

Formal Theory of Communication Topology in Concurrent ML

Thomas Logan

July 29, 2018

1 Summary

The goal of this master's thesis is to 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 `make` 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 Reppey 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 $!$ 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 $\text{fn } x = \lambda x. x + 1$, $\text{fn } y = \lambda y. y + 2$, $\text{fn } z = \lambda z. z + 3$, respectively. x has the abstract values of $\text{fn } y = \lambda y. y + 2$, $\text{fn } z = \lambda z. z + 3$, 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 in 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 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 state's 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. Additionally, 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, then 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 Informal Objectives

To optimize a Concurrent ML program with specialized implementations of communication, one needs to determine which specialized implementations can be swapped in for which original implementations. If there is at most one thread attempting to send or receive on a channel during a run of a program, then the implementation of the synchronization on that channel can be swapped for a faster specialized version that doesn't even consider the possibility of competition from other threads.

We require that the example specialized implementations of figure ? are sound with respect to the general implementation given particular communication topologies, although we do not formally prove this result. For example, we state the implementation soundness result for one-to-one channels.

Suppose structure `GenChan` is a general implementation of communication, and `OneToOneChan` is a specialized implementation of communication. If channel `c` is one-to-one during the execution of a program, where synchronization in the program is implemented with `GenChan`, then `GenChan` can be replaced with `OneToOneChan` for channel `c`, and the behavior of the modified program is equivalent to that of the original.

The implementation soundness is predicated on knowing the communication topology of a channel. Since programs are not required to terminate, it's not always possible to determine the most precise communication topologies of channels. However, it should be possible to determine an approximation of communication topologies for all programs. We expect that the communication topologies are computable from the relations in figure ?, although we do not formally prove this result.

To use the approximate communication topologies, we must ensure that the static relations of figure ? are sound with respect to the dynamic relations of communication topologies of figure ?. We formally prove this result. Figure ? state the topology classification soundness result for one-to-one channels.

8 Mathematical Artifacts

```
1  type thread_id
2  val spawn : (unit -> unit) -> thread_id
3
4  type 'a chan
5  val channel : unit -> 'a chan
6  val recv : 'a chan -> 'a
7  val send : ('a chan * 'a) -> unit
8
```

```

1
2 signature SERV = sig
3   type serv
4     val make : unit -> serv
5     val call : serv * int -> int
6 end
7
8 structure Serv : SERV = struct
9   datatype serv = S of (int * int chan) chan
10
11   fun make () = let
12     val reqChn = channel ()
13     fun loop state = let
14       val (v, replCh) = recv reqChn in
15       send (replCh, state);
16       loop v end in
17     spawn (fn () => loop 0);
18     S reqChn end
19
20   fun call (server, v) = let
21     val S reqChn = server
22     val replChn = channel () in
23     send (reqCh, (v, replCh));
24     recv replChn end end
25
26
1
2 type 'a event
3 val sync : 'a event -> 'a
4 val recvEvt : 'a chan -> 'a event
5 val sendEvt : 'a chan * 'a -> unit event
6 val choose : 'a event * 'a event -> 'a event
7
8 fun send (ch, v) = sync (sendEvt (ch, v))
9 fun recv v = sync (recvEvt v)
10
11 val thenEvt : 'a event * ('a -> 'b event) -> 'b event
12
13
1 val server = Serv.make ()
2 val _ = spawn (fn () => Serv.call (server, 35))
3 val _ = spawn (fn () =>
4   Serv.call (server, 12);
5   Serv.call (server, 13))
6 val _ = spawn (fn () => Serv.call (server, 81))
7 val _ = spawn (fn () => Serv.call (server, 44))
8

```

```

1  structure Serv : SERV = struct
2      datatype serv = S of (int * int chan) chan
3
4      fun make () = let
5
6          val reqChn = FanIn.channel()
7
8          fun loop state = let
9              val (v, replCh) = FanIn.recv reqChn in
10             OneShot.send (replCh, state);
11             loop v end in
12
13             spawn (fn () => loop 0);
14             S reqChn end
15
16     fun call (server, v) = let
17         val S reqChn = server
18         val replChn = OneShot.channel () in
19         FanIn.send (reqCh, (v, replCh));
20         OneShot.recv replChn end
21
22     end
23

```

```

1  let
2      val w = 4
3      val x = ref 1
4      val y = ref 2
5      val z = (!x + 1) + (!y + 2) + (w - 3)
6      val w = 1 in
7      y := 0;
8      (!y + 2) - (!x + 1) * (w - 3) end
9

```

```

1  let
2      val x = 1
3      val y = 2
4      val z = ref (4 * 73)
5      val x = 4 in
6      z := 1;
7      x * !z end
8

```

```

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

```

```

1
2  datatype 'a list = Nil | Cons 'a ('a list)
3
4  inductive
5    sorted ::
6      ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$ 
7      'a list  $\Rightarrow$  bool where
8    Nil : sorted P Nil |
9    Single : sorted P (Cons x Nil) |
10   Cons :
11     P x y  $\Rightarrow$ 
12     sorted P (Cons y ys)  $\Rightarrow$ 
13     sorted P (Cons x (Cons y ys))
14
15
16 1  datatype nat = Z | S nat
17 2
18 3  inductive
19 4    lte ::
20 5      nat  $\Rightarrow$  nat  $\Rightarrow$  bool where
21 6      Eq : lte n n |
22 7      Lt : lte n1 n2  $\Rightarrow$  lte n1 (S n2)
23 8
24 9  theorem "
25 10    sorted lte
26 11      (Cons (Z) (Cons (S Z)
27 12        (Cons (S Z) (Cons
28 13          (S (S (S Z))) Nil))))"
29 14    apply (rule Cons)
30 15    apply (rule Lt)
31 16    apply (rule Eq)
32 17    apply (rule Cons)
33 18    apply (rule Eq)
34 19    apply (rule Cons)
35 20    apply (rule Lt)
36 21    apply (rule Lt)
37 22    apply (rule Eq)
38 23    apply (rule Single)
39 24    done
40 25
41
42 1  definition True :: bool where
43 2    True  $\equiv$  (( $\lambda x :: \text{bool}. x$ ) = ( $\lambda x. x$ ))
44 3
45 4  definition False :: bool where
46 5    False  $\equiv$  ( $\forall P. P$ )
47 6
48 7
49
50 1  signature CHAN = sig

```



```

2   type 'a chan
3   val channel : unit -> 'a chan
4   val send : 'a chan * 'a -> unit
5   val recv : 'a chan -> 'a
6   end

1
2   structure ManyToManyChan : CHAN = struct
3     type message_queue = 'a option ref queue
4
5     datatype 'a chan_content =
6       Send of (condition * 'a) queue |
7       Recv of (condition * 'a option ref) queue |
8       Inac
9
10    datatype 'a chan =
11      Chn of 'a chan_content ref * mutex_lock
12
13    fun channel () = Chn (ref Inac, mutexLock ())
14
15    fun send (Chn (conRef, lock)) m =
16      acquire lock;
17      (case !conRef of
18        Recv q => let
19          val (recvCond, mopRef) = dequeue q in
20          mopRef := Some m;
21          if (isEmpty q) then conRef := Inac else ();
22          release lock; signal recvCond; () end |
23        Send q => let
24          val sendCond = condition () in
25          enqueue (q, (sendCond, m));
26          release lock; wait sendCond; () end |
27        Inac => let
28          val sendCond = condition () in
29          conRef := Send (queue [(sendCond, m)]);
30          release lock; wait sendCond; () end)
31
32    fun recv (Chn (conRef, lock)) =
33      acquire lock;
34      (case !conRef of
35        Send q => let
36          val (sendCond, m) = dequeue q in
37          if (isEmpty q) then
38            conRef := Inac
39          else
40            ();
41          release lock; signal sendCond; m end |
42        Recv q => let
43          val recvCond = condition ()
44          val mopRef = ref None in

```

```

45         enqueue (q, (recvCond, mopRef));
46         release lock; wait recvCond;
47         valOf (!mopRef) end |
48     Inac => let
49         val recvCond = condition ()
50         val mopRef = ref None in
51         conRef := Recv (queue [(recvCond, mopRef)]);
52         release lock; wait recvCond;
53         valOf (!mopRef) end)
54
55 end
56
57
1
2 structure FanOutChan : CHAN = struct
3
4 datatype 'a chan_content =
5     Send of condition * 'a |
6     Recv of (condition * 'a option ref) queue |
7     Inac
8
9 datatype 'a chan =
10     Chn of 'a chan_content ref * mutex_lock
11
12 fun channel () = Chn (ref Inac, mutexLock ())
13
14 fun send (Chn (conRef, lock)) m = let
15     val sendCond = condition () in
16     case cas (conRef, Inac, Send (sendCond, m)) of
17         Inac => (* conRef already set *)
18             wait sendCond; () |
19         Recv q =>
20             (* the current thread is
21              * the only one that updates from this state *)
22             acquire lock;
23             (let
24                 val (recvCond, mopRef) = dequeue q in
25                 mopRef := Some m;
26                 if (isEmpty q) then conRef := Inac else ();
27                 release lock; signal (recvCond);
28                 () end) |
29         Send _ => raise NeverHappens end
30
31 fun recv (Chn (conRef, lock)) =
32     acquire lock;
33     (case !conRef of
34         Inac => let
35             val recvCond = condition ()
36             val mopRef = ref None in

