↦ VUnt)) ks),
          pir @ [LNxt xr] ↦
            (Stt er (envr(xr ↦ vm)) kr)),
          ys ∪ {(pis, c, pir)})


```

```
inductive
  star ::
    ('a ⇒ 'a ⇒ bool) ⇒
    'a ⇒ 'a ⇒ bool for r where
  refl : star r x x |
  step : r x y ⟹ star r y z ⟹ star r x z

```

11 Dynamic Communication

```
inductive
  is_send_path ::
    trace_pool ⇒ chan ⇒
    control_path ⇒ bool where
  intro :
    trpl piy = Some
      (Stt (Let xy (Sync xe) en) env k) ⟹
    env xe = Some
      (VClsr (SendEvt xsc xm) enve) ⟹
    enve xsc = Some (VChn c) ⟹
    is_send_path trpl c piy

inductive
  is_recv_path ::
    trace_pool ⇒ chan ⇒
    control_path ⇒ bool where
  intro :
    trpl piy = Some
      (Stt (Let xy (Sync xe) en) env k) ⟹
    env xe = Some
```

```
      (VClsr (RecvEvt xrc) enve) ⟹
    enve xrc = Some (VChn c) ⟹
    is_recv_path trpl c piy


```

```

inductive
  every_two ::
    ('a ⇒ bool) ⇒
    ('a ⇒ 'a ⇒ bool) ⇒ bool where
  intro : (∀ pi1 pi2 .
      p x1 ⟶
      p x2 ⟶
      r x1 x2) ⟹
    every_two p r

inductive
  ordered ::
    'a list ⇒ 'a list ⇒ bool where
  left : prefix pi1 pi2 ⟹ ordered pi1 pi2 |
  right : prefix pi2 pi1 ⟹ ordered pi1 pi2


```

```

inductive one_shot :: trace_pool ⇒ chan ⇒ bool where
  intro :
    every_two
      (is_send_path trpl c) op= ⟹
    one_shot trpl c

inductive fan_out :: trace_pool ⇒ chan ⇒ bool where
  intro :
    every_two
      (is_send_path trpl c) ordered ⟹
    fan_out trpl c

inductive fan_in :: trace_pool ⇒ chan ⇒ bool where
  intro :
    every_two
      (is_recv_path trpl c) ordered ⟹
    fan_in trpl c

inductive one_to_one :: trace_pool ⇒ chan ⇒ bool where
  intro :
    fan_out trpl c ⟹
    fan_in trpl c ⟹
    one_to_one trpl c

```

12 Static Semantics

```

datatype abstract_value =
  AChn var |
  AUnt |
  APrim prim

type_synonym abstract_env = var ⇒ abstract_value set

fun rslt_var :: exp ⇒ var where
  rslt_var (Rslt x) = x |
  rslt_var (Let _ _ e) = (rslt_var e)


```

