Masters Thesis Proposal (Draft - Oct 18, 2017)

Thomas Logan

The goal of this masters thesis is to develop formal propositions and proofs of communication topologies in Concurrent ML[?] that have practical benefits towards performance in multiprocessor environments. We will develop a constraint-based static analysis that describes all the communications possible with varying precision. We will also define a small-step operational semantics for Concurrent ML and prove that our analysis is sound with respect to the semantics. Our propositions, proofs, and theorems will use Isabelle/HOL[?] as the formal language of reasoning. The proofs will be mechanically checked using Isabelle[?].

Concurrent programming languages provide linguistic features that separate the placement of code from the execution order of code. The separation may be advantageous for a variety of reasons. Conceptually distinct tasks may need to overlap in time, but are easier to understand if they are kept separate. If it's not necessary for tasks to be ordered in a precise way, then it may be better that the program not specify the timing and instead allow a scheduler to find an execution order based on runtime conditions. A common use case for concurrent languages is GUI programming, in which the 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 that allows the placement of code to have no or little bearing on the execution order. The language provides a synchronization mechanism that can specify the execution order between non-adjacent code segments. 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. The uniform mechanism is safer[1] than two distinct mechanisms, since a program is unsound if it doesn't synchronize when necessary but still sound if it synchronizes when unnecessary. Additional process abstractions can be used for sharing data asynchronously, which can provide better usability or performance in some instances.

Processes communicate by having shared access to a common channel. A channel can be used to either send data or receive data. A given channel can have multiple processes sending data on it and multiple processes receiving data from it over the course of the program's execution. A many-to-many

1. The communication mechanism does not guarantee proper synchronization. It is still possible to write a deadlocked program. Perhaps a past/future static analysis can detect deadlock[?]

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 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.

A static analysis that describes communication patterns 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.

Our static analysis will describe 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.

We will ensure the correctness of our results by constructing our analysis, semantics, and theorems in the formal language of Isabelle/HOL, which will enable mechanical verification. To aid the development of formal proofs, we will design the analysis as constraint-based, in terms of axioms and inference rules, in contrast to recursive functions. Additionally, to aid the scrutiny of the adequacy of our definitions and propositions, we will express our definitions and propositions, with the fewest number of structures necessary. Thus, efficiency of computation will be ignored.

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 mappings from names or variables to abstract values, for functions, channels and other values. Relying on information from the mappings to abstract (or approximate) values, it constructs a control-flow graph that indicates the possible paths of execution via function application, channel communication, and process spawning. 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 there analysis by showing that their analysis claims 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 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 analysis developed for this thesis will be designed to aid formal proof rather than efficient computation. We will define our analysis as a set of constraints rather than as an algorithm. Additionally, we will not rely on intermediate map or graph structures to aid the efficiency of computation. Our analysis will be defined with the fewest judgements, inferences rules, and axioms necessary. In order to relate our analysis to the operational semantics, we will borrow the strategy of stepping between sets of control paths tied to terms.

A constraint-based control-flow analysis of System F by Fluet offers insight into how we might develop our analysis of communication topologies of Concurrent ML. The System F under analysis is restricted to a form that requires every abstraction and application to be bound to a variable. The restricted grammar allows the operational semantics to 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 is possible to relate the abstract (approximate) values of the control-flow analysis to the values produced by the operational semantics, which in turn is relied on to prove flow soundness. Fluet also demonstrates that the constraint-based control-flow analysis is computable via interpretation as a regular-tree grammar, which can be decided by an algorithm.

The communication topology analysis by Reppy and Xiao, and the control-flow analysis by Fluet both demonstrate the importance of maintaining static information through the steps in operational semantics for the purpose of proving soundness. In this thesis work, we are interested in communication soundness, rather than flow soundness. Nevertheless, it may be useful to keep the static term information around and prove additional soundness theorems for debugging purposes, and to ensure adequacy of the theorems we prove successfully. Thus, we will incorporate the restricted grammar and the environment based semantics into this work. Additionally, the restricted grammar melds nicely with the control path semantics from Reppy and Xiao. 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.

****Logics in proof assistants by Paulson, Harrison, Gordon, Klein, Leroy, Chlipala etc****

- competing systems:
  - dependent vs simple
  - CoC vs HOL
  - proofs as terms vs proofs as meta-terms
  - props/sets as types vs props/sets as terms
  - types as terms vs types distinct from terms
  - programming with dependent types:
    - http://adam.chlipala.net/cpdt/cpdt.pdf
  - simple theory of types:
    - http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-175.pdf

****How this work relates to logics in proof assistants****

****Future work****

1. extend analysis to include events and choose combinator
2. extend analysis to include sequence (TE) and pair (reagents) combinators.
3. Implement topology recognition algorithm in MLton.
4. Implement parallel processes in MLton.
5. Implement specialized communication in MLton.
6. Collect and analyze empirical data on communication overhead in MLton.
7. Develop algorithm that dynamically finds optimal grouping of processes on available processors?