1 Introduction

Formal verification is the process of verifying the correctness of a computer program, using the logical reasoning techniques of pure mathematics. As we depend upon software, built up from programs, in more and more situations in daily life, it becomes ever more important that this software be reliable and trustworthy. In the most crucial applications, it is not enough to be satisfied that some human programmer has visually stepped through the code and determined it to be bug-free. For most significant projects today, which consist of thousands, if not millions of lines of code, such a claim is outright impossible. The solution is to use computers themselves to aid in the systematic verification of a program.

More specifically, interest in formal verification has motivated the development of proof assistants, a term used to describe a class of software which aids in proof development. Proofs assistants provide an environment in which proof writers may manage hypotheses and claims, an interface between real written code and the logical constructions necessary for reasoning about it, and, most importantly, the ability to check the validity of a proof according to the assistant’s internal logic engine. Perhaps the best known proof assistant is Coq, which was developed starting in 1984 by Thierry Coquand and Gerard Huet of inria. Coq’s applications range from the verification of real-world software such as CompCert or CertiKos, to being a computational aid in proving famous mathematical theorems such as the 4-color theorem or Feit-Thompson theorem.

This semester, I have been learning of the basics of formal verification using Coq, through Software Foundations by Pierce et al. In this report, I give a summary of what I have learned. I begin with an overview of the basic logical ideas underlying Coq, with an emphasis towards the powerful reasoning derivable from inductively-defined objects. Next, I explain how Coq may be harnessed to reason about programs, using a tiny toy language, Imp, to model general principles found in many languages: variables, assignment, conditionals, and loops. Finally, I give a brief description of formal verification in CertiKos, a verified operating system developed by Yale’s FLINT research group, directed by Zhong Shao, who is also the supervisor of my project.

The examples that I discuss in the next two sections all are exercises from Software Foundations (abbreviated as SF). The actual proofs, unless noted, are my own. I will usually give an informal proof whose structure is parallel to the Coq proof. However, it may also be illuminating to follow the proofs “live” in the user interface provided by Coq itself, so I have also included the Coq files in the same directory as this report.

2 Basics of Logic in Coq

2.1 Inductively Defined Types

The first important concept to understand in Coq is that of an inductively defined type. For example, Coq defines the datatype of natural numbers inductively as follows:

\[ \text{Inductive nat : Type :=} \]
\[ | \ O : \ nat \]
\[ | \ S : \ nat \rightarrow \ nat. \]
Intuitively, this says that a natural number is either 0 (corresponding to $0$), or the successor to (i.e. one more than) another natural number (corresponding to $S : \text{nat} \to \text{nat}$). More formally, the body of an \textbf{Inductive} definition for \texttt{nat} consists of a list of constructors, each of which is a function that takes some input (which could be anything, including none or another \texttt{nat}) and returns a \texttt{nat}.

By itself, this definition does not specify all of the structure of \texttt{nat}'s, but that structure too may be constructed, by defining functions on and proving statements about \texttt{nat}'s. Through function definition, we observe one of the first conveniences afforded by inductively defined types. As Coq is a functional language, one may simply pattern match according to the various constructors. For example, we can define addition:

\begin{verbatim}
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O  ⇒ m
  | S n' ⇒ S (plus n' m)
  end.
end.
\end{verbatim}

and equality:

\begin{verbatim}
Fixpoint beq_nat (n m : nat) : bool :=
  match n with
  | O  ⇒ match m with
    | O  ⇒ true
    | S m' ⇒ false
  end
  | S n' ⇒ match m with
    | O  ⇒ false
    | S m' ⇒ beq_nat n' m'
  end
end.
\end{verbatim}

We may also reason about constructors to do casework when proving statements. In the following example, taken from \texttt{SF}, \texttt{destruct} is used to independently consider $n$ as either $0$ or $S \ n'$, for some $n'$. Though the proof follows immediately in each case, progress cannot be made without considering each constructor separately.

\begin{verbatim}
Theorem plus_1_neq_0 : forall n : nat, beq_nat (n + 1) 0 = false.
Proof.
  intros n. destruct n as [\[ n'].
  - reflexivity.
  - reflexivity. Qed.
\end{verbatim}

Note that \texttt{destruct} is an example of a \textit{tactic}, the name given to Coq’s implementation of some common method of reasoning in formal logic. For example, the \texttt{destruct} tactic can be thought of as Coq’s equivalent to reasoning by casework.