```


inductive
  static_eval_exp ::
    abstract_env * abstract_env ⇒
    exp ⇒ bool where
  Rslt :
    static_eval_exp (V, C) (Rslt x) |
  let_unt :
    {AUnt} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x Unt e) |
  let_chan :
    {AChn x} ⊆ V x  ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (MkChn) e) |
  let_send_evt :
    {APrim (SendEvt xc xm)} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C)
      (Let x (Prim (SendEvt xc xm)) e) |
  let_recv_evt :
    {APrim (RecvEvt xc)} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C)
      (Let x (Prim (RecvEvt xc)) e) |
  let_pair :
    {APrim (Pair x1 x2)} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Pair x1 x2) e) |
  let_left :
    {APrim (Left xp)} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Left xp) e) |
  let_right :
    {APrim (Right xp)} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Right xp) e) |
  let_abs :
    {APrim (Abs f' x' e')} ⊆ V f' ⟹
    static_eval_exp (V, C) e' ⟹
    {APrim (Abs f' x' e')} ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Abs f' x' e') e) |
  let_spawn :
    {AUnt} ⊆ V x ⟹
    static_eval_exp (V, C) ec ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Spwn ec) e) |
  let_sync  :
    ∀ xsc xm xc .
      (APrim (SendEvt xsc xm)) ∈ V xe ⟶
      AChn xc ∈ V xsc ⟶
      {AUnt} ⊆ V x ∧ V xm ⊆ C xc ⟹
    ∀ xrc xc .
      (APrim (RecvEvt xrc)) ∈ V xe ⟶
      AChn xc ∈ V xrc ⟶
      C xc ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Syync xe) e) |
  let_fst :
    ∀ x1 x2.
      (APrim (Pair x1 x2)) ∈ V xp ⟶
      V x1 ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Fst xp) e) |
  let_snd :
  ∀ x1 x2 .
    (APrim (Pair x1 x2) ∈ V xp ⟶
    V x2 ⊆ V x ⟹
  static_eval_exp (V, C) e ⟹
  static_eval_exp (V, C) (Let x (Snd xp) e) |
let_case :
  ∀ xl' .
    (APrim (Left xl')) ∈ V xs ⟶
      V xl' ⊆ V xl ∧ V (rslt_var el) ⊆ V x ∧
      static_eval_exp (V, C) el ⟹
  ∀ xr' .
    (APrim (Right xr')) ∈ V xs ⟶
      V xr' ⊆ V xr ∧ V (rslt_var er) ⊆ V x ∧
      static_eval_exp (V, C) er ⟹
      static_eval_exp (V, C) e ⟹
    static_eval_exp (V, C) (Let x (Case xs xl el xr er) e)
  |
let_app :
    ∀ f' x' e' .
    (APrim (Abs f' x' e') ∈ V f ⟶
    V xa ⊆ V x' ∧
    V (rslt_var e') ⊆ V x ⟹
  static_eval_exp (V, C) e ⟹
  static_eval_exp (V, C) (Let x (App f xa) e)


```

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

```

```


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


```

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

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

```

inductive
  static_eval_val ::
    abstract_env * abstract_env ⇒ val ⇒ bool and
  static_eval_env ::
    abstract_env * abstract_env ⇒ val_env ⇒ bool where
  Unt :
    static_eval_val (V, C) VUnt |
  Chan :
    static_eval_val (V, C) VChn c |
  SendEvt :
    static_eval_env (V, C) env ⟹
    static_eval_val (V, C) (VClsr (SendEvt _ _) env) |
  RecvEvt :
    static_eval_env (V, C) env ⟹
    static_eval_val (V, C) (VClsr (RecvEvt _) env) |
  Left :
    static_eval_env (V, C) env ⟹
    static_eval_val (V, C) (VClsr (Left _) env) |
  Right :
    static_eval_env (V, C) env ⟹
    static_eval_val (V, C) (VClsr (Right _) env) |
  Abs :
    {(APrim (Abs f x e)} ⊆ V f ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_env (V, C) env ⟹
    static_eval_val (V, C) (VClsr (Abs f x e) env) |
  Pair :
    static_eval_env (V, C) env ⟹
    static_eval_val (V, C) (VClsr (Pair _ _) env) |
  intro :
    ∀ x v .
      env x = Some v ⟶
      {abstract v} ⊆ V x ∧ static_eval_val (V, C) v ⟹
    static_eval_env (V, C) env
```

```

inductive static_eval_stack ::
  abstract_env * abstract_env ⇒
  abstract_value set ⇒ cont list ⇒ bool where
  Empty :
    static_eval_stack (V, C) valset [] |
  Nonempty :
    valset ⊆ V x ⟹
    static_eval_exp (V, C) e ⟹
    static_eval_env (V, C) env ⟹
    static_eval_stack (V, C) (V (rslt_var e)) k ⟹
    static_eval_stack (V, C) valset ((Ctn x e env) # k)


inductive static_eval_state ::
  abstract_env * abstract_env ⇒
  state ⇒ bool where
  intro :
    static_eval_exp (V, C) e ⟹
    static_eval_env (V, C) env ⟹
    static_eval_stack (V, C) (V (rslt_var e)) k ⟹
    static_eval_state (V, C) (Stt e env k)

inductive static_eval_pool ::
  abstract_env * abstract_env ⇒
  trace_pool ⇒ bool where
  intro :
    ∀ pi st .
      trpl pi = Some st ⟶
      static_eval_state (V, C) st ⟹
    static_eval_pool (V, C) trpl