```

```

37         conRef := Recv (queue [(recvCond, mopRef)]);
38         release lock; wait recvCond;
39         valOf (!mopRef) end |
40     Recv q => let
41         val recvCond = condition ()
42         val mopRef = ref None in
43         enqueue (q, (recvCond, mopRef));
44         release lock; wait recvCond;
45         valOf (!mopRef) end |
46     Send (sendCond, m) =>
47         conRef := Inac;
48         release lock;
49         signal sendCond;
50         m end)
51
52     end
53
54
55 structure FanInChan : CHAN = struct
56
57 datatype 'a chan_content =
58     Send of (condition * 'a) queue |
59     Recv of condition * 'a option ref |
60     Inac
61
62 datatype 'a chan =
63     Chn of 'a chan_content ref * mutex_lock
64
65 fun channel () = Chn (ref Inac, mutexLock ())
66
67 fun send (Chn (conRef, lock)) m =
68     acquire lock;
69     case !conRef of
70     Recv (recvCond, mopRef) =>
71         mopRef := Some m; conRef := Inac;
72         release lock; signal recvCond;
73         () |
74     Send q => let
75         val sendCond = condition () in
76         enqueue (q, (sendCond, m));
77         release lock; wait sendCond;
78         () end |
79     Inac => let
80         val sendCond = condition () in
81         conRef := Send (queue [(sendCond, m)])
82         release lock; wait sendCond; () end
83
84 fun recv (Chn (conRef, lock)) = let
85     val recvCond = condition ()
86     val mopRef = ref None in

```

```

33     case cas (conRef, Inac, Recv (recvCond, mopRef)) of
34         Inac => (* conRef already set *)
35             wait recvCond; valOf (!mopRef) |
36         Send q =>
37             (* the current thread is the only one
38              * that updates the state from this state *)
39             acquire lock;
40             (let
41                 val (sendCond, m) = dequeue q in
42                 if (isEmpty q) then conRef := Inac else ();
43                 release lock; signal sendCond; m end) |
44         Recv _ => raise NeverHappens end end
45
46
1
2 structure OneToOneChan : CHAN = struct
3
4     datatype 'a chan_content =
5         Send of condition * 'a |
6         Recv of condition * 'a option ref |
7         Inac
8
9     datatype 'a chan = Chn of 'a chan_content ref
10
11     fun channel () = Chn (ref Inac)
12
13     fun send (Chn conRef) m = let
14         val sendCond = condition () in
15         case cas (conRef, Inac, Send (sendCond, m)) of
16             Inac =>
17                 (* conRef already set to Send *)
18                 wait sendCond; () |
19             Recv (recvCond, mopRef) =>
20                 (* the current thread is the only one
21                  * that accesses conRef for this state *)
22                 mopRef := Some m; conRef := Inac;
23                 signal recvCond; () |
24             Send _ => raise NeverHappens end end
25
26
27     fun recv (Chn conRef) = let
28         val recvCond = condition ();
29         val mopRef = ref None in
30         case cas (conRef, Inac, Recv (recvCond, mopRef)) of
31             Inac => (* conRef already set to Recv *)
32                 wait recvCond; valOf (!mopRef) |
33             Send (sendCond, m) =>
34                 (* the current thread is the only one
35                  * that accesses conRef for this state *)

```

```

36         conRef := Inac; signal sendCond; m |
37         Recv _ => raise NeverHappens end end
38
39     end
40
41
42     structure OneShotChan : CHAN = struct
43
44     datatype 'a chan_content =
45         Send of condition * 'a |
46         Recv of condition * 'a option ref |
47         Inac
48
49     datatype 'a chan = Chn of 'a chan_content ref * mutex_lock
50
51     fun channel () = Chn (ref Inac, lock ())
52
53     fun send (Chn (conRef, lock)) m = let
54         val sendCond = condition () in
55         case (conRef, Inac, Send (sendCond, m)) of
56             Inac =>
57                 (* conRef already set to Send*)
58                 wait sendCond; () |
59             Recv (recvCond, mopRef) =>
60                 mopRef := Some m; signal recvCond;
61                 () |
62             Send _ => raise NeverHappens end end
63
64     fun recv (Chn (conRef, lock)) = let
65         val recvCond = condition ()
66         val mopRef = ref None in
67         case (conRef, Inac, Recv (recvCond, mopRef)) of
68             Inac =>
69                 (* conRef already set to Recv*)
70                 wait recvCond; valOf (!mopRef) |
71             Send (sendCond, m) =>
72                 acquire lock; signal sendCond;
73                 (* never releases lock;
74                 -* blocks others forever *)
75                 m |
76             Recv _ =>
77                 acquire lock;
78                 (* never able to acquire lock;
79                 -* blocked forever *)
80                 raise NeverHappens end end
81
82     end
83
84 end

```

```

1 structure OneShotToOneChan : CHAN = struct
2
3   datatype 'a chan =
4     Chn of condition * condition * 'a option ref
5
6   fun channel () =
7     Chn (condition (), condition (), ref None)
8
9   fun send (Chn (sendCond, recvCond, mopRef)) m =
10     mopRef := Some m; signal recvCond;
11     wait sendCond; ()
12
13   fun recv (Chn (sendCond, recvCond, mopRef)) =
14     wait recvCond; signal sendCond;
15     valOf (!mopRef)
16
17 end
18

```

9 Syntax

```

1
2   datatype var = Var string
3
4   datatype exp =
5     Let var boundexp exp |
6     Rslt var
7
8   boundexp =
9     Unt |
10    MkChn |
11    Prim prim |
12    Spwn exp |
13    Sync var |
14    Fst var |
15    Snd var |
16    Case var var exp var exp |
17    App var var and
18
19   prim =
20     SendEvt var var |
21     RecvEvt var |
22     Pair var var |
23     Lft var |
24     Rht var |
25     Abs var var ex

```

10 Dynamic Semantics

```

1  datatype ctrl_label =
2      LNxt var | LSpwn var | LCall var | LRtn var
3
4  type_synonym ctrl_path = (ctrl_label list)
5
6  datatype chan = Chn ctrl_path var
7
8  datatype val =
9      VUnt | VChn chan | VClsr prim (var  $\rightarrow$  val)
10
11  datatype ctn = Ctn var exp (var  $\rightarrow$  val)
12
13  datatype state = Stt exp (var  $\rightarrow$  val) (ctn list)
14
15

```

```

1
2  inductive
3      seq_step ::
4          bind * (var  $\rightarrow$  val))  $\Rightarrow$  val  $\Rightarrow$  bool where
5      LetUnt :
6          seq_step (Unt, env) VUnt |
7      LetPrim :
8          seq_step (Prim p, env) (VClsr p env) |
9      LetFst :
10         env xp = Some (VClsr (Pair x1 x2) envp)  $\Rightarrow$ 
11         envp x1 = Some v  $\Rightarrow$ 
12         seq_step (Fst xp, env) v |
13      LetSnd :
14         env xp = Some (VClsr (Pair x1 x2) envp)  $\Rightarrow$ 
15         envp x2 = Some v  $\Rightarrow$ 
16         seq_step (Snd xp, env) v
17
18
19

```

```

1
2
3  inductive
4      seq_step_up ::
5          bind * (var  $\rightarrow$  val))  $\Rightarrow$ 
6          exp * val_env  $\Rightarrow$  bool where
7      LetCaseLft :
8          env xs = Some (VClsr (Lft x1') envl)  $\Rightarrow$ 
9          envl x1' = Some v1  $\Rightarrow$ 
10         seq_step_up
11             (Case xs x1 el xr er, env)
12             (el, env(x1  $\mapsto$  v1)) |
13      LetCaseRht :
14         env xs = Some (VClsr (Rht xr') envr)  $\Rightarrow$ 

```

```

15     envr xr' = Some vr  $\implies$ 
16     seq_step_up
17       (Case xs xl el xr er, env)
18       (er, env(xr  $\mapsto$  vr)) |
19   LetApp :
20     env f = Some (VClsr (Abs fp xp el) envl)  $\implies$ 
21     env xa = Some va  $\implies$ 
22     seq_step_up
23       (App f xa, env)
24       (el, envl(
25         fp  $\mapsto$  (VClsr (Abs fp xp el) envl),
26         xp  $\mapsto$  va))
27
28
1
2
3   type_synonym cmmn_set = (ctrl_path * chan * ctrl_path) set
4
5   type_synonym trace_pool = ctrl_path  $\rightarrow$  state
6
7   inductive
8     leaf ::
9     trace_pool  $\Rightarrow$  ctrl_path  $\Rightarrow$  bool where
10   intro :
11     trpl pi  $\neq$  None  $\implies$ 
12     ( $\nexists$  pi' . trpl pi'  $\neq$  None  $\wedge$  strict_prefix pi pi')  $\implies$ 
13     leaf trpl pi
14
15
1
2   inductive
3     concur_step ::
4     trace_pool * cmmn_set  $\Rightarrow$ 
5     trace_pool * cmmn_set  $\Rightarrow$ 
6     bool where
7   Seq_Ststep_Down :
8     leaf trpl pi  $\implies$ 
9     trpl pi = Some
10     (Stt (Rslt x) env
11      ((Ctn xk ek envk) # k))  $\implies$ 
12     env x = Some v  $\implies$ 
13     concur_step
14     (trpl, ys)
15     (trpl(pi @ [LRtn xk]  $\mapsto$ 
16      (Stt ek (envk(xk  $\mapsto$  v)) k)), ys) |
17   Seq_Step :
18     leaf trpl pi  $\implies$ 
19     trpl pi = Some

