Master's Thesis: Formal Theory of Communication in Concurrent ML
Thomas Logan

Summary

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

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.

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

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

channel before the sending thread can continue. Likewise, when a thread receives on a channel, another thread must send on the same channel before the receiving thread can continue.

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

A given channel can have any arbitrary number of threads sending or receiving data on it over the course of the program's execution. 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.

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

Listing 1: Server signature

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

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

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

Listing 2: Implementation of servers

The server implementation, given in Listing 2, defines a server that holds a number `state`. When a client gives the server a number `v`, the server gives back `state`, and holds onto `v` to as its new `state`, which it gives to the next client and so on. A request and reply is equivalent to reading and writing a mutable cell in isolation. The function `make` makes a new server. It creates a new channel `reqCh`, from which the server will receive requests. The sever behavior is defined by the infinite loop `loop`, which takes a number as the state of each iteration. Each iteration, the server tries to receive requests on `reqCh`. It expects the request to be composed of a number `v` and a channel `replCh`, through which to reply. Once a request has been received, it sends the current state back to the client through `replCh` by calling `send (replCh, state)`. It initiates the next iteration of the loop by calling `loop` with a new state from the client. The server is created with a new thread by calling `spawn (fn () => loop 0)`. A handle to the new server is returned as `reqCh` wrapped in the constructor `S`. The function `call` makes a request to a server `server` with a number `v` and returns the number from the server's reply. It extracts the request channel `reqCh` from the server handle and creates a new channel `replCh`, from which the client will receive replies. It makes a request to the server with the number `v` and the reply channel `replCh` by calling `send (reqCh, (v, replCh))`. Then it receives the reply with the new number by calling `recv replCh`.

Reppy's original design of Concurrent ML allows for events other than sending and receiving to be triggered by synchronization. One such event chooses between one of many events to synchronize on. Only one of the events is chosen for synchronization, but all choices must be represented. Thus, event synchronization must be separated from event values, similar to the way function application is separated from function abstraction. `send` and `recv` are just shorthand for synchronization on send and receive events, respectively.

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

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

`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.

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

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.

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

Since there are four threads that make calls to the server, the server's particular `reqCh` has four senders. Servers are created with only one thread listening for requests, so the `reqCh` of this server has just one receiver. So the server's `reqCh` is classified as fan-in. Each use of `call` creates a

distinct new channel `replCh` for receiving data. The function `call` receives on the channel once and the server sends on the channel once, so each instance of `replCh` is one-shot.

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

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

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

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

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.

```
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
7. in
8.   y := 0;
9.   (!y + 2) - (!x + 1) * (w - 3)
10.     end
```

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.

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

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

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

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

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

analyses are known as control flow analyses or CFAs. With the control flow information, other data flow analyses like available expression analysis and liveness analysis can provide greater coverage.

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

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

The algorithm proceeds in multiple steps that produce intermediate data structures, used for efficient lookup in the subsequent steps. It starts with a control-flow analysis[12, 13] that results in multiple mappings. One mapping is from variables to abstract values that may bind to the variables. Another mapping is from channel-bound variables to abstract values that are sent on the respective channels. Another is from function-bound variables to abstract values that are the result of respective function applications. It constructs a control-flow graph with possible paths for pattern matching and thread spawning determined directly from the primitives used in the program. Relying on information from the mappings to abstract values, it constructs the possible paths of execution via function application and channel communication. It uses the graph for live variable analysis of channels, which limits the scope for the remaining analysis. Using the spawn and application edges of the control-flow graph, the algorithm then performs a data-flow analysis to determine a mapping from program points to all possible control paths leading into the respective program points. Using the CFA's mappings to abstract values, the algorithm determines the program points for sends and receives per channel variable. Then it uses the mapping to control paths to determine all control paths that send or receive on each channel, from which it classifies channels as one-shot, one-to-one, fan-in, fan-out, or many-to-many.

Reppy and Xiao informally prove soundness of their analysis by showing that their analysis claims that more than one thread sends (or receives) on a channel if the execution allows more than one to send (or receive) on a that channel. The proof of soundness depends on the ability to relate the execution of a program to the static analysis of a program. The static analysis describes threads in terms of control paths, since it can only describe threads in terms of statically available information.