```

```

theorem static_eval_preserved_under_concur_step :
  static_eval_pool (V, C) trpl ⟹
  concur_step (trpl, ys) (trpl', ys') ⟹
  static_stati_eval_pool (V, C) trpl'
proof sketch
qed

theorem static_eval_preserved_under_concur_step_star :
  static_eval_pool (V, C) trpl ⟹
  star concur_step (trpl, ys) (trpl', ys')  ⟹
  static_concur_step (V, C) trpl'
proof sketch
qed

```

```

theorem trace_pool_snapshot_not_static_bound_sound :
  env x = Some v ⟹
  trpl pi = Some (Stt e env k) ⟹
  static_eval_pool (V, C) trpl ⟹
  {abstract v} ⊆ V x
proof sketch
qed

```

```

theorem trace_pool_always_not_static_bound_sound :
  env' x = Some v ⟹
  static_eval_pool (V, C) trpl ⟹
  star concur_step (trpl, ys) (trpl', ys') ⟹
  trpl' pi = Some (Stt e' env' k') ⟹
  {abstract v} ⊆ V x
proof sketch
qed


```

```

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

inductive
  static_reachable_over_prim :: exp ⇒ prim ⇒ bool where
  SendEvt :
    static_reachable_over_prim e0 (SendEvt xC xM) |
  RecvEvt :
    static_reachable_over_prim e0 (RecvEvt xC) |
  Pair :
    static_reachable_over_prim e0 (Pair x1 x2) |
  Left :
    static_reachable_over_prim e0 (Left x) |
  Right :
    static_reachable_over_prim e0 (Right x) |
  Abs :
    static_reachable_left e0 eb ⟹
    static_reachable_over_prim e0 (Abs fp xp eb)

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

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

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

```
lemma static_reachable_trans :
  static_reachable ez ey ⟹
  static_reachable ey ex ⟹
```

```
    static_reachable ez ex
  proof sketch
  qed


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

```

```

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


```

```
    interpretation semantics_sound
    proof sketch
    qed

```

13 Static Communication

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

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

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

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

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

```

```
datatype node_label = NLet var | NRslt var

fun top_node_label :: exp ⇒ node_label where
  top_node_label (Let x b e) = NLet x |
  top_node_label (Rslt y) = NRslt y

type_synonym node_set = node_label set

type_synonym node_map = node_label ⇒ var set

inductive
  static_static_send_node_label ::
    abstract_env ⇒ exp ⇒
    var ⇒ node_label ⇒ bool where
  intro:
    {AChn xC} ⊆ V xSC ⟹
    {APrim (SendEvt xSC xM)} ⊆ V xE ⟹
    static_reachable e (Let x (Sync xE) e') ⟹
    static_static_send_node_label V e xC (NLet x)

inductive
  static_static_recv_node_label ::
    abstract_env ⇒ exp ⇒
    var ⇒ node_label ⇒ bool where
  intro:
    {AChn xC} ⊆ V xRC ⟹
    {APrim (RecvEvt xRC)} ⊆ V xE ⟹
    static_reachable e (Let x (Sync xE) e') ⟹
```

```
      static_static_recv_node_label V e xC (NLet x)


```

14 Static Communication part A

```

datatype edge_label = ENxt | ESpwn | ECall | ERtn

type_synonym flow_label = node_label * edge_label *
  node_label

type_synonym flow_set = flow_label set

type_synonym step_label = node_label * edge_label

type_synonym abstract_path = step_label list