Constructors not only specify the forms that any instance of an inductively defined type may take, but also specify that for any particular instance, there is exactly one way in which it is constructed. That is, each individual constructor is injective, and distinct constructors
map to disjoint sets. The tactic which implements this is known as \textit{inversion}. Here’s a simple example, again copied from \textit{SF}

\textbf{Theorem} \texttt{S\_injective} :
\texttt{forall (n m : nat),}
\texttt{S n = S m \rightarrow n = m.}

\textbf{Proof}.
\texttt{intros n m H.}
\texttt{(* H says \[S n = S m\] *)}
\texttt{inversion H. reflexivity. Qed}

Inversion also allows for proof of vacuously true statements. Recall that an example of such a statement is something like, “If 1 = 0, then pigs can fly”. Even though pigs cannot fly, this statement is logically valid, because the hypothesis itself is false. In Coq, the false hypothesis would read as \texttt{S O = O}. By performing an \texttt{inversion} on this hypothesis, Coq would see that different constructors (namely \texttt{S} and \texttt{O}) are producing the same value, which violates its internal logic. Thus, Coq recognizes the false hypothesis, and allows the proof of any conclusion to follow. Based on similar methods involving inversion, general proof by contradiction is also possible.

The final tactic to discuss is \textit{induction}. As its name suggests, \textit{induction} is made to work with inductively defined datatypes. The best known form of induction, typically seen in an introductory discrete math class, is on the natural numbers. It goes something like this: to prove that some claim \( P \) holds for all natural numbers \( n \), it suffices to show that

\begin{itemize}
  \item \( P \) holds for 0, and that
  \item \( P \) being true for some natural number \( k \) implies that \( P \) holds for \( k + 1 \).
\end{itemize}

We call this induction on \( n \). Note that this exactly parallels the inductive definition of \texttt{nat}. The method also looks similar to proof by casework on \( n \), except in the \( n = S n’ \) case, we get an additional hypothesis that the claim we want to prove holds for \( n’ \).

In general, if we use \texttt{induction} on some inductively defined variable \( v \) of type \( t \), it destructs \( v \) according to its constructors, and for each constructor argument \( v’ \) that is also of type \( t \), it generates the additional hypothesis that the desired claim already holds for \( v’ \), which may be used to make progress in the overall proof.

While it is easy to see that this general description of induction matches induction on natural numbers, it is more interesting to see how the tactic works on another inductively defined type. A good candidate will be lists:

\begin{verbatim}
Inductive list (X:Type) : Type :=
| nil : list X
| cons : X \rightarrow list X \rightarrow list X.
\end{verbatim}

For any existing type \( X \), we can define the inductive type \texttt{list X}, which has constructors \texttt{nil} corresponding to the empty list and \texttt{cons} which takes an element \( x \) of type \( X \) and an existing list \( l’ \) to form a new list \texttt{cons x l’}, with \( x \) as its first element and the elements of \( l’ \) trailing.

A useful function to perform on lists is reversal. This can be implemented naively as follows:
Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil   ⇒ nil
  | cons h t ⇒ app (rev t) (cons h nil)
  end.

As the function app has linear running time and is called a linear number of times, the implementation rev has quadratic running time.

Reverse can also be implemented tail-recursively, as follows:

Fixpoint rev_append {X} (l1 l2 : list X) : list X :=
  match l1 with
  | []   ⇒ l2
  | x :: l1' ⇒ rev_append l1' (x :: l2)
  end.

Definition tr_rev {X} (l : list X) : list X :=
  rev_append l []

rev_append runs in constant time, and tr_rev calls it a linear number of times, so it has linear running time. The implementation, though, is a little less transparent. Thus, if we could show that tr_rev and rev are indeed identical, then we can rest assured that our faster implementation is also correct. We do this by induction.

Theorem 1 rev and tr_rev are identical as functions on lists.

Proof We begin by proving a lemma.

Lemma 2 For all lists l1 and l2, rev_append l1 l2 = rev_append l1 [] ++ l2.

Proof We induct on l1.

In the first case, l1 = []. Then, we must show that rev_append [] l2 = rev_append [] [] ++ l2. This follows immediately from the definitions of rev_append and app.

In the second case, l1 = x1 :: l1' and we additionally have the induction hypothesis that rev_append l1' l2 = rev_append l1' [] ++ l2 for all lists l2. Our goal is to show that rev_append (x1 :: l2) 12 = rev_append (x1 :: l2) [], for all lists l2.

Using definitions, we may simplify our goal to the form rev_append l1' (x1 :: l2) = rev_append l1' [x1] ++ l2, for all lists l2. If we instantiate the induction hypothesis with l2 = [x1], we may apply it to the right hand side of our goal, to get rev_append l1' (x1 :: l2) = (rev_append l1' [] ++ [x1]) ++ l2. Using associativity of app and simplification, our goal becomes rev_append l1' (x1 :: l2) = rev_append l1' [] ++ x1 :: l2. Now we apply the induction hypothesis again, this time with l2= [], and our goals turns into an identity, so we are done.

Now that the lemma is proven, we first recall that we’ve defined two functions to be identical so long as they behave the identically on all their inputs. Thus, we want to show that rev l = tr_rev l for all lists l.
We induct on \( l \). The case where \( l = \[] \) follows trivially from definitions.