Thus, in order to describe the relationship between the threads of the static analysis and the operational semantics, the operational semantics is defined as stepping between sets of control paths paired with terms. Divergent control paths are added whenever a new thread is spawned.

The syntax, semantics, and analysis need to describe many details. Proving propositions relating all of these definitions requires manipulation of all those details. To ensure the correctness of proofs, it is necessary to check that there are no subtle errors in the definitions or proofs. Proofs in general require many subtle manipulations of symbols. The difference between a false statement and a true statement can often be difficult to spot, since the two may be very similar lexically. However, a mechanical proof checker, such as the one in Isabelle, has no difficulty discerning between valid and invalid derivations of statements. Mechanical checking of proofs can notify us of errors in the proofs or definitions far better and faster than manual checking. I have already benefitted from Isabelle's proof checker in order to correctly define the language semantics and abstract value flow analysis for this work. While trying to prove soundness of the analysis, the proof assistant would not accept my proof unless I provided derivation of facts that I believed to be false. I determined that my intuition was correct but my definitions had errors. After correcting the errors, I was able to complete the proof, such that the proof checker was satisfied.

Although Isabelle is described as a proof assistant[17], it is really a generic system for processing any kind of code. The code could be proofs, propositions, programs, or types. The processing could be checking proofs, interpreting programs, or translating code. Code and logics for processing code are defined by users using its meta-language Standard ML, and other user-defined languages. Isabelle/HOL is a higher-order logic built from Isabelle's primitives and other logics. It is useful for both programming and proving. Its ability to check that proofs satisfy propositions is simply one instance of its verification capabilities. It can also check that program terms satisfy types, similar to other programming systems for ML. Proofs and propositions are analogous to terms and types, respectively, yet Isabelle/HOL treats the two concepts distinctly. The practical uses for terms are quite different from that of proofs. If a *term* satisfies a *type*, then the term has utility for the data or computation it represents. The type is only valuable for confirming or denying the usage of a term. In contrast, once a *proof* satisfies a *proposition*, the proof becomes irrelevant, while the proposition is elevated to a theorem. The theorem is useful on its own without regard to any particular proof.

Similar to other programming languages, type `bool` can be satisfied by values `True` or `False`. In contrast to other programming languages, additional syntax, or data constructors, can be defined to satisfy the type `bool`. A constructor can take any number of terms of any types as input in order to create a boolean term. Although these new terms could be used in programs, just as `True` and `False` are, their main utility is in theorem proving. In Isabelle/HOL, propositions are isomorphic to terms of type `bool`. The constructors are defined with a set of inference rules, where each inference rule defines the conditions sufficient for a construction to be valid, and at least one of the enumerated

conditions is necessary for a valid construction . In other words, the constructor is equivalent to the boolean sum of all the conditions. The proposition definitions in terms of inference rules, or inductive definitions, are analogous to datatype definitions, just as propositions are analogous to types.

```
datatype 'a list = Nil | Cons 'a "'a list"

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

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`.

```
datatype nat = Z | S nat

inductive lte :: "nat => nat => bool" where
  Eq : "lte n n" |
  Lt : "lte n1 n2 ==> lte n1 (S n2)"

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

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.

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

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

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.

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.

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.

Implementation

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

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

The benefits of specialization would be much more significant ins multiprocessor implementations rather than single processor implementations. A single processor implementation

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

ManyToMany Channel

```
structure ManyToManyChan : CHAN = struct

  type message_queue = 'a option ref queue

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

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

  fun channel () = Ch (ref Inactive, mutexLock ())

  fun send (Ch (contentRef, lock)) m =
    acquire lock;
    (case !contentRef of
      Recv q =>
        let
          val (recvCond, mopRef) = dequeue q
        in
          mopRef := Some m;
          if (isEmpty q) then contentRef := Inactive else ();
          release lock;
          signal recvCond;
          ()
        end |
      Send q =>
        let
          val sendCond = condition ()
        in
          enqueue (q, (sendCond, m));
          release lock;
          wait sendCond;
          ()
        end |
```

