Master's Thesis Proposal (Draft – Oct 25, 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's and Xiao's static analysis algorithm for computing sound approximations of communication topologies[?]. We will define a small-step operational semantics for Concurrent ML that 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 atomic 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 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 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. A simple example derived from one used by Reppy and Xiao illustrates these essential features of Concurrent ML.

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

fun mkServer () = let
  val reqCh = channel()
  fun loop state = let
    val (v, replCh) = recv reqCh
    val x = state * v
  in
    send(replCh, x);
    loop (rem (state + 1, 7))
  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
```

The function `mkServer` creates 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. It computes a new value using the request's number `v` and the `state` then sends the new value back to the client through `replCh` by calling `send(replCh, x)`. It initiates the next iteration of the loop by calling `loop` with a new state. The server is started in 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 available or there's nothing waiting. The implementation doesn't need to consider states where other processes are also trying to communicate on the same channel, since the course-grain interleaving ensures that other 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 and sends data only once.

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. 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.

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. Using the control-flow graph, the algorithm then performs data-flow analysis in order to determine a mapping from program points to all possible control paths leading into the respective program points. The algorithm determines the program points for sends and receives per channel variable, using the mappings to abstract values. 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, fan-out, and so on.

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.

Hypothesis

We will derive a static analysis from Reppy's 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's 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 structures, which Reppy and Xiao used for efficient computation. In order to relate our analysis to the operational semantics, we will borrow Reppy's and Xiao's strategy of stepping between sets of control paths tied to terms.

In this thesis work, we are interested in communication soundness, rather than flow soundness. Nevertheless, it may be useful to prove additional flow soundness theorems both for debugging purposes, and to ensure adequacy of the theorems we prove successfully. 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 a language 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 the previous example illustrates the restrictive grammar applied to Concurrent ML.

```
let mkServer = 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 x = state * v in
    let p = (reply, x) in
    let u1 = send p in
    let s1 = 1 in
    let s2 = state + s1 in
    let s7 = 7 in
    let sp = (s2, s7) in
    let r = rem sp in
    let lres = loop r 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 pl = (v, replCh) in
  let rp = (reqCh, pl) in
  let u2 = send rp 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)