```



```

20      (Stt (Let x b e) env k) ==>
21      seq_step (b, env) v ==>
22      concur_step
23      (trpl, ys)
24      (trpl(pi @ [LNxt x] ↦
25      (Stt e (env(x ↦ v)) k), ys) |
26  Seq_Step_Up :
27      leaf trpl pi ==>
28      trpl pi = Some
29      (Stt (Let x b e) env k) ==>
30      seq_step_up (b, env) (e', env') ==>
31      concur_step
32      (trpl, ys)
33      (trpl(pi @ [LCall x] ↦
34      (Stt e' env'
35      ((Ctn x e env) # k))), ys) |
36  LetMkCh :
37      leaf trpl pi ==>
38      trpl pi = Some (Stt (Let x MkChn e) env k) ==>
39      concur_step
40      (trpl, ys)
41      (trpl(pi @ [LNxt x] ↦
42      (Stt e (env(x ↦ (VChn (Chn pi x)))) k))), ys) |
43  LetSpwn :
44      leaf trpl pi ==>
45      trpl pi = Some
46      (Stt (Let x (Spwn ec) e) env k) ==>
47      concur_step
48      (trpl, ys)
49      (trpl(
50      pi @ [LNxt x] ↦
51      (St e (env(x ↦ VUnt)) k),
52      pi @ [LSpwn x] ↦
53      (St ec env []), ys) |
54  LetSync :
55      leaf trpl pis ==>
56      trpl pis = Some
57      (Stt (Let xs (Sync xse) es) envs ks) ==>
58      envs xse = Some
59      (VClsr (SendEvt xsc xm) envse) ==>
60      leaf trpl pir ==>
61      trpl pir = Some
62      (Stt (Let xr (Sync xre) er) envr kr) ==>
63      envr xre = Some
64      (VClsr (RecvEvt xrc) envre) ==>
65      envse xsc = Some (VChn c) ==>
66      envre xrc = Some (VChn c) ==>
67      envse xm = Some vm ==>
68      concur_step
69      (trpl, ys)

```

```

70      (trpl(
71        pis @ [LNxt xs]  $\mapsto$ 
72        (Stt es (envs(xs  $\mapsto$  VUnt)) ks),
73        pir @ [LNxt xr]  $\mapsto$ 
74        (Stt er (envr(xr  $\mapsto$  vm)) kr)),
75        ys  $\cup$  {(pis, c, pir)})
76
77
1  inductive
2    star ::
3      ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$ 
4      'a  $\Rightarrow$  'a  $\Rightarrow$  bool for r where
5    refl : star r x x |
6    step : r x y  $\Rightarrow$  star r y z  $\Rightarrow$  star r x z
7

```

11 Dynamic Communication

```

1  inductive
2    is_send_path ::
3      trace_pool  $\Rightarrow$  chan  $\Rightarrow$ 
4      control_path  $\Rightarrow$  bool where
5    intro :
6      trpl piy = Some
7      (Stt (Let xy (Sync xe) en) env k)  $\Rightarrow$ 
8      env xe = Some
9      (VClsr (SendEvt xsc xm) enve)  $\Rightarrow$ 
10     enve xsc = Some (VChn c)  $\Rightarrow$ 
11     is_send_path trpl c piy
12
13  inductive
14    is_recv_path ::
15      trace_pool  $\Rightarrow$  chan  $\Rightarrow$ 
16      control_path  $\Rightarrow$  bool where
17    intro :
18      trpl piy = Some
19      (Stt (Let xy (Sync xe) en) env k)  $\Rightarrow$ 
20      env xe = Some
21      (VClsr (RecvEvt xrc) enve)  $\Rightarrow$ 
22      enve xrc = Some (VChn c)  $\Rightarrow$ 
23      is_recv_path trpl c piy
24
25
1
2  inductive
3    every_two ::
4      ('a  $\Rightarrow$  bool)  $\Rightarrow$ 

```

```

5      ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  bool where
6      intro : ( $\forall$  pi1 pi2 .
7          p x1  $\rightarrow$ 
8          p x2  $\rightarrow$ 
9          r x1 x2)  $\Rightarrow$ 
10         every_two p r
11
12     inductive
13         ordered ::
14             'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool where
15         left : prefix pi1 pi2  $\Rightarrow$  ordered pi1 pi2 |
16         right : prefix pi2 pi1  $\Rightarrow$  ordered pi1 pi2
17
18
19
20     inductive one_shot :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
21         intro :
22             every_two
23             (is_send_path trpl c) op=  $\Rightarrow$ 
24             one_shot trpl c
25
26     inductive fan_out :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
27         intro :
28             every_two
29             (is_send_path trpl c) ordered  $\Rightarrow$ 
30             fan_out trpl c
31
32     inductive fan_in :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
33         intro :
34             every_two
35             (is_recv_path trpl c) ordered  $\Rightarrow$ 
36             fan_in trpl c
37
38     inductive one_to_one :: trace_pool  $\Rightarrow$  chan  $\Rightarrow$  bool where
39         intro :
40             fan_out trpl c  $\Rightarrow$ 
41             fan_in trpl c  $\Rightarrow$ 
42             one_to_one trpl c
43
44
45
46

```

12 Static Semantics

```

1
2     datatype abstract_value =
3         AChn var |
4         AUnt |
5         APrim prim

```

```

6
7 type_synonym abstract_env = var  $\Rightarrow$  abstract_value set
8
9 fun rslt_var :: exp  $\Rightarrow$  var where
10   rslt_var (Rslt x) = x |
11   rslt_var (Let _ _ e) = (rslt_var e)
12
13
1
2
3 inductive
4   static_eval_exp ::
5     abstract_env * abstract_env  $\Rightarrow$ 
6     exp  $\Rightarrow$  bool where
7   Rslt :
8     static_eval_exp (V, C) (Rslt x) |
9   let_unt :
10    {AUnt}  $\subseteq$  V x  $\Longrightarrow$ 
11    static_eval_exp (V, C) e  $\Longrightarrow$ 
12    static_eval_exp (V, C) (Let x Unt e) |
13   let_chan :
14    {AChn x}  $\subseteq$  V x  $\Longrightarrow$ 
15    static_eval_exp (V, C) e  $\Longrightarrow$ 
16    static_eval_exp (V, C) (Let x (MkChn) e) |
17   let_send_evt :
18    {APrim (SendEvt xc xm)}  $\subseteq$  V x  $\Longrightarrow$ 
19    static_eval_exp (V, C) e  $\Longrightarrow$ 
20    static_eval_exp (V, C)
21      (Let x (Prim (SendEvt xc xm)) e) |
22   let_recv_evt :
23    {APrim (RecvEvt xc)}  $\subseteq$  V x  $\Longrightarrow$ 
24    static_eval_exp (V, C) e  $\Longrightarrow$ 
25    static_eval_exp (V, C)
26      (Let x (Prim (RecvEvt xc)) e) |
27   let_pair :
28    {APrim (Pair x1 x2)}  $\subseteq$  V x  $\Longrightarrow$ 
29    static_eval_exp (V, C) e  $\Longrightarrow$ 
30    static_eval_exp (V, C) (Let x (Pair x1 x2) e) |
31   let_left :
32    {APrim (Left xp)}  $\subseteq$  V x  $\Longrightarrow$ 
33    static_eval_exp (V, C) e  $\Longrightarrow$ 
34    static_eval_exp (V, C) (Let x (Left xp) e) |
35   let_right :
36    {APrim (Right xp)}  $\subseteq$  V x  $\Longrightarrow$ 
37    static_eval_exp (V, C) e  $\Longrightarrow$ 
38    static_eval_exp (V, C) (Let x (Right xp) e) |
39   let_abs :
40    {APrim (Abs f' x' e')}  $\subseteq$  V f'  $\Longrightarrow$ 
41    static_eval_exp (V, C) e'  $\Longrightarrow$ 

```

```

42     {APrim (Abs f' x' e')} ⊆ V x ⇒
43     static_eval_exp (V, C) e ⇒
44     static_eval_exp (V, C) (Let x (Abs f' x' e') e) |
45   let_spawn :
46     {AUnt} ⊆ V x ⇒
47     static_eval_exp (V, C) ec ⇒
48     static_eval_exp (V, C) e ⇒
49     static_eval_exp (V, C) (Let x (Spwn ec) e) |
50   let_sync :
51     ∀ xsc xm xc .
52       (APrim (SendEvt xsc xm)) ∈ V xe →
53       AChn xc ∈ V xsc →
54       {AUnt} ⊆ V x ∧ V xm ⊆ C xc ⇒
55     ∀ xrc xc .
56       (APrim (RecvEvt xrc)) ∈ V xe →
57       AChn xc ∈ V xrc →
58       C xc ⊆ V x ⇒
59       static_eval_exp (V, C) e ⇒
60       static_eval_exp (V, C) (Let x (Syync xe) e) |
61   let_fst :
62     ∀ x1 x2 .
63       (APrim (Pair x1 x2)) ∈ V xp →
64       V x1 ⊆ V x ⇒
65       static_eval_exp (V, C) e ⇒
66       static_eval_exp (V, C) (Let x (Fst xp) e) |
67   let_snd :
68     ∀ x1 x2 .
69       (APrim (Pair x1 x2)) ∈ V xp →
70       V x2 ⊆ V x ⇒
71       static_eval_exp (V, C) e ⇒
72       static_eval_exp (V, C) (Let x (Snd xp) e) |
73   let_case :
74     ∀ x1' .
75       (APrim (Left x1')) ∈ V xs →
76       V x1' ⊆ V x1 ∧ V (rslt_var el) ⊆ V x ∧
77       static_eval_exp (V, C) el ⇒
78     ∀ xr' .
79       (APrim (Right xr')) ∈ V xs →
80       V xr' ⊆ V xr ∧ V (rslt_var er) ⊆ V x ∧
81       static_eval_exp (V, C) er ⇒
82       static_eval_exp (V, C) e ⇒
83       static_eval_exp (V, C) (Let x (Case xs x1 el xr er) e)
84   |
85   let_app :
86     ∀ f' x' e' .
87       (APrim (Abs f' x' e')) ∈ V f →
88       V xa ⊆ V x' ∧
89       V (rslt_var e') ⊆ V x ⇒
90       static_eval_exp (V, C) e ⇒
91       static_eval_exp (V, C) (Let x (App f xa) e)

