Distributed IMP

Robin Kelby and Alex Snyder

Ohio University, Athens, OH 45701

1 Introduction

We were first introduced to Imp from our textbook Software Foundations by Pierce et al. Imp is a small imperative programming language. It contains a small core portion of major imperative programming languages like C and Java(Cite Textbook). The textbook introduces Imp to demonstrate how to model a programming language in Coq. Then it could introduce different ways to reason about programs in Coq. These included program equivalence and Hoare Logic. In one chapter they introduce a extension to Imp called concurrent Imp. This is an extension that models an Imp program running commands in parallel.

1.1 Concurrent Imp

In concurrent Imp the textbook adds an extension to Imp to model two command running in parallel. This is a simple task because they only need to add a new command to the inductive type com, notation for the command and lastly they add three new step rules to the inductive type cstep.

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  (* New: *)
  | CPar : com → com → com.

Inductive cstep : (com * state) → (com * state) → Prop :=
(*New Part*)
 | CS_Par1 : st c1 c1' c2 st',
      c1 / st ⇒ c1' / st' →
      (PAR c1 WITH c2 END) / st ⇒ (PAR c1' WITH c2 END) / st'
  | CS_Par2 : st c1 c2 c2' st',
      c2 / st ⇒ c2' / st' →
      (PAR c1 WITH c2 END) / st ⇒ (PAR c1 WITH c2' END) / st'
  | CS_ParDone : st,
      (PAR SKIP WITH SKIP END) / st ⇒ SKIP / st
```

The first two steps give the choice of either c1 command stepping or c2 command stepping. Unlike previous commands that were made to be deterministic these two commands are non-deterministic. This is how the parallel computation is modeled. Both c1 and c2 can step and one does not have to be a value before the other one can step. And, then to allow the program to stop, the last rule is added, for when both c1 and c2 are skip, then the whole command steps to skip.

1.2 Distributed IMP

The extension concurrent Imp gave us the idea to make our own extension to Imp. We decided on extending the model of Imp to include distributed computing. Imp was modeled as one program running commands. We wanted to extend it so it had some representation of different machines running those commands and the ability to send and receive data between those different machines. We updated the state with buffers to hold the data sent and received. We modeled machines as inductive data types and defined step relations for these different machines. The rules were non-deterministic to model the parallel nature of distributed computing. The programs can themselves step or send data to another machine. We then tested basic programs to determine if they worked correctly.

1.3 Rationale

While distributed computing is not new, it is important and relevant for many application today. In fact, distributed computing comes from the 1960s operating system architectures. That is an operating system with concurrent processes communicated by message passing(Cite this). While automatic theorem provers like Coq are relatively new and exciting area of computer science, it is unlikely that much has been done yet to model distributed systems with them. Distributed systems are widely used and therefore it would be useful to model them and be able to prove things about them.

2 Technical Development

Since we already had a model in Coq for Imp that is of one program running, the first step was to give that model some ability to send and receive data. The end goal was to define an overall inductive relation that would non-deterministically choose a machine and make one step of the program or send information between the machines. To get to that point, we needed to update the state by giving it a send and receive buffer. That way when a send command is called in the Imp program it could update its send buffer and when a receive command is called it could get the data from its receive buffer. The overall inductive definition could then take from the one machine's send buffer and put the data in another machines receive buffer.

2.1 Distributed on a Single Imp Program

To update the state, we first defined a triple type to make the definition of the send buffer easier. Then the send buffer is a triple with the (aexp ∗ id ∗ id) where the aexp is the data and one id is the machine id and the other is the variable id. The receive buffer is just the data and variable id and st is the regular state of an Imp program. We then put all these together in a inductive type called State. Lastly, we defined the empty state with the added nil and nil for the send and receive buffer respectively.

```
Inductive triple (A B C : Type) : Type :=
  | trip : A → B → C → triple A B C.

Notation "x '∗' y '∗' z" := (triple x y z)
  (at level 70, right associativity).

Definition sb := list (aexp ∗ id ∗ id).

Definition rb := list (aexp ∗ id).

Definition st := total_map nat.

Inductive State : Type :=
| state : sb → rb → st → State.

Definition empty_state : State := state nil nil (t_empty 0).
```

We updated the inductive com definition with the two new commands send and receive. The send command taking an aexp and an id to identify the machine and a variable id. The receive command does not take any arguments since it just modifies the receive buffer. Then we gave them notation like the other Imp commands.

```
Inductive com : Type :=
  (∗ Distributed Commands ∗)
  | CSend : aexp → id → id → com
  | CRecieve: com.

  Notation "'SEND' a 'TO' id1 'CALLED' id2" :=
  (CSend a id1 id2) (at level 80, right associativity).
  Notation "'RECEIVE'" :=
  (CRecieve) (at level 80, right associativity).
```

We then added four new constructors to the cstep inductive definition. We defined a send constructor for which the aexp could step and then another for when that aexp was a number and then it could add it to the send buffer. We defined two receive constructors. The first was for the case when there is nothing in the receive buffer. Then the program should step to a SKIP command

sequenced with another receive command. The second receive constructor would take the first element in the receive buffer and update the state with the new binding of variable and number.