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

  fun recv (Ch (contentRef, lock)) =
    acquire lock;
    (case !contentRef of
      Send q =>
        let
          val (sendCond, m) = dequeue q
        in
          if (isEmpty q) then contentRef := Inactive else ();
          release lock;
          signal sendCond;
          m
        end |
      Recv q =>
        let
          val recvCond = condition ()
          val mopRef = ref None
        in
          enqueue (q, (recvCond, mopRef));
          release lock;
          wait recvCond;
          valOf (!mopRef) |
        end
      Inactive =>
        let
          val recvCond = condition ()
          val mopRef = ref None
        in
          contentRef := Recv (queue [(recvCond, mopRef)]);
          release lock;
          wait recvCond;
          valOf (!mopRef)
        end)

end
```

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.

FanOut Channel

```
structure FanOutChan : CHAN = struct

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

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

  fun channel () = Ch (ref Inactive, mutexLock ())

  fun send (Ch (contentRef, lock)) m = let
    val sendCond = condition ()
  in
    case compareAndSwap (contentRef, Inactive, Send (sendCond, m)) of
      Inactive =>
```

```
        (* contentRef already set *)
        wait sendCond;
        () |
      Recv q =>
        (* the current thread is the only one that updates from this state *)
        acquire lock;
        (let
          val (recvCond, mopRef) = dequeue q
        in
          mopRef := Some m;
          if (isEmpty q) then contentRef := Inactive else ();
          release lock;
          signal (recvCond);
          ()
        end) |
      Send _ => raise NeverHappens
    end

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

end
```

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

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

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

FanIn Channel

```
structure FanInChan : CHAN = struct

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

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

  fun channel () = Ch (ref Inactive, mutexLock ())

  fun send (Ch (contentRef, lock)) m =
    acquire lock;
```

```
    case !contentRef of
      Recv (recvCond, mopRef) =>
        mopRef := Some m;
        contentRef := Inactive;
        release lock;
        signal recvCond;
        () |
      Send q =>
        let
          val sendCond = condition ()
        in
          enqueue (q, (sendCond, m));
          release lock;
          wait sendCond;
          ()
        end |
      Inactive =>
        let
          val sendCond = condition ()
        in
          contentRef := Send (queue [(sendCond, m)])
          release lock;
          wait sendCond;
          ()
        end


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

  end
```

```
end
```

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

OneToOne Channel

```
structure OneToOneChan : CHAN = struct

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

      datatype 'a chan = Ch of 'a chan_content ref

  fun channel () = Ch (ref Inactive)

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


  fun recv (Ch contentRef) = let
    val recvCond = condition ();
    val mopRef = ref None;
  in
    case compareAndSwap (contentRef, Inactive, Recv (recvCond, mopRef)) of
      Inactive =>
        (* contentRef already set to Recv*)
        wait recvCond;
        valOf (!mopRef) |
      Send (sendCond, m) =>
        (* the current thread is the only one that accesses contentRef for this
state*)
        contentRef := Inactive;
        signal sendCond;
```

```
        m |
      Recv _ => raise NeverHappens
  end

end
```

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

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

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

OneShot Channel

```
structure OneShotChan : CHAN = struct

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

  datatype 'a chan = Ch of 'a chan_content ref * mutex_lock
```

```
  fun channel () = Ch (ref Inactive, lock ())

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


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

end
```

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

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

The recv operation checks if the channel's state is inactive and tries to use the compareAndSwap

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

OneShotToOne Channel