Now suppose \( l = x :: l' \), and our induction hypothesis tells us that \( \text{tr}_\text{rev} \ l' = \text{rev} \ l' \). We wish to show that \( \text{tr}_\text{rev} \ (x :: l') = \text{rev} \ (x :: l') \).

Unfolding the definition of \( \text{tr}_\text{rev} \) and simplifying, our induction hypothesis tells us that \( \text{rev} \ \text{append} \ l' \ [] = \text{rev} \ l' \) and our goal becomes \( \text{rev} \ \text{append} \ l' \ [x] = \text{rev} \ l' \ ++ \ [x] \). Using the induction hypothesis, we may substitute for \( \text{rev} \ l' \) in our goal, so now our goal is \( \text{rev} \ \text{append} \ l' \ [x] = \text{rev} \ \text{append} \ l' \ [] \ ++ \ [x] \). Note that this statement is exactly a case of our lemma above, with \( l1 \) instantiated as \( l' \) and \( l2 \) instantiated as \( [x] \).

Thus, by our lemma, we are done.

Other than the uses of induction, there are a few other interesting things to note. The first is that in the proof of our lemma, we use the inductive hypothesis twice, once with \( l2 = [x1] \) and once with \( l2 = \[] \). We are able to do this, because when we induct on \( l1 \), we leave the “for all lists \( l2 \)” intact, so we get an inductive hypothesis that quantifies over all \( l2 \). In Coq, you do this by only specifying \( l1 \) and not \( l2 \), i.e. doing \text{intros} \( l1 \) and not doing \text{intros} \( l2 \).

The second is that we simply claim two functions are identical if their outputs are always identical. Unlike the claim that \text{app} is associative, which was proven, this claim is an axiom. In Coq, \text{functional_extensionality} is just the axiom for doing this.

The analagous proof in Coq reads:

\[ \text{Lemma rev_append_fact: forall } X : \text{Type} \ (l1 l2: \text{list X}), \]
\[ \text{rev_append} \ l1 \ l2 = \text{rev_append} \ l1 \ [] \ ++ \ l2. \]
\[ \text{Proof.} \]
\[ \text{intros} \ X \ l1. \ \text{induction} \ l1 \ \text{as} \ [|x1 l1' IH]. \]
\[ \text{− reflexivity.} \]
\[ \text{− intros} \ l2. \ \text{simpl. rewrite} \ ➞ \ (IH [x1]). \ \text{rewrite} \ ➞ \ \text{app_assoc}. \]
\[ \text{simpl. rewrite} \ ➞ \ IH. \ \text{reflexivity.} \]
\[ \text{Qed.} \]

\[ \text{Theorem tr_rev_correct : forall } X, \ \cdot \text{tr}_\text{rev} \ X = \cdot \text{rev} \ X. \]
\[ \text{Proof.} \]
\[ \text{intros} \ X. \ \text{apply} \ \text{functional_extensionality}. \]
\[ \text{intros} \ l. \ \text{induction} \ l \ \text{as} \ [|x1 l' IH]. \]
\[ \text{− reflexivity.} \]
\[ \text{− unfold} \ \text{tr}_\text{rev}. \ \text{simpl. unfold} \ \text{tr}_\text{rev} \ \text{in} \ IH. \]
\[ \text{rewrite} \ ➞ \ IH. \ \text{apply} \ \text{rev_append_fact}. \]
\[ \text{Qed.} \]

### 2.2 Inductively Defined Propositions

Types are not the only things that can be defined inductively in Coq. Propositions (i.e. claims that can be proven) can also be defined inductively, and when done in this way, come with much of the same reasoning power that inductively defined types come with.

Here is a simple example of how property that natural number is even can be inductively defined:
**Inductive** \( \text{ev} : \text{nat} \rightarrow \text{Prop} := \)

\[ \begin{align*}
| \text{ev}_0 & : \text{ev} 0 \\
| \text{ev}_{\text{SS}} & : \forall n : \text{nat}, \text{ev} n \rightarrow \text{ev} (S (S n)).
\end{align*} \]

It can be illuminating to consider the parallels between this definition and the original inductive definition of natural numbers:

**Inductive** \( \text{nat} : \text{Type} := \)

\[ \begin{align*}
| \text{O} & : \text{nat} \\
| S & : \text{nat} \rightarrow \text{nat}.
\end{align*} \]

We defined \( \text{nat} \) to be a type, with constructors \( \text{O} \) and \( S \) returning instances of that type. Now, we define \( \text{ev} \) to be a function from a natural number to a proposition, with constructors \( \text{ev}_0 \) and \( \text{ev}_{\text{SS}} \) returning instances of such propositions (i.e. expressions of the form \( \text{ev} n \) for some \( n \)).

