Independent Study: Mechanized Theorem Proving

Matthew Le

May 20, 2014

1 Introduction

This independent study focuses on an effort to formalize my current research on speculative parallelism and shared state using a mechanized proof assistant. More specifically, my research is currently focussed on how to guarantee a deterministic semantics in the presence of IVars [ANP89] and cancellation. In this independent study I formalized the semantics of a parallel functional language that has been extended with speculation and IVars using the Coq Theorem Prover and prove a number of supporting lemmas that are used in proving an equivalence to a non-speculative version of the same language.

2 Representation

Perhaps the most important component of this project was choosing a representation for expressing these semantics. In the pen and paper version of the semantics, each rule takes the form:
$H;\ T \rightarrow H';\ T'$, where H and T are metavariables ranging over heaps and thread pools respectively. A heap maps names of IVars to IVar contents, where the contents of an IVar can be full or empty. If an IVar is full, then we also store the thread IDs of threads who read from the IVar, whether or not it is speculatively full, the thread ID of the writer, and the contents of the IVar. The heap is simply represented as a list of pairs in Coq, where the first element of each pair is an integer, and the second element is of type IVar_contents, which is an inductively defined data type with two constructors: full and empty.

The question of how to represent the thread pool was much more challenging. My initial approach was to define it as:

```
Definition thread := tid * actionStack * term.
Inductive pool : Type :=
  |par : pool -> pool -> pool
  |single : thread -> pool
  |empty : pool.
```

Figure 1: Thread Pool Representation: Failed Attempt

The problem with this representation is that in our operational semantics, we need to include rules that explicitly rearrange the thread pool so that any arbitrary thread is able to take a step at a given point in time. Instead, I have chosen to represent the thread pool as an unordered set of threads using Ensembles from the Coq standard library, where threads are defined as they are in Figure 1. This approach was inspired by the Coq development presented in [EDKG08]. With

this representation we are able to define our operational semantics without having to worry about including congruence rules for rearranging the thread pool.

The next design choice came up in the context of variable binding. I had originally represented lambda terms as an integer denoting a variable and an expression corresponding to the body of the lambda. I soon ran into problems with proving that two lambda terms where equivalent due to the fact that at the time of proving the equivalence, you only know that the variables are two integers, however, there is no way to prove that they are the same integer. Instead, I switched to the locally nameless representation presented in [ACP+08]. With this method, we represent variables bound in lambda terms using De Bruijn indices. The idea is that the integer used to represent a bound variable is the nesting depth of lambdas that it is inside of. With this, proving equivalence of lambda terms that are alpha equivalent is trivial.

3 Automation

Many of the theorems that I chose to formalize using Coq where chosen specifically because they require exhaustive case analysis. This sort of reasoning can be tedious and sometimes unmanageable in a pen and paper setting, however, with a proof assistant we can make use of automation to make this easier. Some of these theorems contain a large number of vacuous cases that need to be discharged, which can be done using automation. Typically this amounts to finding a contradictory hypothesis and applying inversion to it. The problem is that if we explicitly name the hypothesis in our tactic, we run into the problem of writing proof scripts that are not very robust. If we modify the statement of the theorem, or any of the definitions of inductive types used in the statement of the theorem, then the names of the generated hypotheses change. I have gotten around this issue by using Coq's powerful pattern matching features. This allows us to specify the form of the hypothesis that we are trying to apply inversion to, yet allows us to leave many of the specific details abstract by using unification variables.

Automation was also heavily used with respect to much of the reasoning used towards simple facts about sets. By representing sets as functions, as they are in the Ensembles library, proving seemingly trivial facts can be quite time consuming and tedious. For example, in order to prove the obvious theorem that a set containing one element is not equal to the empty set, rather than simply performing inversion, we must do the following:

```
Theorem SingletonNeqEmpty : forall T e, Singleton T e <> Empty_set T.
Proof.
  unfold not. intros. apply eqImpliesSameSet in H. inversion H.
  unfold Included in *. assert(I:In T (Singleton T e) e).
  constructor. apply I in H2. inversion H2.
Qed.
```

Figure 2: Proof that the singleton set is not equal to the empty set

Clearly, this is more work than we would like to do each time that we want to show that the singleton set is not equal to the empty set. To alleviate some of this pain, I have implemented a small library that will automatically match hypotheses corresponding to many of these simple facts about sets and dispatch to the appropriate theorem.

4 Unresolved Issues

Given the amount of time, I was not able to prove all of determinism, but only a subset of the supporting lemmas used in the pen and paper proof. One of the big problems that I ran into was an issue of monotonicity that shows up in a number of theorems regarding multistep derivations. As an example, Theorem 1 turned out to be a surprisingly difficult theorem to prove.

Theorem 1 (Action Preservation) $\forall H\ T\ T'\ T''$, *if* $H;\ T \to H;\ T' \to^* H;\ T''$ *and* $\mathcal{SA}[\![T]\!] = \mathcal{SA}[\![T'']\!]$ *and* $\mathcal{CA}[\![T]\!] = \mathcal{CA}[\![T'']\!]$, *then* $\mathcal{SA}[\![T]\!] = \mathcal{SA}[\![T']\!]$ *and* $\mathcal{CA}[\![T]\!] = \mathcal{CA}[\![T']\!]$

In attempting to prove this theorem, we need to consider all possible rules that $H;\ T \to H;\ T'$ could potentially correspond to. In the cases where it is a pure step, this goes through easily by invoking the induction hypothesis, however, for the cases corresponding to impure steps we have no way of showing that they are vacuous. In order to get this to go through, we would need some sort infrastructure for expressing that the sum of speculative and commit actions is strictly monotonically increasing in some fashion. Certainly this is not an impossible task, but time did not permit this exploration.

5 Conclusion

Making use of a mechanized proof assistant has personally benefited my research in two major ways. First, it has allowed me to be comprehensive about proving theorems that require tedious and exhaustive case analysis by making use of proof automation. Second, it provides additional confidence that the theorems presented in my work are actually correct. Formal verification is something that has grown to be increasingly popular in the programming languages research community, and I believe that taking part in this independent study has given me the tools and insights necessary to join that community.

References

[ACP+08] Aydemir, B., A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL'08*, San Francisco, California, USA, 2008. ACM, pp. 3–15.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, **11**(4), October 1989, pp. 598–632.

[EDKG08] Effinger-Dean, L., M. Kehrt, and D. Grossman. Transactional events for ml. In *ICFP'08*, Victoria, BC, Canada, 2008. ACM, pp. 103–114.