```
Inductive cstep : (com * State) → (com * State) → Prop :=
(* Distribued Steps *)
 | CS_Send1 : forall (sb1 : sb) (rb1 : rb)
                     (st1 : st) (a a' : aexp) (x z : id),
      a / state sb1 rb1 st1 ⇒ a a' →
      cstep (SEND a TO x CALLED z, state sb1 rb1 st1)
            (SEND a' TO x CALLED z, state sb1 rb1 st1)
  | CS_Send2 : forall (sb1 : sb) (rb1 : rb) (st1 : st)
                     (a : aexp) (x z : id) (n : nat),
      a = ANum n →
      cstep (SEND a TO x CALLED z, state sb1 rb1 st1)
            (SKIP, state (app sb1 (cons (a, x, z) nil)) rb1 st1)
  | CS_Rec1 : forall (sb1 : sb) (st1 : st),
      cstep (RECEIVE, state sb1 nil st1)
            (SKIP ;; RECEIVE, state sb1 nil st1)
  | CS_Rec2 : forall (sb1 : sb) (rb1 : rb)
                     (st1 : st) (a : aexp) (z : id),
      cstep (RECEIVE, state sb1 (app (cons (a, z) nil) rb1) st1)
            (z ::= a, state sb1 rb1 st1).
```

2.2 Modeling Multiple Imp Programs

```
Inductive imp : Type :=
  | machine : id → com → State → imp.

Inductive dist_imp : (imp * imp) → (imp * imp) → Prop :=
  | imp_step_1 : forall (c1 c1' c2 : com) (st1 st1' st2 : State) (x y : id),
    cstep (c1, st1) (c1', st1') →
    dist_imp ((machine x c1 st1), (machine y c2 st2))
             ((machine x c1' st1'), (machine y c2 st2))
  | imp_step_2 : forall (c1 c2' c2 : com) (st1 st2' st2 : State) (x y : id),
    cstep (c2, st2) (c2', st2') →
    dist_imp ((machine x c1 st1), (machine y c2 st2))
             ((machine x c1 st1), (machine y c2' st2'))
  | send_y : forall (c1 c2 : com) (sb1 sb2 : sb) (rb1 rb2 : rb) (st1 st2 : st)
                    (a : aexp) (x y z : id),
    dist_imp ((machine x c1 (state (cons (a, y, z) sb1) rb1 st1)),
              (machine y c2 (state sb2 rb2 st2)))
             ((machine x c1 (state sb1 rb1 st1)),
             ((machine y c2 (state sb2 (app (cons (a, z) nil) rb2) st2))))
  | send_x : forall (c1 c2 : com) (sb1 sb2 : sb) (rb1 rb2 : rb) (st1 st2 : st)
```

```
                    (a : aexp) (x y z : id),
  dist_imp ((machine x c1 (state sb1 rb1 st1)),
            (machine y c2 (state (cons (a, x, z) sb2) rb2 st2)))
           ((machine x c1 (state sb1 (app (cons (a, z) nil) rb1) st1)),
           (machine y c2 (state sb2 rb2 st2))).

Definition cdist_imp := multi dist_imp.

Lemma proof_of_concept : forall x y n z,
  multi dist_imp ((machine x (SEND (ANum n) TO y CALLED z) empty_state),
                  (machine y (RECEIVE) empty_state))
                  ((machine x SKIP empty_state),
                  (machine y (z ::= (ANum n)) empty_state)).
Proof.
  intros. eapply multi_step. apply imp_step_1.
    eapply CS_Send2. reflexivity.
  eapply multi_step. apply send_y. fold empty_state.
  eapply multi_step. apply imp_step_2. apply CS_Rec2. fold empty_state.
  eapply multi_refl.
Qed.
```

3 Conclusion

We modeled distributed computing for the Imp language and demonstrated that some basic examples worked correctly. While this is a good start, it is still fairly limited in scope. For one thing, are model only has two machines while most distributed systems have many different machines. Another problem with our model is that it only allows a program to send a number to the other machine. It does not allow it to send any other data type like bool. On top of that, it sends data with a predefined variable name. This could be problematic if the other Imp machine already has a definition for that variable.

Going forward the next step would be to model a system with more than two computers. This would be more like a real world distributed computing and less restricting. We could also add the ability for the programs to be able to send different types of data other than just numbers like it does now. And, the variable with the data type that is sent could be removed. It was originally added because the model will not work without it. It needs to be modified in such a way that we can remove this restriction.

When the model is improved then there are many steps forward for this project. We can define larger distributed examples and see how they work within or model. We could define problems that are unique to distributed computing and how to determine if they exist within a program. We could define and prove things that are unique to distributed computing in general. And, finally the end goal would be to model real world distributed programs and prove things about them.

4 Reference Stuff

This is a citation. [1]

References

1. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482. Springer-Verlag, 2010.