Just as the inductive definition of natural numbers tells us that a natural number is either constructed as \( \text{O} \) or \( S n \) for some other natural number \( n \), the inductive definition of \( \text{even} \) tells us that the proposition \( \text{ev} n \) may be constructed either as \( \text{ev} 0 \) or \( \text{ev}_{\text{SS}} n \) \( (\text{ev} n) \), where \( n \) is a natural number and \( \text{ev} n \) has already been constructed. The arguments given to any constructor of a proposition can be thought of as “evidence” for the resulting returned proposition.

The tactics that we encountered for reasoning about inductively defined types naturally extend to inductively defined propositions:

- When Coq destructs an inductively defined proposition (that has been given as a hypothesis), it generates a new proof context for each constructor, which now contains the relevant evidence that would have constructed the destructed proposition.

- When Coq inverts an inductively defined proposition, it acts like destruct, but anytime it encounters a contradictory constructor, it does not consider that case at all.

- When Coq inducts on an inductively defined proposition, it acts like destruct, but anytime it encounters an argument \( P \) to a constructor that is itself an inductively defined proposition by the same definition, it adds a hypothesis stating the desired claim holds when instantiated with the corresponding values from \( P \).

The difference between destructing and inverting evidence is subtle, but is illustrated by considering what each tactic would do to the proposition \( \text{ev} (S (S n)) \) would result in. In the case of destruction, it would generate two cases: one for \( \text{ev}_0 \), which would result in no evidence, and one for \( \text{ev}_{\text{SS}} \), which would result in the evidence \( \text{ev} n \). In the case of inversion, inversion would notice that by injectivity, \( \text{ev}_0 \) can only construct \( \text{ev} 0 \), so it is impossible for \( \text{ev} (S (S n)) \) to have been constructed from this case. Consequently, it would only generate one case, with evidence that \( \text{ev} n \).

Inductively defined propositions turn out to make incredibly expressive statements about programming languages. Thus, while no detailed example proofs are given in this section, many will follow.
3 Verification in a Simple Language

3.1 Defining a language and its execution

With its means for reasoning about inductively defined objects, it starts become clear why Coq is a natural tool for verification of programs. The reason is that almost any programming language can be specified as abstract syntax, abstract syntax is easily defined by a context free grammar, and context free grammars are often inductively defined. For example, a context free grammar for the C language, given in Backus-Naur form, may be found here. Note how similar it looks to Coq’s inductive definitions.

To make discussion simpler, ideas about verification will be developed using the toy language Imp, whose syntax is given (as inductively defined types) in SF as follows:

\[
\text{Inductive aexp : Type :=} \\
\quad \text{ANum : nat → aexp} \\
\quad \text{AId : id → aexp} \\
\quad \text{APlus : aexp → aexp → aexp} \\
\quad \text{AMinus : aexp → aexp → aexp} \\
\quad \text{AMult : aexp → aexp → aexp}. \\
\]

\[
\text{Inductive bexp : Type :=} \\
\quad \text{BTrue : bexp} \\
\quad \text{BFalse : bexp} \\
\quad \text{BEq : aexp → aexp → bexp} \\
\quad \text{BLe : aexp → aexp → bexp} \\
\quad \text{BNot : bexp → bexp} \\
\quad \text{BAnd : bexp → bexp → bexp}. \\
\]

\[
\text{Inductive com : Type :=} \\
\quad \text{CSkip : com} \\
\quad \text{CAss : id → aexp → com} \\
\quad \text{CSeq : com → com → com} \\
\quad \text{CIf : bexp → com → com → com} \\
\quad \text{CWhile : bexp → com → com}. \\
\]

In order to support variables (AId’s in the abstract syntax) at any point of execution, a program has a state, which is just a total map from id’s to nat’s. It is easy to define a recursive function \(\text{aeval}\) that takes an aexp and calculates the corresponding nat value and a function \(\text{beval}\) that takes a bexp and calculates the corresponding bool value.

While it may be tempting to define a similar recursive function \(\text{ceval}\) that takes a com and a beginning state \(\text{st}\), “simulates” the program, and returns a final state \(\text{st’}\), this is in fact lacking, because not every program terminates.

However, if we consider evaluation is an inductively defined relation, then infinite looping behavior may also be accounted for. We let the notation \(c / st\) \(st’\) denote the situation that command \(c\), when executed with initial state \(st\), will terminate with final state \(st’\). Then, we may define evaluation inductively, as follows:

