2 Some Quick Examples

I start off with a fully worked set of examples, building certified compilers from increasingly complicated source languages to stack machines. We meet a few useful tactics and see how they can be used in manual proofs and also how easily these proofs can be automated instead. This chapter is not meant to give full explanations of the features that are employed. Rather, it is a survey of what is possible. Later chapters introduce all the concepts in bottom-up fashion. In other words, it is expected that most readers will not understand what exactly is going on here, but I hope this demo will whet their appetite for the remaining chapters.

You can step through the source file `StackMachine.v` for this chapter interactively in Proof General. Alternatively, to get a feel for the whole life cycle of creating a Coq development, you can enter the pieces of source code in this chapter in a new `.v` file in an Emacs buffer. If you do the latter, include these two lines at the start of the file:

```
Require Import Bool Arith List CpdtTactics.
Set Implicit Arguments.
```

In general, similar commands are hidden in the book rendering of each chapter's source code, so you will need to insert them in from-scratch replayings of the code that is presented. Every chapter begins with those two lines, with the import list tweaked as appropriate, considering which definitions the chapter uses. The second command affects the default behavior of definitions regarding type inference.

2.1 Arithmetic Expressions over Natural Numbers

We begin with that staple of compiler textbooks, arithmetic expressions over a single type of numbers.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

2.1.1 Source Language

We begin with the syntax of the source language:

```
Inductive binop : Set := Plus | Times.
```

This first line of Coq code should be unsurprising to ML and Haskell programmers. An algebraic datatype **binop** is defined to stand for the binary operators of the source language. There are just two differences compared to ML and Haskell. First, we use the keyword `Inductive` in place of `data`, `datatype`, or `type`. This is not just a trivial surface syntax difference; inductive types in Coq are much more expressive than ordinary algebraic datatypes, essentially enabling us to encode all of mathematics. Second, there is the `: Set` fragment, which declares that we are defining a datatype that should be thought of as a constituent of programs. Later, we meet other options for defining datatypes in the universe of proofs or in an infinite hierarchy of universes, encompassing both programs and proofs, that is useful in higher-order constructions.

```
Inductive exp : Set :=
| Const : nat → exp
| Binop : binop → exp → exp → exp.
```

Now we define the type of arithmetic expressions. We write that a constant may be built from one argument, a natural number; and a binary operation may be built from a choice of operator and two operand expressions.

A note for readers who want to relate the book contents to Coq source code: coqdoc supports pretty-printing of tokens in LaTeX or HTML. Where you see a right arrow character, the source contains the ASCII text `->`. Other examples of this substitution appearing in this chapter are a double right arrow for `=>`, the inverted A symbol for `forall`, and the Cartesian product X for `*`. When in doubt about the ASCII version of a symbol, consult the chapter source code.

Now we are ready to say what programs in the expression language mean. We do this by writing an interpreter that can be thought of as a trivial operational or denotational semantics. (If you are not familiar with these semantic techniques, no need to worry: I stick to common sense constructions.)

```
Definition binopDenote (b : binop) : nat → nat → nat :=
  match b with
    | Plus ⇒ plus
    | Times ⇒ mult
  end.
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

The meaning of a binary operator is a binary function over naturals, defined with pattern-matching notation analogous to the `case` and `match` of ML and Haskell, and referring to the functions plus and mult from the Coq standard library. The keyword `Definition` is Coq's all-purpose notation for binding a term of the programming language to a name, with some associated syntactic sugar, like the notation we see here for defining a function. That sugar could be expanded to yield this definition:

```
Definition binopDenote : binop → nat → nat → nat :=
  fun (b : binop) ⇒
    match b with
      | Plus ⇒ plus
      | Times ⇒ mult
    end.
```

In this example, we could also omit all the type annotations, arriving at