```
structure OneShotToOneChan : CHAN = struct

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

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

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

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

end
```

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

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

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.

Informal Objectives

To optimize a Concurrent ML program with specialized implementations of communication, one needs to determine which specialized implementations in what context of the program are behaviorally equivalent to the original implementations. In this work we take the view that if some channel has some maximum number of threads competing to send or receive on it, then the implementation of the channel, send, and receive may be specialized to a faster version. We support this view informally, by providing example specialized implementations and stating the communication topologies for which they are valid. However, such claims should not be trusted, and we hope that future work can develop formal proofs of correct transformations by relying on the formal proofs of soundness that developed in this work. For example:

Suppose structure Chan is a general implementation of communication, and OneToOneChan is a specialized implementation of communication; if channel c is one-to-one,
c is implemented with Chan in p, and program px = p with c's implementation replaced by OneToOneChan, then evaluation of p is behaviorally equivalent to the evaluation of px.

Such a theorem tells us if a transformation is correct, but it doesn't tell if we can find the necessary conditions for making the transformation. In addition, we need proof that we can determine that a channel is one-to-one or can be classified as some other topology. Since programs are not required to terminate communication topologies are defined by the runtime communication, it is not always possible to determine a precise communication topology. However it should be possible to determine an approximate communication topology, or the static communication topology. With a proof that the static communication topology is computable, along with proof that static topology implies some runtime topology, the we can show that the necessary conditions attainable.

In this work, we focus on the formal proofs that demonstrate that static communication topology classifications imply certain runtime communication topology classifications, i.e. proofs of soundness of static communication topology classification. For one-to-one classification, the soundness of static classification

if variable x is statically determined to be a channel with one-to-one communication topology in program p, then for all states v if p evaluates to v, then all channels created with binding x have one-to-one topology in program p's evaluation to state v.

We could also create an analogous description of evaluation with a static description of evaluation, describing abstract values representing all possible values produced by evaluation program p up to any state v.

if program p statically evaluates to abstract values in W, and variable x is statically determined to be a channel with one-to-one communication topology in program p with abstract values W, then if variable x is statically determined to be a channel with one-to-one communication topology in program p, then for all states v where p evaluates to v, all channels created with binding x have one-to-one topology in program p's evaluation to state v.

Factoring out the static evaluation from communication topology classification, not only gives a nice symmetry with the runtime evaluation, but also emphasizes the importance of the describing abstract values, common to all communication topologies.

Formal Objectives

In order to give a convincing proof of the soundness, we need precise descriptions of the soundness theorems. In fact, we give descriptions that are as precise as possible – they are formal, and our proofs are as convincing as possible – they are mechanically verifiable and automatically checked by the actively maintained software Isabelle.

Syntax

```
datatype var = Var string

datatype exp =
  Let var bind exp |
  Result var

and bind =
  Unit |
  Chan |
  Prim prim |
  Spawn exp |
  Sync var |
  Fst var |
  Snd var |
  Case var var exp var exp |
  App var var