\[
\text{Inductive ceval : com → state → state → Prop :=} \\
\]
| E_Skip : \forall st, \\
| \hspace{1cm} \text{SKIP} / st \setminus st \\
| E_Ass : \forall st a1 n x, \\
| \hspace{1cm} \text{aeval} st a1 = n \rightarrow \\
| \hspace{1cm} (x ::= a1) / st \setminus (t_update st x n) \\
| E_Seq : \forall c1 c2 st st', \\
| \hspace{1cm} c1 / st \setminus st' \rightarrow \\
| \hspace{1cm} c2 / st' \setminus st'' \rightarrow \\
| \hspace{1cm} (c1 ;; c2) / st \setminus st'' \\
| E_IfTrue : \forall st st' b c1 c2, \\
| \hspace{1cm} \text{beval} st b = \text{true} \rightarrow \\
| \hspace{1cm} c1 / st \setminus st' \rightarrow \\
| \hspace{1cm} (\text{IFB b THEN c1 ELSE c2 FI}) / st \setminus st' \\
| E_IfFalse : \forall st st' b c1 c2, \\
| \hspace{1cm} \text{beval} st b = \text{false} \rightarrow \\
| \hspace{1cm} c2 / st \setminus st' \rightarrow \\
| \hspace{1cm} (\text{IFB b THEN c1 ELSE c2 FI}) / st \setminus st' \\
| E_WhileEnd : \forall b st c, \\
| \hspace{1cm} \text{beval} st b = \text{false} \rightarrow \\
| \hspace{1cm} (\text{WHILE b DO c END}) / st \setminus st \\
| E_WhileLoop : \forall st st' st'' b c, \\
| \hspace{1cm} \text{beval} st b = \text{true} \rightarrow \\
| \hspace{1cm} c / st \setminus st' \rightarrow \\
| \hspace{1cm} (\text{WHILE b DO c END}) / st' \setminus st'' \rightarrow \\
| \hspace{1cm} (\text{WHILE b DO c END}) / st \setminus st'' \\

Infinite looping is accounted for in the sense that, for a command c that infinite loops when starting in state st, there will exist no state st' such that the proposition c / st st' is provable from the constructors listed above.

One downside of not formulating ceval as a function is that for terminating programs, we cannot use Coq’s ability to compute functions to make claims about the final state, but rather must work backwards and invert/apply the various constructors. A potential concern is that ceval is not even a partial function, i.e. that it might be possible for c / st \setminus st1 and c / st \setminus st2 to hold, but for st1 and st2 not be the same. This would violate the intuition that a straight-line programming language be deterministic – however, it is easily shown in SF that ceval is indeed deterministic.

Let us define loop : \text{com} := \text{WHILE BTrue DO SKIP END} to be a command that (to our eyes) obviously infinite loops. In our inductive proposition formulation, we can actually show that it never terminates, meaning there is no state st' that will satisfy loop / st \setminus st'.

**Theorem 3** For any starting state st, the program loop will never terminate.

**Proof** The statement is equivalent to showing that loop / st \setminus st' implies False for all st, st'.

We first remember that loop = \text{WHILE BTrue DO SKIP END}. We then induct on the the proposition loop / st \setminus st', in search of a contradiction. Induction destructs the proposition, in each case instatiating loop with the corresponding command generated by the
constructor, and adding any necessary inductive hypotheses. In most cases, the instantiation of loop will lead to a contradiction in loop = WHILE BTrue DO SKIP END, such as SKIP = WHILE BTrue DO SKIP END for the E_Skip case. These cases may be immediately disregarded.

There are only two constructors that do not result in this immediate contradiction. Unsurprisingly, they are the E_WhileEnd and E_WhileLoop cases.

In the E_WhileEnd case, the destruct generates evidence instantiating b with BTrue and c with SKIP, and also adds the hypothesis H: beval st b = false, corresponding to the loop’s condition evaluating to false, causing the loop to terminate. However, plugging in BTrue for b in H then simplifies to true = false, a clear contradiction, which completes this case.

In the E_WhileLoop case, the destruct generates evidence again instantiating b with BTrue and c with SKIP, but now adds the hypothesis H: beval st b = true, corresponding to the loop’s condition evaluating to true, causing the loop to run (at least) once. There are two additional propositions that must hold, namely c / st \st' and WHILE b DO c END / st' \st'', for some states st, st', st''.

As each of these propositions is also inductively defined in ceval, we get an inductive hypothesis for each. As the claim we seek to prove is loop / st0 \st1 implies False for any states st0 and st1, the right inductive hypotheses are c /st \st' implies False if c = loop and WHILE b DO c END / st' \st'' implies False if WHILE b DO c END = loop. In other words, we can prove False if one of c = loop or WHILE b DO c END = loop holds. But using the instantiated values b = BTrue and c = SKIP, we see that the latter is clearly an equality. Thus, by the inductive hypothesis generated by WHILE b DO c END / st' \st'', we can conclude False, as desired.

This accounts for all the cases, so by induction, the claim always holds.

---

3.2 Verifying a simple compiler

As a more extended project, I implemented and verified a simple stack compiler that turns Imp’s abstract representation of arithmetic aexp’s into a list of stack instructions for a typical stack calculator.

The following instructions are possible:
Inductive sinstr : Type :=
| SPush : nat → sinstr
| SLoad : id → sinstr
| SPlus : sinstr
| SMinus : sinstr
| SMult : sinstr.