```

```
inductive
  static_traversable ::
    abstract_env ⇒
    (node_label * edge_label * node_label) set ⇒
    exp ⇒ bool  where
  Rslt:
    static_traversable V F (Rslt x) |
  let_Unit:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e  ⟹
    static_traversable V F (Let x Unt e) |
  let_Chan:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x MkChn e) |
  let_SendEvt:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x = (SendEvt xc xm) e) |
  let_RecvEvt:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x = (RecvEvt xc) in e) |
  let_Pair:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Pair x1 x2) e) |
  let_Left:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Lft xp) e) |
  let_Right:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Rht xp) e)" |
  let_Abs:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F eb ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Abs f xp eb) e) |
  let_Spawn:
    {
      (NLet x, ENxt, top_node_label e),
      (NLet x, ESpwn, top_node_label ec)}
      ⊆ F ⟹
    static_traversable V F ec ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Spwn ec)  e) |
  let_Sync:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Sync xSE) e) |
  let_Fst:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Fst xp) e) |
  let_Snd:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Snd xp) e) |
  let_Case:
    {
      (NLet x, ECall, top_node_label el),
      (NLet x, ECall, top_node_label er),
      (NRslt (rslt_var el), ERtn, top_node_label e),
      (NRslt (rslt_var er), ERtn, top_node_label e)}
      ⊆ F ⟹
    static_traversable V F el ⟹
    static_traversable V F er ⟹
    static_traversable V F e ⟹
    static_traversable V F (Let x (Case xs xl el xr er) e)
  |
  let_App:
    ∀ fp xp eb .
      APrim (Abs fp xp eb) ∈ V f ⟶
      {
        (NLet x, ECall, top_node_label eb),
        (NRslt (rslt_var eb), ERtn, top_node_label e)}
        ⊆ F ⟹
      static_traversable V F e ⟹
      static_traversable V F (Let x (App f xa) e)


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


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


inductive
  singular ::
    abstract_path ⇒
    abstract_path ⇒ bool where
equal:
  pi1 = pi2 ⟹
  singular pi1 pi2 |
exclusive:
  /(static_inclusive pi1 pi2) ⟹
  singular pi1 pi2
```

```

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

```

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

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

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

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


```

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

```

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

```

```
  lemma not_static_inclusive_sound: "
    star concur_step
      ([[] ↦ (Stt e (λ _ . None) [])], {})
      (trpl', ys') ⟹
    trpl' pi1 ≠ None ⟹
    trpl' pi2 ≠ None ⟹
    paths_correspond pi1 path1 ⟹
    paths_correspond pi2 path2 ⟹
    static_inclusive path1 path2"


```

```
inductive
  static_traversable_env ::
    abstract_env ⇒ flow_set ⇒ env ⇒ bool and
  static_traversable_val ::
    abstract_env ⇒ flow_set ⇒ val ⇒ bool where
  Intro:
    ∀ x v .
      env x = Some v ⟶
      {rslt_var v} ⊆ V x ∧ static_traversable_val V F v
 ⟹
    static_traversable_env V F env |

  Unit:
    static_traversable_val V F VUnit |
  Chan:
    static_traversable_val V F (VChn c) |
  SendEvt:
    static_traversable_env V F env ⟹
    static_traversable_val V F (VClsr (SendEvt _ _) env) |
  RecvEvt:
    static_traversable_env V F env ⟹
    static_traversable_val V F (VClsr (RecvEvt _) env) |
  Left:
    static_traversable_env V F env ⟹
    static_traversable_val V F (VClsr (Left _) env) |
  Right:
    static_traversable_env V F env ⟹
    static_traversable_val V F (VClsr (Right _) env) |
  Abs:
    static_traversable V F e ⟹
    static_traversable_env V F  env ⟹
    static_traversable_val V F (VClsr (Abs f x e) env) |
Pair:
  static_traversable_env V F env ⟹
  static_traversable_val V F (VClsr (Pair _ _) env)



inductive static_traversable_stack ::
  abstract_env ⇒ flow_set ⇒ cont list ⇒ bool where
  Empty:
    static_traversable_stack V F [] |
  Nonempty:
    static_traversable V F e ⟹
```

