Distributed Imp

Robin Kelby and Alex Snyder

Ohio University, Athens, OH 45701

1 Introduction

Our first introduction to Imp came 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 [2]. The textbook introduces Imp to demonstrate how to model a programming language in Coq, and the following chapters outline different ways to reason about programs in Coq. These methods included program equivalence and Hoare Logic. In a chapter called Smallstep, which covered small-step operational semantics for programming languages, the authors introduce a extension to Imp called Concurrent Imp. This extension models an Imp program running commands in parallel. We were able to use our experience with Imp, along with the methodology of Concurrent Imp, to create our Distributed Imp.

1.1 Concurrent Imp

In Concurrent Imp, the textbook adds an extension to Imp to model two commands running in parallel. This is a simple task because the authors only needed to add a new command, CPar, to the inductive type com, notation for Par, and lastly 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 : forall st c1 c1' c2 st',
      c1 / st ⇒ c1' / st' →
      (PAR c1 WITH c2 END) / st ⇒ (PAR c1' WITH c2 END) / st'
  | CS_Par2 : forall st c1 c2 c2' st',
      c2 / st ⇒ c2' / st' →
      (PAR c1 WITH c2 END) / st ⇒ (PAR c1 WITH c2' END) / st'
  | CS_ParDone : forall st, (PAR SKIP WITH SKIP END) / st ⇒ SKIP / st
```

The first two rules allow either the c1 or c2 command to step to its next state. Unlike previous commands that were forced through inference rules to be deterministic, these two commands are non-deterministic. This is how the parallel computation is modelled. 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 extensions in Concurrent Imp gave us the idea to make our own extension to the Imp model. We decided to make new rules to support distributed computing. Imp was modelled as one program running commands, so we wanted to extend it so it can have 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 modelled 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 applications today. In fact, distributed computing comes from the 1960s operating system architectures, an operating system with concurrent processes communicated by message passing [1]. While automatic theorem provers like Coq are relatively new and an exciting area of computer science, it is unlikely many people have modelled 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 for the execution of one program, 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. The send buffer is a triple with the (aexp ∗ id ∗ id) where the aexp is the data, the first id is the machine id, and the other is the variable id. The receive buffer is simpler, just the aexp data and variable id. The final part of the state, st, is the original implementation of state for an Imp program, which is a total map of natural numbers. We then packaged these three together in a inductive type called State. Lastly, we defined a new empty state for our implementation, 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).
```

Next, we updated the inductive com definition with two new commands: send and receive. The send command takes as arguments an aexp, 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. 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 is not a value and could therefore step, and another for when the aexp is a value and it is added to the send buffer.

We defined two receive constructors. The first receive was for the case when the receive buffer is empty; the program steps to a SKIP command sequenced with another receive command. The second receive constructor takes the first element in the receive buffer and updates 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

At this point, our implementation differs from that of Concurrent Imp in several ways. Because the parallel commands are occurring on one machine, the two parallel commands share the same state. When one command changes the state, the other command will be affected by that change. In contrast, our distributed system required something quite different. Each machine in our distributed system requires three things: an id which functions as its name, a set of Imp commands, and a State. We used an inductive type called imp to package those three types together.

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

Furthermore, we needed to define an inductive step relation over imp, as the cstep relation could only step Imp commands, not the entire distributed system. We limited our distributed system to two machines for simplicity. Our dist_imp relation has four rules, two for when each machine can execute a command and take a step, and two for when one machine sends data to the other. The first two rules state that the dist_imp relation will take a step when one machine can take a step. These rules make our implementation non-deterministic, because our relation does not force one machine to step as far as it can before the other machine can step. The last two rules change the states of both machines, removing the data and id from one machine's send buffer and updating the receive buffer of the other machine.

```
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))).
```

Finally, we needed a way to test whether our implementation correctly models distributed commands. We defined a multi-step relation on dist_imp, and then used that multistep to prove that a set of commands in initial states, in our case the empty state, step to the correct commands in the final states.

```
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.
    (*Uses the cstep relation to update
           machine x's send buffer with data and id's*)
  eapply multi_step. apply send_y. fold empty_state.
    (*Uses the send_y rule to move the data from
            machine x's send buffer to the receive buffer of machine y*)
  eapply multi_step. apply imp_step_2. apply CS_Rec2. fold empty_state.
    (*Uses the cstep relation to remove the data from
            machine y's receive buffer and assign it to an id*)
  eapply multi_refl.
Qed.
```

3 Conclusion

We modelled 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.

References

1. Gregory Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.
2. Benjamin Pierce, C, Arthur Azevedo de Amorim, Chris Casinghino, and et al. *Software Foundations.* 2016.