```
Definition binopDenote := fun b ⇒
  match b with
    | Plus ⇒ plus
    | Times ⇒ mult
  end.
```

Languages like Haskell and ML have a convenient *principal types* property, which gives strong guarantees about how effective type inference will be. Unfortunately, Coq's type system is so expressive that any kind of complete type inference is impossible, and the task even seems to be hard in practice. Nonetheless, Coq includes some very helpful heuristics, many of them copying the workings of Haskell and ML type checkers for programs that fall in simple fragments of Coq's language.

This is as good a time as any to mention the profusion of different languages associated with Coq. The theoretical foundation of Coq is a formal system called the *Calculus of Inductive Constructions* (CIC) [31], which is an extension of the older *Calculus of Constructions* (CoC) [9]. CIC is quite a spartan foundation, which is helpful for proving metatheory but not so helpful for real development. Still, it is nice to know that it has been proved that CIC enjoys properties like *strong normalization* [31], meaning that every program (and more important, every proof term) terminates; and *relative consistency* [48] with systems like versions of Zermelo-Fraenkel set theory, which roughly means that you can believe that Coq proofs mean that the corresponding propositions are "really true" if you believe in set theory.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

Coq is actually based on an extension of CIC called Gallina. The text after the := and before the period in the last code example is a term of Gallina. Gallina includes several useful features that must be considered as extensions to CIC. The important metatheorems about CIC have not been extended to the full breadth of the features that go beyond the formalized language, but most Coq users do not seem concerned over this omission.

Next, there is Ltac, Coq's domain-specific language for writing proofs and decision procedures. I give some basic examples of Ltac later in this chapter, and much of this book is devoted to more elaborate Ltac examples.

Finally, commands like `Inductive` and `Definition` are part of the Vernacular, which includes all sorts of useful queries and requests to the Coq system. Every Coq source file is a series of vernacular commands, where many command forms take arguments that are Gallina or Ltac programs. (Actually, Coq source files are more like *trees* of vernacular commands, thanks to various nested scoping constructs.)

We can give a simple definition of the meaning of an expression.

```
Fixpoint expDenote (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

We declare explicitly that this is a recursive definition, using the keyword `Fixpoint`. The rest should be familiar to functional programmers.

It is convenient to be able to test definitions before starting to prove things about them. We can verify that the semantics is sensible by evaluating some sample uses, using the command `Eval`. This command takes an argument expressing a *reduction strategy*, or an order of evaluation. Unlike with ML, which hardcodes an *eager* reduction strategy, or Haskell, which hardcodes a *lazy* strategy, in Coq we are free to choose between these and many other orders of evaluation, because all Coq programs terminate. In fact, Coq silently checked termination of the last `Fixpoint` definition, using a simple heuristic based on monotonically decreasing size of arguments across recursive calls. Specifically, recursive calls must be made on arguments that were pulled out of the original recursive argument with `match` expressions. (Chapter 7 shows some ways of getting around this restriction, though simply removing the restriction would leave Coq useless as a theorem-proving tool.)

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

To return to the test evaluations, we run the `Eval` command using the `simpl` evaluation strategy that usually gets the job done.

```
Eval simpl in expDenote (Const 42).
  = 42 : nat

Eval simpl in expDenote (Binop Plus (Const 2) (Const 2)).
  = 4 : nat

Eval simpl in expDenote (Binop Times (Binop Plus (Const 2) (Const 2))
  (Const 7)).
  = 28 : nat
```

2.1.2 Target Language

We compile the source programs onto a simple stack machine, whose syntax is

```
Inductive instr : Set :=
| iConst : nat → instr
| iBinop : binop → instr.

Definition prog := list instr.
Definition stack := list nat.
```

An instruction either pushes a constant onto the stack or pops two arguments, applies a binary operator to them, and pushes the result onto the stack. A program is a list of instructions, and a stack is a list of natural numbers.

We can give instructions meanings as functions from stacks to optional stacks, where running an instruction results in None in case of a stack underflow and results in Some *s'* when the result of execution is the new stack *s'*. The infix operator :: is list cons from the Coq standard library.

```
Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
    | iConst n ⇒ Some (n :: s)
    | iBinop b ⇒
      match s with
        | arg1 :: arg2 :: s' ⇒
          Some ((binopDenote b) arg1 arg2 :: s')
        | _ ⇒ None
      end
  end.
```

With instrDenote defined, it is easy to define a function progDenote, which iterates application of instrDenote through a whole program.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
Fixpoint progDenote (p : prog) (s : stack) : option stack :=
  match p with
    | nil ⇒ Some s
    | i :: p' ⇒
      match instrDenote i s with
        | None ⇒ None
        | Some s' ⇒ progDenote p' s'
      end
  end.
```

With the two programming languages defined, we turn to the compiler definition.

2.1.3 Translation

The compiler itself is now unsurprising. The list concatenation operator ++ comes from the Coq standard library.