and prim =
  SendEvt var var |
  RecvEvt var |
  Pair var var |
  Left var |
  Right var |
  Abs var var exp
```

The syntax of the language of programs is in a very restricted ANF form, which can be viewed as an intermediate representation. It consists of variables and three other mutually dependent parts. An expression has one of two forms. Either it can be a variable binding followed by an expression, or it can be a result variable. A bindee consists of the unit value literal, and a prim, which consists of literals for events, which are used for synchronization, pairs, left and right parts of sums, and function abstractions. Additionally, it consists of syntax for channel generation, thread spawning, synchronization, accessing first and second items of pairs, pattern matching and function application. `prim` is factored out of the bindee in order to concisely define certain relations on just those constructs.

Dynamic Semantics

(* insert definitions from Dynamic_Semantics.thy *)

The dynamic semantics describes how a program is evaluated for any number of steps. A control label is composed of variables and used to indicate how an expression is reached abstractly from a previous step during evaluation. A control path is a list of control labels, which is used to abstractly describe the multiple steps taken to reach a point in evaluation. A channel consists of the control path leading to its generation site and the variable that the new channel becomes bound to. A value is either the unit value, a channel instance, or a closure consisting of a primitive with an environment that maps variables to further values. A continuation frame is composed of a variable, yet to be bound, an expression that may use the variable, and an environment mapping variables in the expression to their bound values. A continuation stack is a stack of continuation frames. A state consists of an expression, yet to be evaluated, an environment containing the values bound to variables determined by previous steps, and a continuation stack.

The sequential step predicate describes that a state takes a single deterministic step in evaluation. The predicate is defined to hold by the inference rules listed in [Figure ?]. The Result rule asserts that a state containing a Result expression and a nonempty continuation stack steps to a state containing an expression from the first frame on the continuation stack. The unbound frame variable becomes bound to the value bound to the Result variable, and the remainder of the continuation stack becomes new continuation stack. The Let_Unit rule asserts that a state consisting of a Let expression with a variable binding to the unit literal, steps to a state containing the Let expression's next expression with Let binding variable bound to the unit value. The Let_Fst rule asserts that a state consisting of a Let expression with variable binding to a Fst bindee, steps to a state containing the Let expression's next expression, with the Let binding variable bound to a new value, that which is the first value of the pair value bound to the Fst bindee's variable. The Let_Snd rule is similar. The Let_Case_Left rule asserts that a state consisting of a Let expression with a variable binding to a Case bindee steps to the left expression of the Case bindee, with the Left binding variable bound in the expression to the value within a Left value, that which is bound to the Case binding variable. The

continuation stack is updated with a frame containing the Let binding variable and the Let expression's next expression, along with the original environment. The Let_Case_Right rule is similar. The Let_App rule asserts that a state consisting of a Let expression with a variable binding to a function application steps a state containing the expression within the function abstraction, along with function parameter bound to the function abstraction itself, and the abstraction's input parameter bound to value of the application argument. As with Let_Case_Left, and Let_Case_Right, the stack is updated with a frame containing the Let binding variable and the Let expression's next expression, along with the original environment.

A trace pool is a mapping from control paths to states, where is control path abstractly represents the steps taken up to the state it points to. The leaf predicate describes that a control path exists in a trace pool and is not a prefix of any other control path in the trace pool.

The concurrent step predicate means that a trace pool takes a nondeterministic step in evaluation. The Seq_Step_Down rule asserts that a trace pool may step to the same trace pool updated with an existing leaf path extended with a Down control label mapping to a new state; The existing control path leads to a state with a Result expression and a nonempty continuation stack. The front continuation frame contains the variable of the Down control label, and the state sequentially steps to the new state of the trace pool. The Seq_Step rule asserts that a trace pool may step to the same trace pool updated with an existing leaf path extended with a Seq control label mapping to a new state. The existing path leads to a state with a Let expression, where the Let binding variable is that of the Seq control label, and the state sequentially steps to the new state of the trace pool, with the continuation unchanged. The Seq_Step_Up rule asserts that a trace pool may step to the same trace pool updated with an existing leaf path extended with an Up control label mapping to a new state. The existing path leads to a state containing a Let expression. The Let binding variable is that of the Up control label, and the state sequentially steps to a new state with a new frame added to the continuation stack. The frame consists of the original state's Let binding variable, the Let expression's next expression and the environment. The existing state sequentially steps to the new state of the trace pool. The Let_Chan rule states that a trace pool steps to the same trace pool updated with a an existing leaf path extended with a Seq control label mapping to a state. The existing path leads to a state consisting of a Let expression binding to channel generation. The Let binding variable is that of the Seq control label, and the trace pool's new state consists of the Let expression's next expression with the Let binding variable bound to a new channel value with an identity constructed from the existing path and the binding variable. The Let_Spawn rule states that a trace pool may step to the same trace pool updated with two new paths mapping to two new states. One path consists of an existing leaf path extended with a Seq control label, and the other consists of a the same path extend with a Spawn control label. The existing path leads to a state containing a Let expression binding with a Spawn bindee. The new state associated with the Spawn control label consists of the Spawn bindee's expression, the existing state's environment, and an empty continuation stack. The new state associated with the Seq control label