```
      static_traversable_env V F env ⟹
      static_traversable_stack V F k ⟹
      static_traversable_stack V F ((Ctn x e env) # k))

  inductive
    static_traversable_pool ::
      abstract_env ⇒ flow_set ⇒
      trace_pool ⇒ bool  where
    Intro:
      ∀ pi e env k .
        E pi = Some (Stt e env k) ⟶
        static_traversable V F e ∧
        static_traversable_env V F env ∧
        static_traversable_stack V F k ⟹
      static_traversable_pool V F E


```

```


  lemma static_traversable_pool_preserved_star: "
    static_traversable_pool V F
      ([[] ↦ (Stt e (λ _ . None) [])]) ⟹
    static_eval (V, C) e ⟹
    trpl' pi = Some (Stt (Let x b en) envk) ⟹
    star concur_step
      ([[] ↦ (Stt e (λ _ . None) [])], {})
      (trpl', ys') ⟹
    isEnd (NLet x) ⟹
    static_traversable_pool V F trpl' "
  proof sketch
  qed

lemma static_traversable_pool_implies_static_traceable: "
  trpl' pi = Some (Stt (Let x b in en) env k) ⟹
  concur_step
    ([[] ↦ (Stt e (λ _ . None) [])], {})
    (trpl', ys') ⟹
  static_eval (V, C) e ⟹
  static_traversable_pool V F trpl' ⟹
  isEnd (NLet x) ⟹
  ∃ path .
    paths_correspond pi path ∧
    static_traceable V F (top_node_label e) isEnd path "
  proof sketch
  qed

```

```

  lemma not_static_traceable_sound: "
```

```
trpl' pi = Some (Stt (Let x b en) env k) ⟹
star concur_step
  ([[] ↦ (Stt e (λ _ . None) [])], {})
  (trpl', ys') ⟹
static_eval (V, C) e ⟹
static_traversable V F e ⟹
isEnd (NLet x) ⟹
∃ path .
  paths_correspond pi path ∧
  static_traceable V F (top_node_label e) isEnd path
proof sketch
qed

```

```


interpretation communication_sound_A
  proof -


```

15 Static Communication part B

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

type_synonym flow_label = (node_label * edge_label *
    node_label)

type_synonym flow_set = flow_label set

type_synonym step_label = (node_label * edge_label)

type_synonym abstract_path = step_label list