```

91
92

```

1  inductive static_reachable :: exp ⇒ exp ⇒ bool where
2    Refl :
3      static_reachable e e |
4    let_Spawn_Child
5      static_reachable ec e ⇒
6      static_reachable (Let x (Spwn ec) en) e |
7    let_Case_Left :
8      static_reachable el e ⇒
9      static_reachable (Let x (case xs xl el xr er) en) e |
10   let_Case_Right :
11     static_reachable er e ⇒
12     static_reachable (Let x (case xs xl el xr er) en) e |
13   let_Abs_Body : "
14     static_reachable eb e ⇒
15     static_reachable (Let x (Abs f xp eb) en) e |
16   Let :
17     static_reachable en e ⇒
18     static_reachable (Let x b en) e
19
20
21 1
22 2
23 3 fun abstract :: val ⇒ abstract_value where
24 4   abstract VUnt = AUnt |
25 5   abstract VChn (Chn pi x) = AChn x |
26 6   abstract VClsr p env = APrim p
27 7
28 8
29
30 1 locale semantics_sound =
31 2   assumes
32 3     exp_always_not_static_bound_sound :
33 4       env' x = Some v ⇒
34 5       (V, C) static_eval e ⇒
35 6       ([[ ] ↦ (Stt e (λ _ . None) [])], H) star concur_step (
36   trpl', H') ⇒
37 7       trpl' pi = Some (Stt e' env' k') ⇒
38 8       {|v|} ⊆ V x and
39 9
40 10  exp_always_exp_not_static_reachable_sound:
41 11  ([[ ] ↦ (Stt e0 (λ _ . None) [])], {}) star concur_step
42 12  (trpl', H') ⇒
43 13  trpl' pi' = Some (Stt e' env' k') ⇒
44 14  static_reachable e0 e'

```

```

1
2 inductive
3   static_eval_val ::
4     abstract_env * abstract_env  $\Rightarrow$  val  $\Rightarrow$  bool and
5   static_eval_env ::
6     abstract_env * abstract_env  $\Rightarrow$  val_env  $\Rightarrow$  bool where
7   Unt :
8     static_eval_val (V, C) VUnt |
9   Chan :
10    static_eval_val (V, C) VChn c |
11  SendEvt :
12    static_eval_env (V, C) env  $\Rightarrow$ 
13    static_eval_val (V, C) (VClsr (SendEvt _ _) env) |
14  RecvEvt :
15    static_eval_env (V, C) env  $\Rightarrow$ 
16    static_eval_val (V, C) (VClsr (RecvEvt _) env) |
17  Left :
18    static_eval_env (V, C) env  $\Rightarrow$ 
19    static_eval_val (V, C) (VClsr (Left _) env) |
20  Right :
21    static_eval_env (V, C) env  $\Rightarrow$ 
22    static_eval_val (V, C) (VClsr (Right _) env) |
23  Abs :
24    {(APrim (Abs f x e))}  $\subseteq$  V f  $\Rightarrow$ 
25    static_eval_exp (V, C) e  $\Rightarrow$ 
26    static_eval_env (V, C) env  $\Rightarrow$ 
27    static_eval_val (V, C) (VClsr (Abs f x e) env) |
28  Pair :
29    static_eval_env (V, C) env  $\Rightarrow$ 
30    static_eval_val (V, C) (VClsr (Pair _ _) env) |
31  intro :
32     $\forall$  x v .
33    env x = Some v  $\Rightarrow$ 
34    {abstract v}  $\subseteq$  V x  $\wedge$  static_eval_val (V, C) v  $\Rightarrow$ 
35    static_eval_env (V, C) env
36
37

```

```

1
2 inductive static_eval_stack ::
3   abstract_env * abstract_env  $\Rightarrow$ 
4   abstract_value set  $\Rightarrow$  cont list  $\Rightarrow$  bool where
5   Empty :
6     static_eval_stack (V, C) valset [] |
7   Nonempty :
8     valset  $\subseteq$  V x  $\Rightarrow$ 
9     static_eval_exp (V, C) e  $\Rightarrow$ 
10    static_eval_env (V, C) env  $\Rightarrow$ 
11    static_eval_stack (V, C) (V (rslt_var e)) k  $\Rightarrow$ 
12    static_eval_stack (V, C) valset ((Ctn x e env) # k)

```

```

13
14
15 inductive static_eval_state ::
16   abstract_env * abstract_env  $\Rightarrow$ 
17   state  $\Rightarrow$  bool where
18   intro :
19     static_eval_exp (V, C) e  $\Rightarrow$ 
20     static_eval_env (V, C) env  $\Rightarrow$ 
21     static_eval_stack (V, C) (V (rslt_var e)) k  $\Rightarrow$ 
22     static_eval_state (V, C) (Stt e env k)
23
24 inductive static_eval_pool ::
25   abstract_env * abstract_env  $\Rightarrow$ 
26   trace_pool  $\Rightarrow$  bool where
27   intro :
28      $\forall$  pi st .
29     trpl pi = Some st  $\rightarrow$ 
30     static_eval_state (V, C) st  $\Rightarrow$ 
31     static_eval_pool (V, C) trpl
32
33
1
2 theorem static_eval_preserved_under_concur_step :
3   static_eval_pool (V, C) trpl  $\Rightarrow$ 
4   concur_step (trpl, ys) (trpl', ys')  $\Rightarrow$ 
5   static_eval_pool (V, C) trpl'
6 proof sketch
7 qed
8
9 theorem static_eval_preserved_under_concur_step_star :
10  static_eval_pool (V, C) trpl  $\Rightarrow$ 
11  star concur_step (trpl, ys) (trpl', ys')  $\Rightarrow$ 
12  static_eval_pool (V, C) trpl'
13 proof sketch
14 qed
15
1
2 theorem trace_pool_snapshot_not_static_bound_sound :
3   env x = Some v  $\Rightarrow$ 
4   trpl pi = Some (Stt e env k)  $\Rightarrow$ 
5   static_eval_pool (V, C) trpl  $\Rightarrow$ 
6   {abstract v}  $\subseteq$  V x
7 proof sketch
8 qed
9
1
2 theorem trace_pool_always_not_static_bound_sound :

```



```

3   env' x = Some v  $\implies$ 
4   static_eval_pool (V, C) trpl  $\implies$ 
5   star_concur_step (trpl, ys) (trpl', ys')  $\implies$ 
6   trpl' pi = Some (Stt e' env' k')  $\implies$ 
7   {abstract v}  $\subseteq$  V x
8 proof sketch
9 qed
10
11
12
13
14
15
16
17
18
19
20
21
22 inductive
23   static_reachable_left :: exp  $\Rightarrow$  exp  $\Rightarrow$  bool where
24   Refl :
25     static_reachable_left e0 e0 |
26   let_Spawn_Child :
27     static_reachable_left e0 (Let x (Spwn ec) en)  $\implies$ 
28     static_reachable_left e0 ec |
29   let_Case_Left :
30     static_reachable_left e0 (Let x (case xs x1 e1 xr er)
31 en)  $\implies$ 
32     static_reachable_left e0 e1 |
33   let_Case_Right :
34     static_reachable_left e0 (Let x (case xs x1 e1 xr er)
35 en)  $\implies$ 
36     static_reachable_left e0 er |
37   let_Abs_Body :
38     static_reachable_left e0 (Let x (Abs f xp eb) en)  $\implies$ 
39     static_reachable_left e0 eb |
40   Let :
41     static_reachable_left e0 (Let x b en)  $\implies$ 
42     static_reachable_left e0 en
43
44 inductive
45   static_reachable_over_prim :: exp  $\Rightarrow$  prim  $\Rightarrow$  bool where
46   SendEvt :
47     static_reachable_over_prim e0 (SendEvt xC xM) |
48   RecvEvt :
49     static_reachable_over_prim e0 (RecvEvt xC) |
50   Pair :
51     static_reachable_over_prim e0 (Pair x1 x2) |
52   Left :
53     static_reachable_over_prim e0 (Left x) |
54   Right :
55     static_reachable_over_prim e0 (Right x) |
56   Abs :
57     static_reachable_left e0 eb  $\implies$ 
58     static_reachable_over_prim e0 (Abs fp xp eb)
59
60 inductive