It is easy to define a recursive function \texttt{s\_execute} which takes a state, an initial stack (a list of \texttt{nat}'s), and stack machine program (a list of \texttt{sinstr}'s), and returns the stack after executing the program. An implementation is given in \textit{Imp.v}.

The following program compiles an \texttt{aexp} into a stack machine program, such that running the program is equivalent to pushing the value of the expression on the stack:

Fixpoint s\_compile (e : aexp) : list sinstr :=
match e with
| ANum n ⇒ [SPush n]
| AId i ⇒ [SLoad i]
| APlus a1 a2 ⇒ s\_compile a1 ++ s\_compile a2 ++ [SPlus]
| AMinus a1 a2 ⇒ s\_compile a1 ++ s\_compile a2 ++ [SMinus]
| AMult a1 a2 ⇒ s\_compile a1 ++ s\_compile a2 ++ [SMult]
end.

Theorem 4 The stack compiler is correct. More formally, for any state \texttt{st} and any arithmetic expression \texttt{e}, the following holds: \texttt{s\_execute st [] (s\_compile e)} = \texttt{[aeval st e]}.

Proof First, we state a lemma describing partial execution of a list of stack instruction.

Lemma 5 For any state \texttt{st}, any arithmetic expression \texttt{e}, and any list of stack instructions \texttt{rest}, running the program \texttt{s\_compile ++ rest} with initial stack \texttt{stack} is equivalent to running the program \texttt{rest} with initial stack \texttt{(aeval st e :: stack)}. More formally, \texttt{s\_execute st stack (s\_compile e ++ rest)} = \texttt{s\_execute st (aeval st e :: stack) (rest)}.

Proof We will induct on a particular arithmetic expression \texttt{e}. If \texttt{e} is constructed by \texttt{ANum} or \texttt{AId}, the claim follows immediately by simplification.

Otherwise, \texttt{e} is of the form \texttt{BINOP e1 e2} where \texttt{BINOP} is one of \texttt{APlus}, \texttt{AMinus}, \texttt{AMult}. Induction supplies us with additional hypotheses of the form \texttt{s\_execute st stack (s\_compile e' ++ rest)} = \texttt{s\_execute st (aeval st e' :: stack) rest}, for all values of \texttt{stack}, \texttt{st}, \texttt{rest}, and \texttt{e'} equal to \texttt{e1} or \texttt{e2}. We wish to show that \texttt{s\_execute st stack (s\_compile (BINOP e1 e2) ++ rest)} = \texttt{s\_execute st (aeval st (BINOP e1 e2) :: stack) rest}.

By simply expanding the definitions of \texttt{s\_execute}, \texttt{s\_compile}, and \texttt{aeval}, applying associativity of list appending (the appends come from \texttt{s\_compile}), and using the inductive hypotheses for \texttt{e1} and \texttt{e2} (each instantiated with appropriate values for \texttt{rest}), we can reduce our goal to an identity of the form \texttt{s\_execute st (aeval st e1 BINSYM aeval st e2 :: stack) rest} = \texttt{s\_execute st (aeval st e1 BINSYM aeval st e2 :: stack) rest} where \texttt{BINSYM} is one of +, -, *.

This covers all cases, so by induction, we are done. □
To prove the theorem, we do casework on $e$. If it is constructed by $\text{Anum}$ or $\text{AId}$, the claim follows immediately by simplification.

Otherwise, $e$ is of the form $\text{BINOP} \ e_1 \ e_2$ where $\text{BINOP}$ is one of $\text{APlus}$, $\text{AMinus}$, $\text{AMult}$. After simplifying, we are left trying to prove a claim of the form $\text{s\_execute \ st \ [] \ (s\_compile \ e_1 \ ++ \ s\_compile \ e_2 \ ++ \ [\text{BINOP}]) = [\text{aeval \ st \ e_1 + aeval \ st \ e_2}]$. The expression on the left hand side is exactly in the form needed by the left hand side of the previous lemma. We apply the lemma twice and then use the definition of $\text{s\_execute}$ to reach an identity: $[\text{aeval \ st \ e_1 \ \text{BINOP} \ aeval \ st \ e_2]} = [\text{aeval \ st \ e_1 \ \text{BINOP} \ aeval \ st \ e_2}$, so we are done.

The proof in Coq goes as:

**Lemma** $\text{s\_execute\_partial} : \forall \text{st \ stack \ e \ rest}$,

$s\_execute \ st \ stack \ (s\_compile \ e \ ++ \ rest) = s\_execute \ st \ ((\text{aeval \ st \ e}) :: stack) \ (rest)$.

**Proof.**

intros st stack e. generalize dependent st. generalize dependent stack.
induction e; intros stack st rest; try (simpl; reflexivity);
(simpl; rewrite (app_assoc_reverse (s\_compile e1))); rewrite IHe1;
rewrite (app_assoc_reverse (s\_compile e2)); rewrite IHe2;
simpl; reflexivity).

Qed.