consists of the Let expression's next expression, the original environment updated with the Let binding variable mapped to the unit value, and the same continuation stack. The Let_Sync rule asserts that a trace pool steps to the same trace pool updated with two new paths mapping to two new states. One new path is an extension of an existing leaf path that points to an expression with a binding to a sync on a variable bound to a send event. The new state associated with the extended path consists of the Let expression's next expression and the Let binding variable bound to the unit value, and unchanged continuation stack. The other new path is an extension of an existing leaf path that points to an expression with a binding to a sync on a variable bound to a receive event. The new path is associated with a new state containing the Let expressions's next expression, the Let binding variable bound to the value of of the send event's message, and an unchanged continuation stack. The channel variables of both events are bound to the same channel.

The star predicate is defined by two inference rules. The Refl rule asserts that star holds for a binary predicate and two identical terms that could be arguments for the predicate. The Step rule asserts that star holds for a binary predicate and two terms, where the predicate holds for the first term and an intermediate term, and star holds for the predicate the intermediate term and the second term. The multi concurrent step predicate is simply the star predicate partially applied to the concurrent step predicate.

Dynamic Communication Topology Classification Analysis

(* insert definitions from Dynamic_Com_Topo_Analysis.thy *)

The dynamic communication topology classification classifies channels based on the number of of threads that compete to synchronize on different events on channels. The is_send_path predicate means that a specified control path leads to a synchronization on a send event on a specified channel in a specified trace pool. The is_recv_path predicate means that a specified control path leads to a synchronization on a receive event on a specified channel in a specified trace pool. The all predicate means that a specified binary relation on control paths holds for all paths satisfying a specified predicate. Two control paths are ordered iff one is the prefix of the other. The one_shot predicate means that there is exactly one send path that leads to a specified channel in a specified trace pool, by saying that all send paths to the channel in the trace pool are equal. The one_to_one predicate means that for a specified trace pool and channel, all send paths are ordered, and all receive paths are ordered. If paths to synchronization on a channel are ordered, then they cannot compete with each other, meaning that at most one thread attempts to synchronize before a synchronization is completed. The fan_out predicate simply asserts that for a specified trace pool and channel, all send paths are ordered. The fan_in predicate means that for a specified trace pool and channel, all receive paths are ordered.

Static Semantics

(* insert definitions from Static_Semantics.thy *)

The static semantics describes an imprecise range of values associated with variables in a program during any step of the program's evaluation. Abstract values are used to represent a range of concrete values. An abstract value is either an abstract channel identified by a variable, an abstract unit value, which is just as precise as its concrete counterpart, or an abstract primitive, which represents the range of closures over that primitive. The abstract value environment is a mapping from variables to sets of abstract values. The result variable function determines the variable of a Result expression or determines the result variable of a Let expression's next expression.

The accept predicate means that two abstract value environments represent value associations of a program. The first abstract value environment V maps variables in the program to values, which they bind to, including Let bindings, Case bindings, and abstraction bindings. The second abstract value environment C maps variables to the values sent as messages over channels that are abstractly identified by the variables. The Result rule asserts that any two abstract value environments accept a Result expression. Since there are no bindings and no mention of channels in a Result expression, any binding environment and any message environment contains all values for the expression vacuously. The Let_Unit rule asserts that two abstract environments accepts a Let expression with a binding to the unit literal, where the binding environment maps the binding variable to the abstract unit value, and the abstract value environments accept the Let expression's next expression. The rules for Let_Chan, Let_Send_Evt, Let_Recv_Evt, Let_Pair, Let_Left, Let_Right are similar. The Let_Abs rule asserts that two abstract environments accept a Let expression with a binding to a function abstraction, where the function parameter of the function abstraction maps to the abstract value of the function abstraction, and the abstract value environments also accept the expression within the function abstraction. Additionally, the