```
Fixpoint compile (e : exp) : prog :=
  match e with
    | Const n ⇒ iConst n :: nil
    | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

Before we set about proving that this compiler is correct, we can try a few test runs, using the sample programs from earlier.

```
Eval simpl in compile (Const 42).
  = iConst 42 :: nil : prog

Eval simpl in compile (Binop Plus (Const 2) (Const 2)).
  = iConst 2 :: iConst 2 :: iBinop Plus :: nil : prog

Eval simpl in compile (Binop Times (Binop Plus (Const 2) (Const 2))
  (Const 7)).
  = iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times
  :: nil : prog
```

We can also run the compiled programs and check that they give the right results.

```
Eval simpl in progDenote (compile (Const 42)) nil.
  = Some (42 :: nil) : option stack

Eval simpl in progDenote (compile (Binop Plus (Const 2)
  (Const 2))) nil.
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
  = Some (4 :: nil) : option stack
Eval simpl in progDenote (compile (Binop Times (Binop Plus (Const 2)
  (Const 2)) (Const 7))) nil.
  = Some (28 :: nil) : option stack
```

So far, so good, but how can we be sure the compiler operates correctly for *all* input programs?

2.1.4 Translation Correctness

We are ready to prove that the compiler is implemented correctly. We can use a new vernacular command `Theorem` to start a correctness proof, in terms of the semantics defined earlier.

```
Theorem compile_correct :
  ∀ e, progDenote (compile e) nil = Some (expDenote e :: nil).
```

Though a pencil-and-paper proof might clock out at this point, writing "by a routine induction on e," it turns out not to make sense to attack this proof directly. We need to use the standard trick of *strengthening the induction hypothesis*. We do that by proving an auxiliary lemma, using the command `Lemma` that is a synonym for `Theorem`, conventionally used for less important theorems that appear in the proofs of primary theorems.

```
Abort.

Lemma compile_correct' : ∀ e p s,
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s).
```

After the period in the `Lemma` command, we are in the *interactive proof-editing mode* and see this screen of text:

```
1 subgoal

============================
 ∀ (e : exp) (p : list instr) (s : stack),
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s)
```

Coq seems to be restating the lemma. What we see is a limited case of a more general protocol for describing where we are in a proof. We are told that we have a single subgoal. In general, during a proof, we can have many pending subgoals, each of which is a logical proposition to prove. Subgoals can be proved in any order, but it usually works best to prove them in the order that Coq chooses.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

Next in the output, the single subgoal is described in full detail. Above the double-dashed line would be free variables and hypotheses if we had any. Below the line is the conclusion, which, in general, is to be proved from the hypotheses.

We manipulate the proof state by running commands called *tactics*. One of the most important tactics is

```
induction e.
```

We declare that this proof will proceed by induction on the structure of the expression e. This swaps out the initial subgoal for two new subgoals, one for each case of the inductive proof.

```
2 subgoals

  n : nat
  ============================
  ∀ (s : stack) (p : list instr),
    progDenote (compile (Const n) ++ p) s =
    progDenote p (expDenote (Const n) :: s)

subgoal 2 is

  ∀ (s : stack) (p : list instr),
    progDenote (compile (Binop b e1 e2) ++ p) s =
    progDenote p (expDenote (Binop b e1 e2) :: s)
```

The first (current) subgoal is displayed with the double-dashed line below free variables and hypotheses, whereas later subgoals are only summarized with their conclusions. In the first subgoal, n is a free variable of type **nat**. The conclusion is the original theorem statement with e replaced by Const n. In a similar manner, the second case has e replaced by a generalized invocation of the Binop expression constructor. Proving both cases corresponds to a standard proof by structural induction.

We begin the first case with another very common tactic.

```
intros.
```

The current subgoal changes to

```
n : nat
s : stack
p : list instr
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
============================
progDenote (compile (Const n) ++ p) s =
progDenote p (expDenote (Const n) :: s)
```

We see that `intros` changes ∀-bound variables at the beginning of a goal into free variables.

To progress further, we need to use the definitions of some of the functions appearing in the goal. The `unfold` tactic replaces an identifier with its definition.

```
unfold compile.
```

```
n : nat
s : stack
p : list instr
============================
progDenote ((iConst n :: nil) ++ p) s =
progDenote p (expDenote (Const n) :: s)
```

```
unfold expDenote.
```

```
n : nat
s : stack
p : list instr
============================
progDenote ((iConst n :: nil) ++ p) s = progDenote p (n :: s)
```

We only need to unfold the first occurrence of progDenote to prove the goal. An `at` clause used with `unfold` specifies a particular occurrence of an identifier to unfold, where occurrences are counted from left to right.

```
unfold progDenote at 1.
```

```
n : nat
s : stack
p : list instr
============================
 (fix progDenote (p0 : prog) (s0 : stack) {struct p0} :
   option stack :=
     match p0 with
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
    | nil ⇒ Some s0
    | i :: p' ⇒
        match instrDenote i s0 with
        | Some s' ⇒ progDenote p' s'
        | None ⇒ None (A:=stack)
        end
    end) ((iConst n :: nil) ++ p) s =
  progDenote p (n :: s)
```

This last `unfold` has left an anonymous recursive definition of progDenote (similar to `fun` or lambda constructs that allow anonymous nonrecursive functions), which will generally happen when unfolding recursive definitions. Note that Coq has automatically renamed the `fix` arguments p and s to $p0$ and $s0$ to avoid clashes with local free variables. There is also a subterm None (A:=stack), which has an annotation specifying that the type of the term ought to be **option** stack. This is phrased as an explicit instantiation of a named type parameter A from the definition of **option**.