```

```

  inductive static_traversable :: abstract_env ⇒ flow_set
    ⇒ (var ⇒ node_label ⇒ bool) ⇒ exp ⇒ bool  where
    result:
        static_traversable V F static_recv_site (Rslt x) |
    let_Unit:
        {(NLet x , ENxt, top_node_label e)} ⊆ F ⟹
        static_traversable V F static_recv_site e ⟹
        static_traversable V F static_recv_site (Let x Unt e
    ) |
    let_Chan:
        {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
        static_traversable V F static_recv_site e ⟹
        static_traversable V F static_recv_site (Let x MkChn e) |
  let_SendEvt:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Prim (SendEvt xc xm)) e)" |
  let_RecvEvt:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Prim (RecvEvt xc)) e) |
  let_Pair:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Prim (Pair x1 x2)) e) |
  let_Left:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Prim (Lft xp)) e) |
  let_Right:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Prim (Rght xp)) e) |
  let_Abs:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    static_traversable V F static_recv_site eb ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Prim (Abs f xp eb)) e) |
  let_Spawn:
    {(NLet x, ENxt, top_node_label e),
      (NLet x, ESpawn, top_node_label ec)} ⊆ F ⟹
    static_traversable V F static_recv_site ec ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Spwn ec) e) |
  let_Sync:
    {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
    (∀ xSC xM xC y .
      {^SendEvt xSC xM} ⊆ V xSE ⟶
      {^Chan xC} ⊆ V xSC ⟶
      static_recv_site xC (NLet y) ⟶
      {(NLet x, ESend xSE, NLet y)} ⊆ F) ⟹
    static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (Sync xSE) e) |
  let_Fst:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Fst xp) e) |
  let_Snd:
      {(NLet x, ENxt, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Snd xp) e) |
  let_Case:
      {
        (NLet x, ECall, top_node_label el),
        (NLet x, ECall, top_node_label er),
        (NRslt (rslt_var el), ERtn x, top_node_label e),
        (NRslt (rslt_var er), ERtn x, top_node_label e)} ⊆ F ⟹
      static_traversable V F static_recv_site el ⟹
      static_traversable V F static_recv_site er ⟹
      static_traversable V F static_recv_site e ⟹
      static_traversable V F static_recv_site (Let x (Case xs xl el xr er) e) |
  let_App:
      (∀ f' xp eb . ^Abs f' xp eb ∈ V f ⟶
        {(NLet x, ECall, top_node_label eb),
          (NRslt (rslt_var eb), ERtn x, top_node_label e)} ⊆ F) ⟹
      static_traversable V F static_recv_site e ⟹
    static_traversable V F static_recv_site (Let x (App f xa) e)


```

```
inductive
  static_built_on_chan :: "abstract_env ⇒ node_map ⇒ var ⇒ var ⇒ bool"
where
  Chan:
      AChn xc ∈ V x ⟹
      static_built_on_chan V Ln xc x |
  SendEvt:
      APrim (SendEvt xsc xm) ∈ V x ⟹
      static_built_on_chan V Ln xc xsc ∨ static_built_on_chan V Ln xc xm ⟹
      static_built_on_chan V Ln xc x |
  RecvEvt:
    APrim (RecvEvt xrc) ∈ V x ⟹
    static_built_on_chan V Ln xc xrc ⟹
    static_built_on_chan V Ln xc x |
  Pair:
      APrim (Pair x1 x2) ∈ V x ⟹
      static_built_on_chan V Ln xc x1 ∨ static_built_on_chan
    V Ln xc x2 ⟹
      static_built_on_chan V Ln xc x |
  Left:
      APrim (Left xa) ∈ V x ⟹
      static_built_on_chan V Ln xc xa ⟹
    static_built_on_chan V Ln xc x |
  Right:
      APrim (Right xa) ∈ V x ⟹
      static_built_on_chan V Ln xc xa ⟹
      static_built_on_chan V Ln xc x |
  Abs:
    APrim (Abs f xp eb) ∈ V x ⟹
    /Set.is_empty (Ln (nodeLabel eb) - {xp}) ⟹
    static_built_on_chan V Ln xc x