Let binding variable also maps to the abstract value of the function abstraction, and like previous rules, the abstract value environments accept the Let expression's next expression. The Let_Spawn rule asserts that two abstract value environments accept a Let expression with binding to a Spawn bindee, where the the Let binding variable maps to the abstract unit value in the abstract value binding environment, and the two abstract value environments accept both the expression within the Spawn bindee and the Let expression's next expression. The Let_Sync rule asserts that two abstract value environments accept a Let expression with binding to a synchronization. For all abstract send event values bound to the Sync's variable, and all abstract channel values of the abstract send event values, the abstract values of the send events' message variables are also bound to the abstract channel identifier in the message binding environment. Additionally, the Let binding variable is bound to the abstract unit value. For all abstract receive event values bound to the Sync's variable, and all abstract channels of the abstract receive event values, the message abstract value of the channel identifier is also bound to the the Let binding's variable in the binding abstract value environment. The Let_Fst rule asserts that two abstract value environments accept a Let expression with binding to an access of first of pair, where for all abstract pair values bound to the Fst's variable, the abstract value of the first variable is also bound to the Let binding's variable in the binding abstract value environment.

Additionally, the two abstract value environments accept the Let expression's next expression. The Let_Snd rule is similar. The Let_Case rule asserts that two abstract value environments accept a Let expression with binding to case pattern matching. For all the left abstract values bound to the variable of the case binding, the abstract value, wrapped in the abstract left value, will also be bound to the case left binding for use in the case left expression. The abstract value bound to the the result variable of the case left expression is also bound to Let binding variable, and the two abstract value environments accept the case left expression. For all the right abstract values bound to the variable of the case binding, the abstract value, wrapped in the abstract right value, will also be bound to the case right binding for use in the case right expression. The abstract value bound to the the result variable of the case right expression is also bound to Let binding variable, and the two abstract value environments accept the case right expression. Additionally, the two abstract value environments accept the Let expression's next expression. The Let_App rule asserts that two abstract value environments accept a Let expression with binding to a function application. For all abstract function abstraction values in the function being applied, the abstract value of the application's argument is also the abstract value of the function abstraction value's input parameter, and the abstract value of the result variable of the function abstraction's expression is also bound to the Let binding variable. As with previous rules, the two abstract value environments accept the Let expression's next expression. It is worth noting that the rule does not require that the two abstract environments also accept the expression within the function abstraction. Doing so would result in a coinductive description, since the function being applied could also be applied in its own inner expression. A coinductive description would be less amenable to a computable description, which could defeat the purpose of the static semantics.

Static Traceability

(* insert definitions from Static_Traceability.thy *)

The notion of traceability in the dynamics semantics is implicit from the existence of a control path mapping to a state in a trace pool. In contrast, the the static description of traceability requires an inductive definition. A control path is balanced iff it is empty; it consists of a exactly one control label that happens to be a Seq control label; it starts with an Up control label of a variable, followed by a balanced control path, followed by a Down control label with the same variable; or it is two balanced control paths stuck together. The static_traceable predicate means that a control path might trace to an expression from a specified starting expression and a binding abstract value environment. The Start rule simply asserts that an empty control path might trace to the same expression that it starts at. The Result rule asserts that a control path ending on a Down control label, with a balancing Up control label preceding it in the path, may trace to the next expression of a Let expression. The control path leading to the Down label might trace to a Result expression. The path between the Up and Down labels is balanced, and the the control path leading to the Up control label leads to the Let expression. The Let_Unit rule asserts that a control path ending with a Seq control label might trace to the next

expression of a Let expression, where the path prefix leading to the Seq label might trace to the Let expression. The rules for Let_Chan, Let_Prim, Let_Spawn, Let_Sync, Let_Fst, and Let_Snd are similar. The Let_Case_Left rule asserts that a control path ending in an Up label might trace to the case left expression of a Let expression, where the path prefix leading to the Up label might trace to the Let expression. The Let binding variable is the the variable of the Up label. The Let_Case_Right rule is similar. The Let_App rule asserts that a control path ending in an Up label might trace to an expression within an function abstraction bound in a Let expression, where the path prefix leading to the Up label might trace to the Let expression. The Let binding variable is the the variable of the Up label.

Static Communication Topology Classification

(* insert definitions from Static_Com_Topo_Analysis.thy *)