Fortunately, in this case, we can eliminate the complications of anonymous recursion right away, since the structure of the argument (iConst n :: nil) ++ p is known, allowing us to simplify the internal pattern match with the `simpl` tactic, which applies the same reduction strategy used earlier with `Eval`.

```
simpl.
```

```
n : nat
s : stack
p : list instr
============================
(fix progDenote (p0 : prog) (s0 : stack) {struct p0} :
 option stack :=
   match p0 with
   | nil ⇒ Some s0
   | i :: p' ⇒
       match instrDenote i s0 with
       | Some s' ⇒ progDenote p' s'
       | None ⇒ None (A:=stack)
       end
   end) p (n :: s) = progDenote p (n :: s)
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

Now we can unexpand the definition of progDenote.

```
fold progDenote.
```

```
n : nat
s : stack
p : list instr
============================
progDenote p (n :: s) = progDenote p (n :: s)
```

It looks like we are at the end of this case, since we have a trivial equality. Indeed, a single tactic finishes the case.

```
reflexivity.
```

On to the second inductive case.

```
b : binop
e1 : exp
IHe1 : ∀ (s : stack) (p : list instr),
         progDenote (compile e1 ++ p) s
         = progDenote p (expDenote e1 :: s)
e2 : exp
IHe2 : ∀ (s : stack) (p : list instr),
         progDenote (compile e2 ++ p) s
         = progDenote p (expDenote e2 :: s)
============================
 ∀ (s : stack) (p : list instr),
 progDenote (compile (Binop b e1 e2) ++ p) s =
 progDenote p (expDenote (Binop b e1 e2) :: s)
```

This is the first example of hypotheses above the double-dashed line. They are the inductive hypotheses *IHe1* and *IHe2*, corresponding to the subterms *e1* and *e2*, respectively.

We start out the same way as before, introducing new free variables and unfolding and folding the appropriate definitions. The seemingly frivolous `unfold`/`fold` pairs are actually accomplishing useful work, because `unfold` will sometimes perform easy simplifications.

```
intros.
unfold compile.
fold compile.
unfold expDenote.
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
fold expDenote.
```

The tactics we have seen so far are insufficient. No further definition unfoldings are useful, so we need to try something different.

```
b : binop
e1 : exp
IHe1 : ∀ (s : stack) (p : list instr),
         progDenote (compile e1 ++ p) s
         = progDenote p (expDenote e1 :: s)
e2 : exp
IHe2 : ∀ (s : stack) (p : list instr),
         progDenote (compile e2 ++ p) s
         = progDenote p (expDenote e2 :: s)
s : stack
p : list instr
============================
 progDenote ((compile e2 ++ compile e1 ++ iBinop b :: nil) ++ p) s
 = progDenote p (binopDenote b (expDenote e1) (expDenote e2) :: s)
```

We need the associative law of list concatenation, which is available as a theorem app_assoc_reverse in the standard library. (It is possible to tell the difference between inputs and outputs to Coq by periods at the ends of the inputs.)

```
Check app_assoc_reverse.
```

```
app_assoc_reverse
     : ∀ (A : Type) (l m n : list A), (l ++ m) ++ n = l ++ m ++ n
```

If we did not already know the name of the theorem, we could use the SearchRewrite command to find it, based on a pattern that we would like to rewrite.

```
SearchRewrite ((_ ++ _) ++ _).
```

```
app_assoc_reverse:
  ∀ (A : Type) (l m n : list A), (l ++ m) ++ n = l ++ m ++ n
app_assoc:
  ∀ (A : Type) (l m n : list A), l ++ m ++ n = (l ++ m) ++ n
```

We use app_assoc_reverse to perform a rewrite

```
rewrite app_assoc_reverse.
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

changing the conclusion to

```
progDenote (compile e2 ++ (compile e1 ++ iBinop b :: nil) ++ p) s
= progDenote p (binopDenote b (expDenote e1) (expDenote e2) :: s)
```

