Master's Thesis Proposal (Draft – Nov 1, 2017)
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[?]. This work will build on Reppy and Xiao's static analysis for computing sound approximations of communication topologies[?]. We will define a small-step operational semantics for Concurrent ML and a constraint-based static analysis[?] that describes all possible communications with varying precision. Then we will prove that our analysis is sound with respect to the semantics. Our semantics, analysis, propositions, proofs, and theorems will use Isabelle/HOL[?] as the formal language of reasoning. The proofs will be mechanically checked by Isabelle[?].

## 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 language that offers a process abstraction, which is a piece of code allowed to have a wide range of evaluation orders relative to code encapsulated in other processes. The language provides a synchronization mechanism that can specify the execution order between parts of expressions in separate processes. 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 process abstractions can be used for sharing data asynchronously, which can provide better usability or performance in some instances. A process, also known as a thread in Concurrent ML, is created using the `spawn` primitive.

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

Processes communicate by having shared access to a common channel. A channel can be used to either send data or receive data. When a process sends on a channel, another process must receive on the same channel before the sending process can continue. Likewise, when a process receives on a channel, another process must send on the same channel before the receiving process 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 processes 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 [? CML book] 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 process 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`.

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 process is waiting or there's nothing waiting. The implementation doesn't need to consider states where competing processes are also trying to communicate on the same channel, since the course-grain interleaving ensures that competing processes have made no partial communication progress. In a multiprocessor setting, processes can run in parallel and multiple processes can simultaneously make partial progress in communication 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 processes making partial communication progress.

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 the cost of handling competing processes can be at least partially eliminated. Concurrent ML does not provide language features for multiple types of channels distinguished by their communication topologies, or the number of processes that may end up sending or receiving on it. However, we can classify channels 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. We demonstrate the topologies with a program that uses the server implementation defined in Listing 2.

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

Since there are four processes that make calls to the server, the server's particular `reqCh` has four senders. Servers are created with only one process listening for requests, so the `reqCh` of this server has just one sender. So the server's `reqCh` is classified as fan-in. Each use of `call` creates a brand 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 static analysis that describes communication topologies of channels has practical benefits in at least two ways. It can highlight which channels are candidates for optimized implementations of communication, or in a language extension allowing specification of restricted channels, it can conservatively verify the correct usage of restricted channels. We can modify the implementation in Listing 2 with a language extension.

```
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, we 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 static 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.

(* overview of data-flow analysis, flow graph, control-flow analysis in context of CML *)

(* example CML program with labels *)

(* introduce data-flow analysis for CML

- available expression (forward, largest)
- reaching sends (forward, smallest)
- very busy (backward, largest)
- live variable (backward, smallest)

*)

(* introduce flow graph *)

(* introduce control-flow analysis *)

(* enhance flow graph *)

(* enhance data-flow analysis*)

An efficient algorithmic analysis by Reppy and Xiao determines for each channel all processes 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 processes 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 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 process spawning determined directly from the primitives used in the program. Relying on information from the mappings to abstract (or approximate) values, it constructs the possible paths of execution via function application and channel communication. It uses the graph for live variable data-flow 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 process sends (receives) on a channel if the execution allows more than one to send (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 processes in terms of control paths, since it can only describe processes in terms of statically available information. Thus, in order to describe the relationship between the processes 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 process 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 our definitions or proofs. The manual process of checking is tedious and error prone. Mechanical checking of proofs can notify us of errors in our proofs or definitions far better and faster than manual checking. The proof assistant Isabelle is one tool that mechanically checks the correctness of proofs of propositions.

Although Isabelle is described as a proof assistant[?], 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 or translating code. Code as well as the logics for processing code are defined by users. 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 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`.[1] The constructors are defined with inductive definitions that declare a set of inference rules, where each inference rule defines the conditions necessary for a construction to be isomorphic to a valid proposition. 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. In Isabelle/HOL, propositions may be proved by applying the inference rules, working backwards from the proposition until no further conditions need to be satisfied.

```
datatype nat = O | S nat

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

theorem "sorted lte [O, (S O), (S O), (S (S (S O)))]"
 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
```

[1] Other systems[?] have chosen to treat propositions as types[?], and let types depend on terms[?].

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 ≡ (ΈP. P)"
```

**Hypothesis**

We will derive a static analysis from Reppy and Xiao's algorithm, describing for each channel in a program, all processes 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.  We 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.  We 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.  We 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 a static analysis derived from Reppy and Xiao's analysis.

**Architecture**

To enable mechanical verification of the correctness of our proofs, we will construct the semantics, analysis and theorems in the formal language of Isabelle/HOL.  To aid the development of formal proofs, we will design the analysis as a set of constraints as opposed to an algorithm.  However, the constraint-based analysis will make the proof of computability less direct.  To aid the scrutiny of our theorems' adequacy, we will express our 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.  We will not rely on intermediate map or graph data structures, which Reppy and Xiao used for efficient computation.  In order to relate our analysis to the operational semantics, we will borrow Reppy and Xiao's strategy of stepping between sets of control paths tied to terms.

In this thesis work, we are interested in communication topology soundness, rather than flow soundness. Nevertheless, we 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[?]. The semantics would be defined not by stepping from a term to a simpler term, but instead by stepping from a term to a new term with a context for looking up previous terms via their binding variables. By avoiding simplification of terms in the operational semantics, it will be possible to relate the abstract (approximate) values of an analysis to the values produced by the operational semantics, which in turn is relied on to prove flow soundness.

We 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[?]. Additionally, the restricted grammar melds nicely with the control path semantics. Instead of defining additional syntax for program points of primitive operations, we can simply use the required variables of the restricted grammar to identify program points, and control paths will simply be sequences of variables. A modification of Listing 2 illustrates the restrictive grammar applied to Concurrent ML.

```
let make = fun f () =>
  let reqCh = channel () in
  let loop = fun loop state =>
    let req = recv reqCh in
    let v = fst req in
    let replCh = snd req in
    let p1 = (reply, state) in
    let u1 = send p1 in
    let lres = loop v in
    lres
  let _ = spawn (
    let z = 0 in
    let sres = loop z in
    sres
  ) in
  let mres = S reqCh in
  mres

let call = fun f (server, v) =>
  let reqCh = case server of (S x) => x in
  let replCh = channel () in
  let p2 = (v, replCh) in
  let p3 = (reqCh, p2) in
  let u2 = send p3 in
```

```
            let cres = recv replCh in
            cres
```

**Deliverables**

- Technical paper
- Isabelle code

**References**

...

**Schedule**

- Define the syntax of Concurrent ML using Isabelle/HOL
- Define the operational semantics
- Define communication topologies in terms of operational semantics
- Construct proofs that simple programs adhere to communication topology definitions
- Define constraint-based analysis
- Construct proofs that analysis is informative with respect to example programs
- Construct proofs that the analysis is sound
- Develop an algorithm that takes a program and determines values described by the analysis
- Prove the analysis is computable using the algorithm.
- (Optional: Extend the definitions and proofs with `choose_evt`)
- (Optional: Extend the definitions and proofs with `then_evt`)
- Write technical paper
- Defense (Spring term 2018)