**Theorem** $\text{s\_compile\_correct} : \forall (\text{st : state}) (\text{e : aexp})$,

$s\_execute \ st \ [\] \ (s\_compile \ e) = [\text{aeval \ st \ e}]$.

**Proof.**

intros st e. destruct e; try(reflexivity);
simpl; repeat rewrite s\_execute\_partial; reflexivity.

Qed.

### 3.3 Program Equivalence

In the last section, I discussed how to verify particular programs behaved in a certain way. This naturally leads to a more general consideration, of what it means for two programs to be equivalent. We will define two programs to be *behaviorally equivalent* if from any starting state, they either both terminate in the same final state or both fail to terminate. We can write this more formally as:

**Definition** $\text{cequiv} (\text{c1 \ c2 : com}) : \text{Prop} := \forall (\text{st \ st'} : \text{state})$, 

$\text{(c1 \ / \ \text{st} \ \text{\backslash} \ \text{st'})} \leftrightarrow (\text{c2 \ / \ \text{st} \ \text{\backslash} \ \text{st'}})$.

There aren’t many new techniques that arise for proving such equivalences – the usual gamut of applying constructors and inducting on evidence will do most of the work. The one more general technique that can save work is observing that program equivalence is a congruence, meaning that if two programs have the same structure (e.g. both are $\text{WHILE}$ loops) and are composed of equivalent subprograms (continuing the $\text{WHILE}$ loop example, have equivalent guard expressions and loop bodies), then the two programs will be equivalent.
Note that this implication goes only one way: two programs with different structures can be equivalent, such as \texttt{WHILE BFalse DO c END} and \texttt{SKIP}.

For example, here is a proof that congruence holds for IF statements:

**Theorem 6** If \((b, b')\) is an equivalent pair of boolean expressions, and \((c_1, c_1')\) and \((c_2, c_2')\) are equivalent pairs of commands, then \(\texttt{IF B THEN c_1 ELSE c_2 FI}\) and \(\texttt{IF B' THEN c_1' ELSE c_2' FI}\) are also equivalent commands.

**Proof** We will just prove the case of the forward direction assuming that \(b\) is true. With this, we wish to show that \(\texttt{IF B THEN c_1 ELSE c_2 FI / st \rightarrow IF B' THEN c_1' ELSE c_2' FI / st}\). Inversion tells us that \(c_1 / st \rightarrow c_1' / st'\). Because \(b'\) and \(b\) are equivalent, if the latter is true, then the former is true. Thus, it is natural to apply the \(E_{\text{IfTrue}}\) constructor to our goal. Then, we need to show that \(b'\) is true and that \(c_1' / st \rightarrow st'\). Both of these follow from assumptions we already have.

The other cases all follow with similar reasoning.

**Theorem CIf_congruence :** \(\forall b \ b' \ c_1 \ c_1' \ c_2 \ c_2', \\left( b \equiv b' \land c_1 \equiv c_1' \land c_2 \equiv c_2' \implies \texttt{IF B THEN c_1 ELSE c_2 FI} = \texttt{IF B' THEN c_1' ELSE c_2' FI} \right)\).

**Proof.**

\begin{verbatim}
unfold bequiv, cequiv. intros b b' c1 c1' c2 c2'. split; intros Heval.
  - inversion Heval; subst; [apply E_IfTrue | apply E_IfFalse];
    [rewrite \rightarrow \exists He | apply Hc1e | rewrite \rightarrow \exists He | apply Hc2e]; assumption.
  - inversion Heval; subst; [apply E_IfTrue | apply E_IfFalse];
    [rewrite \rightarrow \exists He | apply Hc1e | rewrite \rightarrow \exists He | apply Hc2e]; assumption.
Qed.
\end{verbatim}

One interesting application of program equivalence is for justifying the validity of compiler optimizations. Ideally, optimizations should only save the computer work, but not produce any different behavior (otherwise, this is known as a compiler bug). If it can be formally verified that for all input programs, the optimized program is equivalent to the original, then the optimizer is indeed valid.

### 3.4 Hoare Logic

Hoare logic provides a systematic means of reasoning about a program of any size, by breaking down the verification into a chain of claims which must hold between consecutive instructions of the program.

For example, given an assignment instruction of the form \(X ::= a\), the claim that might hold following the instruction is something like, for any state \(st\), we have \(st \ X = \texttt{aeval st a}\). In general, if we expect \(P\) to hold after the assignment, then we want \(P'\) to hold before the assignment, where \(P'\) is the same as \(P\), except every occurrence of \(X\) has been changed to \(\texttt{aeval st a}\).

This relation between \(P\) and \(P'\) is formulaic, and such formulas can be found for just about every possible instruction in a programming language. Thus, the validity of a single,
large program may be reduced to the validity of many small implications. Specifically, we are interested in implications between assertions, where assertion is function from a state to a proposition. An assertion \( P \) implies an assertion \( P' \) if, for all states \( st \), we have \( P \ st = P' \ st \).