```

```

39   static_reachable_over_env :: exp ⇒ env ⇒ bool and
40   static_reachable_over_val :: exp ⇒ val ⇒ bool where
41   VUnt :
42     static_reachable_over_val e0 VUnt |
43   VChn :
44     static_reachable_over_val e0 (VChn c) |
45   VClsr : "
46     static_reachable_over_prim e0 p ⇒
47     static_reachable_over_env e0 env' ⇒
48     static_reachable_over_val e0 (VClsr p env') |
49   intro : "
50     ∀ x v .
51     env x = Some v →
52     static_reachable_over_val e0 v ⇒
53     static_reachable_over_env e0 env
54
55   inductive
56     static_reachable_over_stack ::
57     exp ⇒ cont list ⇒ bool where
58     Empty :
59       static_reachable_over_stack e0 [] |
60     Nonempty :
61       static_reachable_left e0 ek ⇒
62       static_reachable_over_env e0 envk ⇒
63       static_reachable_over_stack e0 k ⇒
64       static_reachable_over_stack e0 ((Ctn xk ek envk) # k)
65
66   inductive
67     static_reachable_over_state ::
68     exp ⇒ state ⇒ bool where
69     intro :
70       static_reachable_left e0 e ⇒
71       static_reachable_over_env e0 env ⇒
72       static_reachable_over_stack e0 k ⇒
73       static_reachable_over_state e0 (Ctn e env k)
74
75
76   lemma static_reachable_trans :
77     static_reachable ez ey ⇒
78     static_reachable ey ex ⇒
79     static_reachable ez ex
80   proof sketch
81   qed
82
83   lemma static_reachable_over_state_preserved :
84     concur_step (trpl, ys) (trpl', ys') ⇒
85     ∀ pi st.
86     trpl pi = Some st →

```

```

13     static_reachable_over_state e0 st  $\Rightarrow$ 
14     trpl' pi' = Some st'  $\Rightarrow$ 
15     static_reachable_over_state e0 st'
16 proof sketch
17 qed
18

1
2 lemma state_always_exp_not_static_reachable_sound :
3   star concur_step (trpl0, ys0) (trpl', ys')  $\Rightarrow$ 
4   trpl0 = [[]  $\mapsto$  (Stt e0 ( $\lambda$  _ . None) [])]  $\Rightarrow$ 
5   trpl' pi' = Some st'  $\Rightarrow$ 
6   static_reachable_over_state e0 st'
7 proof sketch
8 qed
9
10

1   interpretation semantics_sound
2   proof sketch
3   qed
4

```

13 Static Communication

```

1 locale communication_sound =
2   fixes
3     static_one_shot :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$  bool and
4     static_fan_out :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$  bool and
5     static_fan_in :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$  bool and
6     static_one_to_one :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$  bool
7
8   assumes
9     static_one_shot_sound:
10       static_one_shot V e xC  $\Rightarrow$ 
11       static_eval (V, C) e  $\Rightarrow$ 
12       star concur_step ([[]  $\mapsto$  (Stt e ( $\lambda$  _ . None) [])], { })
13       (trpl', H')  $\Rightarrow$ 
14       one_shot trpl' (Ch pi xC) and
15     static_fan_out_sound:
16       static_fan_out V e xC  $\Rightarrow$ 
17       (V, C) static_eval e  $\Rightarrow$ 
18       star concur_step ([[]  $\mapsto$  (Stt e ( $\lambda$  _ . None) [])], { })
19       (trpl', H')  $\Rightarrow$ 
20       fan_out trpl' (Ch pi xC) and
21     static_fan_in_sound:
22       static_fan_in V e xC  $\Rightarrow$ 

```

```

23     (V, C) static_eval e ==>
24     star concur_step ([[]] => (Stt e (λ _ . None) [])), {})
(trpl', H') ==>
25     fan_in trpl' (Ch pi xC) and
26
27     static_one_to_one_sound: "
28     static_one_to_one V e xC ==>
29     (V, C) static_eval e ==>
30     star concur_step ([[]] => (Stt e (λ _ . None) [])), {})
(trpl', H') ==>
31     one_to_one trpl' (Ch pi xC)"
32

1  datatype node_label = NLet var | NRslt var
2
3  fun top_node_label :: exp => node_label where
4      top_node_label (Let x b e) = NLet x |
5      top_node_label (Rslt y) = NRslt y
6
7  type_synonym node_set = node_label set
8
9  type_synonym node_map = node_label => var set
10
11 inductive
12     static_static_send_node_label ::
13     abstract_env => exp =>
14     var => node_label => bool where
15     intro:
16         {AChn xC} ⊆ V xSC ==>
17         {APrim (SendEvt xSC xM)} ⊆ V xE ==>
18         static_reachable e (Let x (Sync xE) e') ==>
19         static_static_send_node_label V e xC (NLet x)
20
21 inductive
22     static_static_recv_node_label ::
23     abstract_env => exp =>
24     var => node_label => bool where
25     intro:
26         {AChn xC} ⊆ V xRC ==>
27         {APrim (RecvEvt xRC)} ⊆ V xE ==>
28         static_reachable e (Let x (Sync xE) e') ==>
29         static_static_recv_node_label V e xC (NLet x)
30
31

```

14 Static Communication part A

```

1
2  datatype edge_label = ENxt | ESpwn | ECall | ERtn

```

```

3
4   type_synonym flow_label = node_label * edge_label *
      node_label
5
6   type_synonym flow_set = flow_label set
7
8   type_synonym step_label = node_label * edge_label
9
10  type_synonym abstract_path = step_label list
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39

```

```

1   inductive
2     static_traversable ::
3       abstract_env  $\Rightarrow$ 
4       (node_label * edge_label * node_label) set  $\Rightarrow$ 
5       exp  $\Rightarrow$  bool   where
6   Rslt:
7     static_traversable V F (Rslt x) |
8   let_Unit:
9     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
10    static_traversable V F e  $\Rightarrow$ 
11    static_traversable V F (Let x Unt e) |
12   let_Chan:
13     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
14     static_traversable V F e  $\Rightarrow$ 
15     static_traversable V F (Let x MkChn e) |
16   let_SendEvt:
17     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
18     static_traversable V F e  $\Rightarrow$ 
19     static_traversable V F (Let x = (SendEvt xc xm) e) |
20   let_RecvEvt:
21     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
22     static_traversable V F e  $\Rightarrow$ 
23     static_traversable V F (Let x = (RecvEvt xc) in e) |
24   let_Pair:
25     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
26     static_traversable V F e  $\Rightarrow$ 
27     static_traversable V F (Let x (Pair x1 x2) e) |
28   let_Left:
29     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
30     static_traversable V F e  $\Rightarrow$ 
31     static_traversable V F (Let x (Lft xp) e) |
32   let_Right:
33     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
34     static_traversable V F e  $\Rightarrow$ 
35     static_traversable V F (Let x (Rht xp) e)" |
36   let_Abs:
37     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\Rightarrow$ 
38     static_traversable V F eb  $\Rightarrow$ 
39     static_traversable V F e  $\Rightarrow$ 

```

```

40     static_traversable V F (Let x (Abs f xp eb) e) |
41 let_Spawn:
42   {
43     (NLet x, ENxt, top_node_label e),
44     (NLet x, ESawn, top_node_label ec)}
45    $\subseteq F \implies$ 
46   static_traversable V F ec  $\implies$ 
47   static_traversable V F e  $\implies$ 
48   static_traversable V F (Let x (Spwn ec) e) |
49 let_Sync:
50   {(NLet x, ENxt, top_node_label e)}  $\subseteq F \implies$ 
51   static_traversable V F e  $\implies$ 
52   static_traversable V F (Let x (Sync xSE) e) |
53 let_Fst:
54   {(NLet x, ENxt, top_node_label e)}  $\subseteq F \implies$ 
55   static_traversable V F e  $\implies$ 
56   static_traversable V F (Let x (Fst xp) e) |
57 let_Snd:
58   {(NLet x, ENxt, top_node_label e)}  $\subseteq F \implies$ 
59   static_traversable V F e  $\implies$ 
60   static_traversable V F (Let x (Snd xp) e) |
61 let_Case:
62   {
63     (NLet x, ECall, top_node_label el),
64     (NLet x, ECall, top_node_label er),
65     (NRslt (rslt_var el), ERtn, top_node_label e),
66     (NRslt (rslt_var er), ERtn, top_node_label e)}
67    $\subseteq F \implies$ 
68   static_traversable V F el  $\implies$ 
69   static_traversable V F er  $\implies$ 
70   static_traversable V F e  $\implies$ 
71   static_traversable V F (Let x (Case xs xl el xr er) e)
72 |
73 let_App:
74    $\forall$  fp xp eb .
75   APrim (Abs fp xp eb)  $\in V f \implies$ 
76   {
77     (NLet x, ECall, top_node_label eb),
78     (NRslt (rslt_var eb), ERtn, top_node_label e)}
79    $\subseteq F \implies$ 
80   static_traversable V F e  $\implies$ 
81   static_traversable V F (Let x (App f xa) e)
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

1 inductive static_traceable ::
2   abstract_env  $\Rightarrow$ 
3   flow_set  $\Rightarrow$  node_label  $\Rightarrow$ 
4   (node_label  $\Rightarrow$  bool)  $\Rightarrow$ 
5   abstract_path  $\Rightarrow$  bool where