```

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

```

```
  inductive static_live_chan ::
    abstract_env ⇒ node_map ⇒ node_map ⇒ var ⇒ exp ⇒ bool
     where
    Result:
      chan_set V Ln xc y = Ln (NRslt y) ⟹
      static_live_chan V Ln Lx xc (Rslt y) |
  Let_Unit:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      Lx (NLet x) = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x Unt e) |
  Let_Chan:
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    (Lx (NLet x) - {x}) = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x MkChn e) |
  Let_SendEvt:
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xsc ∪
      chan_set V Ln xc xm = Ln (NLet x) ⟹
    static_live_chan V Ln Lx xc (Let x (Prim (SendEvt xsc xm
   )) e) |
  Let_RecvEvt:
    static_live_chan V Ln Lx xc e ⟹
    Ln (top_node_label e) = Lx (NLet x) ⟹
    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xrc = Ln (NLet x)
     ⟹
    static_live_chan V Ln Lx xc (Let x (Prim (RecvEvt xrc))
   e) |
  Let_Pair:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪  chan_set V Ln xc x1 ∪
        chan_set V Ln xc x2 = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Pair x1 x2))
    e) |
  Let_Left:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x
   ) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Lft xa)) e)
   |
  Let_Right:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x
   ) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Rght xa)) e)
    |
  Let_Abs:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      static_live_chan V Ln Lx xc eb ⟹
      (Lx (NLet x) - {x}) ∪
        (Ln (top_node_label eb) - {xp}) = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Prim (Abs f xp eb)
   ) e) |
  Let_Spawn:
      static_live_chan V Ln Lx xc e ⟹
      static_live_chan V Ln Lx xc ec ⟹
      Ln (top_node_label e) ∪ Ln (top_node_label ec) = Lx (
   NLet x) ⟹
      (Lx (NLet x) - {x}) = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Spwn ec) e) |
  Let_Sync:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xe = Ln (NLet x
   ) ⟹
      static_live_chan V Ln Lx xc (Let x (Sync xe) e) |
  Let_Fst:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x
    ) ⟹
      static_live_chan V Ln Lx xc (Let x (Fst xa) e) |
  Let_Snd:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xa = Ln (NLet x
    ) ⟹
      static_live_chan V Ln Lx xc (Let x (Snd xa) e) |
  Let_Case:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      static_live_chan V Ln Lx xc el ⟹
      static_live_chan V Ln Lx xc er ⟹
      (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xs ∪
      (Ln (top_node_label el) - {xl}) ∪
      (Ln (top_node_label er) - {xr}) = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (Case xs xl el xr
    er) e) |
  Let_App:
      static_live_chan V Ln Lx xc e ⟹
      Ln (top_node_label e) = Lx (NLet x) ⟹
      (Lx (NLet x) - {x}) ∪
      chan_set V Ln xc f ∪
      chan_set V Ln xc xa = Ln (NLet x) ⟹
      static_live_chan V Ln Lx xc (Let x (App f xa) e)
```

```

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

```

inductive static_live_traversable :: "flow_set ⇒ node_map
  ⇒ node_map ⇒ flow_label ⇒ bool"  where
Next: "
```

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


```

```


inductive static_live_traceable :: "abstract_env ⇒ flow_set
    ⇒ node_map ⇒ node_map ⇒ node_label ⇒ (node_label ⇒
    bool) ⇒ abstract_path ⇒ bool" where
  Empty:
    isEnd start ⟹
    static_live_traceable V F Ln Lx start isEnd [] |
  Edge:
    isEnd end ⟹
    static_live_traversable F Ln Lx (start, edge, end) ⟹
    static_live_traceable V F Ln Lx start isEnd [(start,
    edge)] |
  Step:
    static_live_traceable V F Ln Lx middle isEnd ((middle,
   edge') # path) ⟹
    static_live_traversable F Ln Lx (start, edge, middle) ⟹
    static_live_traceable V F Ln Lx start isEnd ((start,
   edge) # (middle, edge') # path) |
  Pre_Return:
    static_live_traceable V F Ln Lx (NRslt y) isEnd ((NRslt
   y, ERtn x) # post) ⟹
    static_traceable  F (NRslt y) pre ⟹
    /static_balanced (pre @ [(NRslt y, ERtn x)]) ⟹
    /Set.is_empty (Lx (NLet x)) ⟹
    path = pre @ (NRslt y, ERtn x) # post ⟹
    static_live_traceable V F Ln Lx start isEnd path
```

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

```
  inductive singular ::
    abstract_path ⇒ abstract_path ⇒ bool where
    equal:
      pi1 = pi2 ⟹
      singular pi1 pi2 |
    exclusive:
      /(pi1 static_inclusive pi2) ⟹
      singular pi1 pi2

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

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

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

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

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


```

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

```

```


```

```

```

```
interpretation communication_sound_B
proof sketch
qed

```

```

```

```

```

```

```

```


```

```


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

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

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

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