The units upon which Hoare logic is built upon are called, appropriately, Hoare triples. The triple consists of a command \( c \), an assertion \( P \) known as the precondition to \( c \), and an assertion \( Q \) known as the postcondition to \( c \). Casually, \( P \) holds for the state before \( c \) executes and \( Q \) holds for the state after \( c \) terminates, if it does so. More formally, they may be defined as follows:

\[ \text{Definition } \text{hoare_triple} \ (P:\text{Assertion}) \ (c:\text{com}) \ (Q:\text{Assertion}) : \text{Prop} := \forall st \ st', c / st \ \\setminus\ set' \rightarrow P \ st \rightarrow Q \ st'. \]

The development of Hoare logic allows for much of the obvious but tedious work of program verification to be automated, as a list of necessary assertion implications can be procedurally generated, and many of these implications will be simple enough such that a proof assistant like Coq can be programmed to automatically check them to be true (or, potentially, false).

4 Verification in the Wild

While I spent the majority of my time learning formal verification in the self-contained context of Coq and SF, what I learned gave me the ability to better appreciate the process of verifying real software.

For the CertiKOS kernel written in C, the first step for turning the code into something workable by Coq is to translate the C into abstract syntax specified by the CompCert compiler. It is known that this translation preserves behavior of the C program.

As the CompCert compiler was originally verified in Coq, this abstract syntax conveniently already exists as a set of Coq libraries. There are libraries for reasoning about memory, registers, external events, C data types, C expressions, etc., all based upon various inductively data types. The representation of the C language itself is not so different from that of Imp, just much more complex. The main difference lies in the ability to also reason about memory, i.e. something external to the Coq environment.

As reasoning about memory results in a lot to keep track of, especially for software consisting of thousands of lines of code, the CertiKOS approach to verification relies on what are known as abstraction layers.

Formally, a layer interface \( L \) is a pair \((A, P)\). \( A \) is typically a record type in Coq that is some representation of the computer’s abstract state. \( P \) consists of specifications describing the behavior of relevant C/assembly functions at that level. Specifications describe behavior at the level of both memory and abstract state.

Then, an abstraction layer is a triple \((L_1, M, L_2)\), along with a proof that the code \( M \) implements interface \( L_2 \) over \( L_1 \). For example, if \( L_1 = (A_1, P_1) \) and \( L_2 = (A_2, P_2) \), and \( P_2 = \sigma \cup P_1 \), then \( M \) would probably consist of code implementing some function meeting the specification \( \sigma \). The addition of \( \sigma \) in \( L_2 \) would mean that the specifications at \( L_2 \), when compared to the specification at \( L_1 \), would be able to describe more of the computer’s state in
terms of abstract state rather than just memory. Namely, whatever correspondence between memory and abstract state is given by $\sigma$ will be available to $L_2$ but not $L_1$.

Note that a benefit of encapsulating information in $A_i$’s is that when reasoning at any particular level of abstraction, the relevant information can be encoded into the $A_i$ structure while irrelevant info can just be ignored by the $A_i$ structure. This can greatly declutter the proof process. Thus, while often the information contained in $A_i$ will grow as $i$ increases, to reflect that more code has been implemented, some information may disappear, if it represents parts of the computer’s state that are no longer relevant.

The specifications that translate between what is happening in the physical computer and the defined abstract state are known as the low-level specification. Their job is to turn the messiness of C code (or, rather, CompCert’s abstract syntax) into a cleaner set of specifications that can be reasoned about more purely in Coq.

At the next level, rather than functions existing as hard-to-read abstract syntax trees, they are now Coq functions that operate on the abstract state alone. This situation is much more similar to the style of reasoning I worked with in $SF$, as the function definition may be explicitly unfolded and analyzed, and the computer state is totally represented as a Coq data structure rather than being the state of memory itself. Thus, this layered approach to verification allows complex, real-world programs to be broken into units that may be reasoned about in a way that, in relative terms, is closer in difficulty to the exercises out of $SF$.

5 Conclusion

Coq is really cool! It translates the usual technique of logical reasoning into something a computer can follow, which makes it good for reasoning about computer programs. Many of its techniques are based upon inductively defined data types. Even in understanding how to verify programs written in a toy language, it is possible to understand some of the key logical constructs for doing formal verification. In order to make reasoning about real-world software as much like reasoning about a toy language as possible, significant amounts of layering and abstraction should be used, in both the software development and proof-writing phases.

6 Acknowledgements

I’m grateful to Professor Zhong Shao for providing me direction and support over the course of the semester, and understanding the challenges of learning a new ways of understanding both logic and computer programming. I also appreciate him giving me access to his research group’s code, so that I could see what real-world formal verification looks like. Finally, I’m grateful to Mengqi Liu and Jieung Kim for patiently guiding me through the weeds of said verification process.