```

```

6   Empty:
7     isEnd start  $\Rightarrow$ 
8     static_traceable V F start isEnd [] |
9   Edge:
10    isEnd end  $\Rightarrow$ 
11    {(start, edge, end)}  $\subseteq$  F  $\Rightarrow$ 
12    static_traceable V F start isEnd [(start, edge)] |
13  Step:
14    static_traceable V F middle isEnd ((middle, edge') #
    path)  $\Rightarrow$ 
15    {(start, edge, middle)}  $\subseteq$  F  $\Rightarrow$ 
16    static_traceable V F start isEnd ((start, edge) # (
    middle, edge') # path)
17

1   inductive static_inclusive ::
2     abstract_path  $\Rightarrow$  abstract_path  $\Rightarrow$  bool where
3   Prefix1:
4     prefix pi1 pi2  $\Rightarrow$ 
5     static_inclusive pi1 pi2 |
6   Prefix2:
7     prefix pi2 pi1  $\Rightarrow$ 
8     static_inclusive pi1 pi2 |
9   Spawn1:
10    static_inclusive (pi @ (Nlet x, ESpwn) # pi1) (pi @ (
    Nlet x, ENxt) # pi2) |
11  Spawn2:
12    static_inclusive (pi @ (Nlet x, ENxt) # pi1) (pi @ (Nlet
    x, ESpwn) # pi2)
13

1   inductive
2     singular ::
3       abstract_path  $\Rightarrow$ 
4       abstract_path  $\Rightarrow$  bool where
5   equal:
6     pi1 = pi2  $\Rightarrow$ 
7     singular pi1 pi2 |
8   exclusive:
9     /(static_inclusive pi1 pi2)  $\Rightarrow$ 
10    singular pi1 pi2
11
12  inductive
13    noncompetitive ::
14      abstract_path  $\Rightarrow$  abstract_path  $\Rightarrow$  bool where
15    ordered:
16      ordered pi1 pi2  $\Rightarrow$ 
17      noncompetitive pi1 pi2 |
18    exclusive:
19      /(not_inclusive pi1 pi2)  $\Rightarrow$ 

```

```

20     noncompetitive pi1 pi2
21
1   inductive
2     static_one_shot ::
3       abstract_env  $\Rightarrow$  exp  $\Rightarrow$ 
4       var  $\Rightarrow$  bool where
5     Sync:
6       every_two
7         (static_traceable V F (top_node_label e)
8           (static_static_send_node_label V e xC))
9       singular  $\Rightarrow$ 
10      static_traversable V F e  $\Rightarrow$ 
11      static_one_shot V e xC
12
13  inductive
14    static_one_to_one ::
15      abstract_env  $\Rightarrow$  exp  $\Rightarrow$ 
16      var  $\Rightarrow$  bool where
17    Sync:
18      every_two
19        (static_traceable V F
20          (top_node_label e) (
21            static_static_send_node_label V e xC))
22      noncompetitive  $\Rightarrow$ 
23      every_two
24        (static_traceable V F
25          (top_node_label e) (
26            static_static_recv_node_label V e xC))
27      noncompetitive  $\Rightarrow$ 
28      static_traversable V F e  $\Rightarrow$ 
29      static_one_to_one V e xC
30
31  inductive
32    static_fan_out ::
33      abstract_env  $\Rightarrow$  exp  $\Rightarrow$ 
34      var  $\Rightarrow$  bool where
35    Sync:
36      every_two
37        (static_traceable V F
38          (top_node_label e) (static_static_send_node_label
39            V e xC))
40      noncompetitive  $\Rightarrow$ 
41      static_traversable V F e  $\Rightarrow$ 
42      static_fan_out V e xC
43
44  inductive
45    static_fan_in ::
46      abstract_env  $\Rightarrow$  exp  $\Rightarrow$ 
47      var  $\Rightarrow$  bool where

```



```

45   Sync:
46     every_two (static_traceable V F (top_node_label e) (
47       static_static_recv_node_label V e xC)) noncompetitive  $\Rightarrow$ 
48       static_traversable V F e  $\Rightarrow$ 
49       static_fan_in V e xC
50
1  locale communication_sound_A =
2    Static_Communication.communication_sound static_one_shot
3      static_fan_out static_fan_in static_one_to_one
4
1  inductive paths_correspond ::
2    control_path  $\Rightarrow$  abstract_path  $\Rightarrow$  bool where
3    Empty:
4      paths_correspond [] [] |
5    Next:
6      paths_correspond pi path  $\Rightarrow$ 
7      paths_correspond (pi @ [LNext x]) (path @ [(NLet x,
8        ENxt)]) |
9    Spawn:
10     paths_correspond pi path  $\Rightarrow$ 
11     paths_correspond (pi @ [LSpawn x]) (path @ [(NLet x,
12       ESpwn)]) |
13    Call:
14     paths_correspond pi path  $\Rightarrow$ 
15     paths_correspond (pi @ [LCall x]) (path @ [(NLet x,
16       ECall)]) |
17    Rtn:
18     paths_correspond pi path  $\Rightarrow$ 
19     paths_correspond (pi @ [LRtn x]) (path @ [(NRslt x,
20       ERtn)])
21
1  lemma not_static_inclusive_sound: "
2    star concur_step
3    ([[]  $\mapsto$  (Stt e ( $\lambda$  _ . None) [])], { })
4    (trpl', ys')  $\Rightarrow$ 
5    trpl' pi1  $\neq$  None  $\Rightarrow$ 
6    trpl' pi2  $\neq$  None  $\Rightarrow$ 
7    paths_correspond pi1 path1  $\Rightarrow$ 
8    paths_correspond pi2 path2  $\Rightarrow$ 
9    static_inclusive path1 path2"
10
1  inductive
2    static_traversable_env ::
3    abstract_env  $\Rightarrow$  flow_set  $\Rightarrow$  env  $\Rightarrow$  bool and

```

```

4   static_traversable_val ::
5     abstract_env  $\Rightarrow$  flow_set  $\Rightarrow$  val  $\Rightarrow$  bool where
6   Intro:
7      $\forall x v .$ 
8     env x = Some v  $\longrightarrow$ 
9     {rslt_var v}  $\subseteq V x \wedge$  static_traversable_val V F v
 $\Longrightarrow$ 
10    static_traversable_env V F env |
11
12   Unit:
13     static_traversable_val V F VUnit |
14   Chan:
15     static_traversable_val V F (VChn c) |
16   SendEvt:
17     static_traversable_env V F env  $\Longrightarrow$ 
18     static_traversable_val V F (VClsr (SendEvt _ _) env) |
19   RecvEvt:
20     static_traversable_env V F env  $\Longrightarrow$ 
21     static_traversable_val V F (VClsr (RecvEvt _) env) |
22   Left:
23     static_traversable_env V F env  $\Longrightarrow$ 
24     static_traversable_val V F (VClsr (Left _) env) |
25   Right:
26     static_traversable_env V F env  $\Longrightarrow$ 
27     static_traversable_val V F (VClsr (Right _) env) |
28   Abs:
29     static_traversable V F e  $\Longrightarrow$ 
30     static_traversable_env V F env  $\Longrightarrow$ 
31     static_traversable_val V F (VClsr (Abs f x e) env) |
32   Pair:
33     static_traversable_env V F env  $\Longrightarrow$ 
34     static_traversable_val V F (VClsr (Pair _ _) env)
35
36
37
38   inductive static_traversable_stack ::
39     abstract_env  $\Rightarrow$  flow_set  $\Rightarrow$  cont list  $\Rightarrow$  bool where
40   Empty:
41     static_traversable_stack V F [] |
42   Nonempty:
43     static_traversable V F e  $\Longrightarrow$ 
44     static_traversable_env V F env  $\Longrightarrow$ 
45     static_traversable_stack V F k  $\Longrightarrow$ 
46     static_traversable_stack V F ((Ctn x e env) # k))
47
48   inductive
49   static_traversable_pool ::
50     abstract_env  $\Rightarrow$  flow_set  $\Rightarrow$ 
51     trace_pool  $\Rightarrow$  bool where
52   Intro:

```

```

53   ∀ pi e env k .
54     E pi = Some (Stt e env k) →
55       static_traversable V F e ∧
56       static_traversable_env V F env ∧
57       static_traversable_stack V F k ⇒
58       static_traversable_pool V F E
59
60
1
2
3 lemma static_traversable_pool_preserved_star: "
4   static_traversable_pool V F
5   ([[[] ↦ (Stt e (λ _ . None) [])]]) ⇒
6   static_eval (V, C) e ⇒
7   trpl' pi = Some (Stt (Let x b en) envk) ⇒
8   star concur_step
9   ([[[] ↦ (Stt e (λ _ . None) [])]], { })
10  (trpl', ys') ⇒
11  isEnd (NLet x) ⇒
12  static_traversable_pool V F trpl' "
13 proof sketch
14 qed
15
16 lemma static_traversable_pool_implies_static_traceable: "
17 trpl' pi = Some (Stt (Let x b in en) env k) ⇒
18 concur_step
19 ([[[] ↦ (Stt e (λ _ . None) [])]], { })
20 (trpl', ys') ⇒
21 static_eval (V, C) e ⇒
22 static_traversable_pool V F trpl' ⇒
23 isEnd (NLet x) ⇒
24 ∃ path .
25   paths_correspond pi path ∧
26   static_traceable V F (top_node_label e) isEnd path "
27 proof sketch
28 qed
29
1
2 lemma not_static_traceable_sound: "
3 trpl' pi = Some (Stt (Let x b en) env k) ⇒
4 star concur_step
5 ([[[] ↦ (Stt e (λ _ . None) [])]], { })
6 (trpl', ys') ⇒
7 static_eval (V, C) e ⇒
8 static_traversable V F e ⇒
9 isEnd (NLet x) ⇒
10 ∃ path .
11 paths_correspond pi path ∧

```

```

12   static_traceable V F (top_node_label e) isEnd path
13   proof sketch
14   qed
15