The static communication topology classification classifies abstract identifiers of channels based on the number of threads that might compete to synchronize on different events on channels. The is_static_send_path predicate means that a control path might be traceable to a synchronization on a send event on a specified abstract channel, for a specified expression, using abstract values from a specified abstract binding environment. The is_static_recv_path predicate means that a control path might be traceable to a synchronization on a receive event on a specified abstract channel, for a specified expression, using abstract values from a specified abstract binding environment. The inclusive predicate means that two control paths may occur in the same run of a program. The Ordered rule asserts that two paths are inclusive if they are ordered. The Spawn_Left and Spawn_Right rules say that two paths are inclusive if one path was spawned by the other. Two paths are singular iff either they are the same, or they are not inclusive, thus cannot occur in the same run of the program. Two paths are noncompetitive iff either they are ordered, or they are not inclusive. The static topology definitions follow a similar patter to their dynamic counterparts. The static_one_shot predicate means that all static send paths are singular for a specified abstract channel identifier, abstract environment, and expression. The static_one_to_one predicate means that all static send paths are noncompetitive, and all static receive paths are noncompetitive for a specified abstract channel identifier, abstract environment, and expression. The static_fan_out predicate means that all static send paths are noncompetitive for a specified abstract channel identifier, abstract environment, and expression. The static_fan_in predicate means that all static receive paths are noncompetitive for a specified abstract channel identifier, abstract environment, and expression.

Sound Communication Topology Classification Analysis

(* insert theorem from Sound_Com_Topo_Analysis.thy *)

The static description of communication topology classification analysis is useful only if the classification can be computed from the other parameters of the predicate, and if the dynamic

classification analysis can be inferred from the static classification analysis. The theorems one_shot_sound, one_to_one_sound, fan_out_sound, and fan_in_sound, together enable the inference of dynamic descriptions form the static descriptions. The theorem one_shot_sound claims that a dynamic channel is dynamically classified as one shot in a trace pool that results from multiple concurrent steps from an initial program, if the dynamic channel's variable identifier is statically classified as one shot for the program and also a binding abstract value environment, message abstract value environment, that accept the program. The remaining soundness theorems are similar.

Informal Reasoning

The communication topology classifications are defined in terms of traceability to synchronization on values for events operating on values for channels. Thus, the soundness of classifications naturally follows from the soundness of traceability, and soundness of values bound. The soundness of traceability means that a path does not dynamically trace to an expression if the path cannot statically trace to the expression. The static_traceable predicate, in the Result rule, depends on a previous expression being statically traceable. The previous expression would be one that follows and function application or case matching expression, and thus would be placed in the continuation stack during evaluation. Thus, showing the contrapostive of the soundness claim, that static traceability follows from dynamic traceability, depends on showing that all the expressions in the continuation stack of a traced state are statically traceable.

The soundness of values bound means that a value is not bound to a variable in a state during evaluation, if the abstracted version of the value cannot be statically bound.

(* describe informally top-down the basic dependencies of the soundness of values bound*)

Formal Reasoning

(* describe formal definitions used in proofs and formal proofs *)

References

1. Donnelly, K. and Fluet, M. Transactional events. ACM SIGPLAN Notices 2006, 41(9), pp.124-135.
2. Flanagan C., Sabry A., Duba B.F., Felleisen M. The Essence of Compiling with Continuations. In ACM SIGPLAN Notices; 1993 Aug 1 (Vol. 28, No. 6, pp. 237-247). ACM.
3. Fluet, M. A type-and control-flow analysis for System F. In Symposium on Implementation and Application of Functional Languages (pp. 122-139). 2012 August. Springer, Berlin, Heidelberg.
4. Matichuk D., Murray T., Wenzel M. Eisbach: A Proof Method Language for Isabelle. Journal of Automated Reasoning. 2016 Mar 1;56(3):261-82.
5. Nielson F., Nielson H.R., Hankin C. Principles of program analysis. Springer; 2015 Feb 27.
6. Nipkow T, Klein G. Concrete Semantics: With Isabelle/HOL. Springer; 2014 Dec 3.
7. Paulson L.C., Nipkow T. Isabelle Tutorial and User's Manual. University of Cambridge, Computer Laboratory; 1990.
8. Reppy, J.H. Concurrent Programming in ML. Cambridge University Press; 2007 Sep 14.
9. Reppy, J. H. CML: A Higher-Order Concurrent Language. In PLDI'91, New York, NY, June 1991. ACM, pp. 293–305.
10. Reppy, J., Russo, C.V. and Xiao, Y. Parallel concurrent ML. In ACM SIGPLAN Notices 2009 August (Vol. 44, No. 9, pp. 257-268). ACM.
11. Reppy J, Xiao Y. Specialization of CML message-passing primitives. In ACM SIGPLAN Notices 2007 Jan 17 (Vol. 42, No. 1, pp. 315-326). ACM.
12. Reppy, J. Type-sensitive control-flow analysis. In ML '06, New York, NY, September 2006. ACM, pp. 74–83.
13. Shivers, O. Control-flow analysis of higher-order languages. PhD dissertation, Carnegie Mellon University, 1991.
14. Wenzel M. The Isabelle/Isar Reference Manual.
15. Wenzel M, Haftmann F, Paulson L. The Isabelle/Isar Implementation.
16. Xiao, Y. Toward optimization of Concurrent ML. Master's dissertation, University of Chicago, December 2005.
17. http://isabelle.in.tum.de/
18. Serrano, M. Control flow analysis: a functional languages compilation paradigm. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied Computing*, New York, NY, 1995. ACM, pp. 118–122.