```

```

1
2
3   interpretation communication_sound_A
4   proof -
5
6

```

15 Static Communication part B

```

1 datatype edge_label = ENxt | ESawn | ESend var | ECall |
   ERtn var
2
3 type_synonym flow_label = (node_label * edge_label *
   node_label)
4
5 type_synonym flow_set = flow_label set
6
7 type_synonym step_label = (node_label * edge_label)
8
9 type_synonym abstract_path = step_label list
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

19     static_traversable V F static_recv_site e  $\impl$ 
20     static_traversable V F static_recv_site (Let x (Prim (
    RecvEvt xc)) e) |
21 let_Pair:
22     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\impl$ 
23     static_traversable V F static_recv_site e  $\impl$ 
24     static_traversable V F static_recv_site (Let x (Prim (
    Pair x1 x2)) e) |
25 let_Left:
26     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\impl$ 
27     static_traversable V F static_recv_site e  $\impl$ 
28     static_traversable V F static_recv_site (Let x (Prim (
    Lft xp)) e) |
29 let_Right:
30     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\impl$ 
31     static_traversable V F static_recv_site e  $\impl$ 
32     static_traversable V F static_recv_site (Let x (Prim (
    Rght xp)) e) |
33 let_Abs:
34     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\impl$ 
35     static_traversable V F static_recv_site eb  $\impl$ 
36     static_traversable V F static_recv_site e  $\impl$ 
37     static_traversable V F static_recv_site (Let x (Prim (
    Abs f xp eb)) e) |
38 let_Spawn:
39     {(NLet x, ENxt, top_node_label e),
40      (NLet x, ESpawn, top_node_label ec)}  $\subseteq$  F  $\impl$ 
41     static_traversable V F static_recv_site ec  $\impl$ 
42     static_traversable V F static_recv_site e  $\impl$ 
43     static_traversable V F static_recv_site (Let x (Spwn ec)
    e) |
44 let_Sync:
45     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\impl$ 
46     ( $\forall$  xSC xM xC y .
47      { $\sim$ SendEvt xSC xM}  $\subseteq$  V xSE  $\longrightarrow$ 
48      { $\sim$ Chan xC}  $\subseteq$  V xSC  $\longrightarrow$ 
49      static_recv_site xC (NLet y)  $\longrightarrow$ 
50      {(NLet x, ESend xSE, NLet y)}  $\subseteq$  F)  $\impl$ 
51     static_traversable V F static_recv_site e  $\impl$ 
52     static_traversable V F static_recv_site (Let x (Sync xSE
    ) e) |
53 let_Fst:
54     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\impl$ 
55     static_traversable V F static_recv_site e  $\impl$ 
56     static_traversable V F static_recv_site (Let x (Fst xp
    ) e) |
57 let_Snd:
58     {(NLet x, ENxt, top_node_label e)}  $\subseteq$  F  $\impl$ 
59     static_traversable V F static_recv_site e  $\impl$ 
60     static_traversable V F static_recv_site (Let x (Snd xp

```

```

    ) e) |
61 let_Case:
62   {
63     (NLet x, ECall, top_node_label el),
64     (NLet x, ECall, top_node_label er),
65     (NRslt (rslt_var el), ERtn x, top_node_label e),
66     (NRslt (rslt_var er), ERtn x, top_node_label e)}  $\subseteq$  F
     $\implies$ 
67     static_traversable V F static_recv_site el  $\implies$ 
68     static_traversable V F static_recv_site er  $\implies$ 
69     static_traversable V F static_recv_site e  $\implies$ 
70     static_traversable V F static_recv_site (Let x (Case
    xs xl el xr er) e) |
71 let_App:
72   ( $\forall$  f' xp eb .  $\wedge$  Abs f' xp eb  $\in$  V f  $\longrightarrow$ 
73     {(NLet x, ECall, top_node_label eb),
74      (NRslt (rslt_var eb), ERtn x, top_node_label e)})
     $\subseteq$  F)  $\implies$ 
75     static_traversable V F static_recv_site e  $\implies$ 
76     static_traversable V F static_recv_site (Let x (App f xa
    ) e)
77
78
1 inductive
2   static_built_on_chan :: "abstract_env  $\Rightarrow$  node_map  $\Rightarrow$  var
     $\Rightarrow$  var  $\Rightarrow$  bool"
3 where
4   Chan:
5     AChn xc  $\in$  V x  $\implies$ 
6     static_built_on_chan V Ln xc x |
7   SendEvt:
8     APrim (SendEvt xsc xm)  $\in$  V x  $\implies$ 
9     static_built_on_chan V Ln xc xsc  $\vee$ 
    static_built_on_chan V Ln xc xm  $\implies$ 
10    static_built_on_chan V Ln xc x |
11   RecvEvt:
12     APrim (RecvEvt xrc)  $\in$  V x  $\implies$ 
13     static_built_on_chan V Ln xc xrc  $\implies$ 
14     static_built_on_chan V Ln xc x |
15   Pair:
16     APrim (Pair x1 x2)  $\in$  V x  $\implies$ 
17     static_built_on_chan V Ln xc x1  $\vee$  static_built_on_chan
    V Ln xc x2  $\implies$ 
18     static_built_on_chan V Ln xc x |
19   Left:
20     APrim (Left xa)  $\in$  V x  $\implies$ 
21     static_built_on_chan V Ln xc xa  $\implies$ 
22     static_built_on_chan V Ln xc x |
23   Right:

```

```

24     APrim (Right xa) ∈ V x ⇒
25     static_built_on_chan V Ln xc xa ⇒
26     static_built_on_chan V Ln xc x |
27 Abs:
28     APrim (Abs f xp eb) ∈ V x ⇒
29     /Set.is_empty (Ln (nodeLabel eb) - {xp}) ⇒
30     static_built_on_chan V Ln xc x
31

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

1 inductive static_live_chan ::
2   abstract_env ⇒ node_map ⇒ node_map ⇒ var ⇒ exp ⇒ bool
3   where
4   Result:
5     chan_set V Ln xc y = Ln (NRslt y) ⇒
6     static_live_chan V Ln Lx xc (Rslt y) |
7   Let_Unit:
8     static_live_chan V Ln Lx xc e ⇒
9     Ln (top_node_label e) = Lx (NLet x) ⇒
10    Lx (NLet x) = Ln (NLet x) ⇒
11    static_live_chan V Ln Lx xc (Let x Unt e) |
12  Let_Chan:
13    static_live_chan V Ln Lx xc e ⇒
14    Ln (top_node_label e) = Lx (NLet x) ⇒
15    (Lx (NLet x) - {x}) = Ln (NLet x) ⇒
16    static_live_chan V Ln Lx xc (Let x MkChn e) |
17  Let_SendEvt:
18    static_live_chan V Ln Lx xc e ⇒
19    Ln (top_node_label e) = Lx (NLet x) ⇒
20    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xsc ∪
21    chan_set V Ln xc xm = Ln (NLet x) ⇒
22    static_live_chan V Ln Lx xc (Let x (Prim (SendEvt xsc xm
23      )) e) |
24  Let_RecvEvt:
25    static_live_chan V Ln Lx xc e ⇒
26    Ln (top_node_label e) = Lx (NLet x) ⇒
27    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc xrc = Ln (NLet x)
28    ⇒
29    static_live_chan V Ln Lx xc (Let x (Prim (RecvEvt xrc))
30      e) |
31  Let_Pair:
32    static_live_chan V Ln Lx xc e ⇒
33    Ln (top_node_label e) = Lx (NLet x) ⇒
34    (Lx (NLet x) - {x}) ∪ chan_set V Ln xc x1 ∪
35    chan_set V Ln xc x2 = Ln (NLet x) ⇒

```

```

32     static_live_chan V Ln Lx xc (Let x (Prim (Pair x1 x2))
33     e) |
33 Let_Left:
34     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
35     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
36     (Lx (NLet x) - {x})  $\cup$  chan_set V Ln xc xa = Ln (NLet x
37     )  $\Rightarrow$ 
38     static_live_chan V Ln Lx xc (Let x (Prim (Lft xa)) e)
39     |
38 Let_Right:
39     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
40     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
41     (Lx (NLet x) - {x})  $\cup$  chan_set V Ln xc xa = Ln (NLet x
42     )  $\Rightarrow$ 
43     static_live_chan V Ln Lx xc (Let x (Prim (Rght xa)) e)
44     |
43 Let_Abs:
44     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
45     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
46     static_live_chan V Ln Lx xc eb  $\Rightarrow$ 
47     (Lx (NLet x) - {x})  $\cup$ 
48     (Ln (top_node_label eb) - {xp}) = Ln (NLet x)  $\Rightarrow$ 
49     static_live_chan V Ln Lx xc (Let x (Prim (Abs f xp eb)
50     ) e) |
50 Let_Spawn:
51     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
52     static_live_chan V Ln Lx xc ec  $\Rightarrow$ 
53     Ln (top_node_label e)  $\cup$  Ln (top_node_label ec) = Lx (
54     NLet x)  $\Rightarrow$ 
55     (Lx (NLet x) - {x}) = Ln (NLet x)  $\Rightarrow$ 
56     static_live_chan V Ln Lx xc (Let x (Spwn ec) e) |
56 Let_Sync:
57     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
58     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
59     (Lx (NLet x) - {x})  $\cup$  chan_set V Ln xc xe = Ln (NLet x
60     )  $\Rightarrow$ 
61     static_live_chan V Ln Lx xc (Let x (Sync xe) e) |
61 Let_Fst:
62     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
63     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
64     (Lx (NLet x) - {x})  $\cup$  chan_set V Ln xc xa = Ln (NLet x
65     )  $\Rightarrow$ 
66     static_live_chan V Ln Lx xc (Let x (Fst xa) e) |
66 Let_Snd:
67     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
68     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
69     (Lx (NLet x) - {x})  $\cup$  chan_set V Ln xc xa = Ln (NLet x
70     )  $\Rightarrow$ 
71     static_live_chan V Ln Lx xc (Let x (Snd xa) e) |
71 Let_Case:

```



```

72     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
73     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
74     static_live_chan V Ln Lx xc el  $\Rightarrow$ 
75     static_live_chan V Ln Lx xc er  $\Rightarrow$ 
76     (Lx (NLet x) - {x})  $\cup$  chan_set V Ln xc xs  $\cup$ 
77     (Ln (top_node_label el) - {xl})  $\cup$ 
78     (Ln (top_node_label er) - {xr}) = Ln (NLet x)  $\Rightarrow$ 
79     static_live_chan V Ln Lx xc (Let x (Case xs xl el xr
er) e) |
80 Let_App:
81     static_live_chan V Ln Lx xc e  $\Rightarrow$ 
82     Ln (top_node_label e) = Lx (NLet x)  $\Rightarrow$ 
83     (Lx (NLet x) - {x})  $\cup$ 
84     chan_set V Ln xc f  $\cup$ 
85     chan_set V Ln xc xa = Ln (NLet x)  $\Rightarrow$ 
86     static_live_chan V Ln Lx xc (Let x (App f xa) e)
87
88
89

```

```

1
2 inductive static_traceable ::
3   flow_set  $\Rightarrow$  node_label
4    $\Rightarrow$  abstract_path  $\Rightarrow$  bool" where
5   Empty:
6     static_traceable F end [] |
7   Edge:
8     (start, edge, end)  $\in$  F  $\Rightarrow$ 
9     static_traceable F end [(start, edge)] |
10  Step:
11    static_traceable F end ((middle, edge') # post)  $\Rightarrow$ 
12    (start, edge, middle)  $\in$  F  $\Rightarrow$ 
13    path = [(start, edge), (middle, edge')] @ post  $\Rightarrow$ 
14    static_traceable F end path
15
16

```

```

1
2 inductive static_live_traversable :: "flow_set  $\Rightarrow$  node_map
 $\Rightarrow$  node_map  $\Rightarrow$  flow_label  $\Rightarrow$  bool" where
3 Next: "
4   (l, ENxt, l')  $\in$  F  $\Rightarrow$ 
5   /Set.is_empty (Lx l)  $\Rightarrow$ 
6   /Set.is_empty (Ln l')  $\Rightarrow$ 
7   static_live_traversable F Ln Lx (l, ENxt, l')
8 " |
9 Spawn: "
10  (l, ESawn, l')  $\in$  F  $\Rightarrow$ 
11  /Set.is_empty (Lx l)  $\Rightarrow$ 
12  /Set.is_empty (Ln l')  $\Rightarrow$ 

```

```

13     static_live_traversable F Ln Lx (l, ESpwn, l')
14 " |
15 Call_Live_Outer: "
16   (l, ECall, l') ∈ F ⇒
17   /Set.is_empty (Lx l) ⇒
18   static_live_traversable F Ln Lx (l, ECall, l')
19 " |
20 Call_Live_Inner: "
21   (l, ECall, l') ∈ F ⇒
22   /Set.is_empty (Ln l') ⇒
23   static_live_traversable F Ln Lx (l, ECall, l')
24 " |
25 Return: "
26   (l, ERtn x, l') ∈ F ⇒
27   /Set.is_empty (Ln l') ⇒
28   static_live_traversable F Ln Lx (l, ERtn x, l')
29 " |
30 Send: "
31   ((NLet xSend), ESend xE, (NLet xRecv)) ∈ F ⇒
32   {xE} ⊆ (Ln (NLet xSend)) ⇒
33   static_live_traversable F Ln Lx ((NLet xSend), ESend xE,
34   (NLet xRecv))
35 "
36
1
2
3 inductive static_live_traceable :: "abstract_env ⇒ flow_set
   ⇒ node_map ⇒ node_map ⇒ node_label ⇒ (node_label ⇒
   bool) ⇒ abstract_path ⇒ bool" where
4 Empty:
5   isEnd start ⇒
6   static_live_traceable V F Ln Lx start isEnd [] |
7 Edge:
8   isEnd end ⇒
9   static_live_traversable F Ln Lx (start, edge, end) ⇒
10  static_live_traceable V F Ln Lx start isEnd [(start,
   edge)] |
11 Step:
12  static_live_traceable V F Ln Lx middle isEnd ((middle,
   edge') # path) ⇒
13  static_live_traversable F Ln Lx (start, edge, middle) ⇒
14  static_live_traceable V F Ln Lx start isEnd ((start,
   edge) # (middle, edge') # path) |
15 Pre_Return:
16  static_live_traceable V F Ln Lx (NRslt y) isEnd ((NRslt
   y, ERtn x) # post) ⇒
17  static_traceable F (NRslt y) pre ⇒
18  /static_balanced (pre @ [(NRslt y, ERtn x)]) ⇒

```

```

19   /Set.is_empty (Lx (NLet x)) ==>
20   path = pre @ (NRslt y, ERtn x) # post ==>
21   static_live_traceable V F Ln Lx start isEnd path
22
23
1
2
3   inductive static_inclusive ::
4     abstract_path => abstract_path => bool where
5   Prefix1:
6     prefix pi1 pi2 ==>
7     pi1 static_inclusive pi2 |
8   Prefix2:
9     prefix pi2 pi1 ==>
10    pi1 static_inclusive pi2 |
11   Spawn1:
12    static_inclusive (pi @ (NLet x, ESpwn) # pi1) (pi @ (
13      NLet x, ENxt) # pi2) |
14   Spawn2:
15    static_inclusive (pi @ (NLet x, ENxt) # pi1
16      static_inclusive) (pi @ (NLet x, ESpwn) # pi2) |
17   Send1:
18    static_inclusive (pi @ (NLet x, ESend xE) # pi1) (pi @ (
19      NLet x, ENxt) # pi2) |
20   Send2:
21    static_inclusive (pi @ (NLet x, ENxt) # pi1) (pi @ (NLet
22      x, ESend xE) # pi2)
23
1
2
3
4   inductive singular ::
5     abstract_path => abstract_path => bool where
6   equal:
7     pi1 = pi2 ==>
8     singular pi1 pi2 |
9   exclusive:
10    /(pi1 static_inclusive pi2) ==>
11    singular pi1 pi2
12
13   inductive noncompetitive ::
14     abstract_path => abstract_path => bool" where
15   ordered:
16     ordered pi1 pi2 ==>
17     noncompetitive pi1 pi2 |
18   exclusive:
19    /(pi1 static_inclusive pi2) ==>

```

```

20     noncompetitive pi1 pi2
21
22 inductive static_one_shot :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$ 
    bool where
23   Sync:
24     every_two (static_live_traceable V F Ln Lx (NLet xC) (
25       static_send_node_label V e xC)) singular  $\Rightarrow$ 
26     static_live_chan V Ln Lx xC e  $\Rightarrow$ 
27     static_traversable V F (static_recv_node_label V e) e  $\Rightarrow$ 
28     static_one_shot V e xC
29 inductive static_one_to_one :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$ 
    bool where
30   Sync:
31     every_two (static_live_traceable V F Ln Lx (NLet xC) (
32       static_send_node_label V e xC)) noncompetitive  $\Rightarrow$ 
33     every_two (static_live_traceable V F Ln Lx (NLet xC) (
34       static_recv_node_label V e xC)) noncompetitive  $\Rightarrow$ 
35     static_live_chan V Ln Lx xC e  $\Rightarrow$ 
36     static_traversable V F (static_recv_node_label V e) e  $\Rightarrow$ 
37     static_one_to_one V e xC
38 inductive static_fan_out :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$ 
    bool where
39   Sync:
40     every_two (static_live_traceable V F Ln Lx (NLet xC) (
41       static_send_node_label V e xC)) noncompetitive  $\Rightarrow$ 
42     static_live_chan V Ln Lx xC e  $\Rightarrow$ 
43     static_traversable V F (static_recv_node_label V e) e  $\Rightarrow$ 
44     static_fan_out V e xC
45 inductive static_fan_in :: abstract_env  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$  bool
    where
46   Sync:
47     every_two (static_live_traceable V F Ln Lx (NLet xC) (
48       static_recv_node_label V e xC)) noncompetitive  $\Rightarrow$ 
49     static_live_chan V Ln Lx xC e  $\Rightarrow$ 
50     static_traversable V F (static_recv_node_label V e) e  $\Rightarrow$ 
51     static_fan_in V e xC

1 locale communication_sound_B =
2   Static_Communication.communication_sound static_one_shot
    static_fan_out static_fan_in static_one_to_one
3

1
2

```

```

1
1 interpretation communication_sound_B
2 proof sketch
3 qed
4
1
1
1
1
2
1
2
3 let lp = fun lp x =>
4   let z1 = case x of
5     L y => let z2 = lp y in z2 |
6     R () => let z3 = () in z3
7   in ()
8   in
9
10 let mksr = fun _ x =>
11   let ch1 = mkChan () in
12   let z4 = (lp (L (L (R ()))) in
13   let srv = fun srv x =>
14     let p = sync (recv_evt ch1) in
15     let v1 = fst p in
16     let ch2 = snd p in
17     let z5 = sync (send_evt ch2 x) in
18     let z6 = srv v1 in ()
19   in
20   let z7 = spawn (
21     let z8 = srv (R ()) in ()) in
22   ch1 in
23
24 let rqst = fun _ pair =>
25   let ch3 = fst pair in
26   let v2 = snd pair in
27   let ch4 = chan () in
28   let z9 = sync (send_evt ch3 (v2, ch4)) in
29   let v3 = sync (recv_evt ch4) in
30   v3 in
31
32 let srvr = mksr () in
33 let z10 = spawn (
34   let z11 = rqst (srvr, R ()) in ())

```

```
35      in
36  let z12 = rqst (srvr, L (R ())) in
37  ()
38
39
1
1
```