The left side of the equality matches the left side of the second inductive hypothesis, so we can rewrite with that hypothesis, too.

```
rewrite IHe2.
```

```
progDenote ((compile e1 ++ iBinop b :: nil) ++ p)
  (expDenote e2 :: s)
= progDenote p (binopDenote b (expDenote e1) (expDenote e2) :: s)
```

The same process lets us apply the remaining hypothesis.

```
rewrite app_assoc_reverse.
rewrite IHe1.
```

```
progDenote ((iBinop b :: nil) ++ p)
  (expDenote e1 :: expDenote e2 :: s)
= progDenote p (binopDenote b (expDenote e1) (expDenote e2) :: s)
```

Now we can apply a similar sequence of tactics to the one that ended the proof of the first case.

```
unfold progDenote at 1.
simpl.
fold progDenote.
reflexivity.
```

The proof is completed, as indicated by the message

```
Proof completed.
```

Even for simple theorems like this, the final proof script is unstructured and not very enlightening to readers. If we extend this approach to more serious theorems, we arrive at the unreadable proof scripts complained of by opponents of tactic-based proving. Fortunately, Coq has rich support for scripted automation, and we can take advantage of such a scripted tactic (defined elsewhere in the book source) to make short work of this lemma. We abort the old proof attempt and start again.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
Abort.

Lemma compile_correct' : ∀ e s p, progDenote (compile e ++ p) s =
  progDenote p (expDenote e :: s).
  induction e; crush.
Qed.
```

We need only state the basic inductive proof scheme and call a tactic that automates the tedious reasoning in between. In contrast to the period tactic terminator from the last proof, the semicolon tactic separator supports structured, compositional proofs. The tactic *t1*; *t2* has the effect of running *t1* and then running *t2* on each remaining subgoal. The semicolon is one of the most fundamental building blocks of effective proof automation. The period terminator is very useful for exploratory proving, when we need to see intermediate proof states, but final proofs of any serious complexity should have just one period, terminating a single compound tactic that probably uses semicolons.

The *crush* tactic comes from the library associated with this book and is not part of the Coq standard library. The book's library contains a number of other tactics that are especially helpful in highly automated proofs.

The `Qed` command checks that the proof is finished and, if so, saves it. The sequence of tactic commands we have used is an example of a *proof script*, or a series of Ltac programs; `Qed` uses the result of a script to generate a *proof term*, a well-typed term of Gallina. To believe that a theorem is true, we need only trust that the (relatively simple) checker for proof terms is correct; the use of proof scripts is immaterial. Part I of this book introduces the principles behind encoding all proofs as terms of Gallina.

The proof of the main theorem is now easy. We prove it with four period-terminated tactics, though separating them with semicolons would work as well; the version here is easier to step through.

```
Theorem compile_correct : ∀ e, progDenote (compile e) nil
  = Some (expDenote e :: nil).
  intros.


  e : exp
  ============================
   progDenote (compile e) nil = Some (expDenote e :: nil)
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

At this point, we want to format the left side of the equality to match the statement of compile_correct'. A theorem from the standard library is useful.

```
Check app_nil_end.
```

```
app_nil_end
     : ∀ (A : Type) (l : list A), l = l ++ nil
```

```
  rewrite (app_nil_end (compile e)).
```

This time, we explicitly specify the value of the variable l from the theorem statement, since multiple expressions of list type appear in the conclusion. The `rewrite` tactic might choose the wrong place to rewrite if we did not specify which we want.

```
e : exp
============================
 progDenote (compile e ++ nil) nil = Some (expDenote e :: nil)
```

Now we can apply the lemma.

```
  rewrite compile_correct'.
```

```
e : exp
============================
 progDenote nil (expDenote e :: nil) = Some (expDenote e :: nil)
```

We are almost done. The left and right sides of the equality match by simple symbolic evaluation. That means we are in luck because Coq identifies any pair of terms as equal whenever they normalize to the same result by symbolic evaluation. By the definition of progDenote, that is the case here, but we do not need to worry about such details. A simple invocation of `reflexivity` does the normalization and checks that the two results are syntactically equal.

```
  reflexivity.
Qed.
```

This proof can be shortened and automated, but that task is left as an exercise for the reader.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

2.2 Typed Expressions

This section builds on the initial example by adding expression forms that depend on static typing of terms for safety.

2.2.1 Source Language

We define a trivial language of types to classify expressions.

```
Inductive type : Set := Nat | Bool.
```

Like most programming languages, Coq uses case-sensitive variable names, so the user-defined type `type` is distinct from the `Type` keyword that appeared in the statement of a polymorphic theorem, and the constructor names Nat and Bool are distinct from the types **nat** and **bool** in the standard library.

Now we define an expanded set of binary operators.

```
Inductive tbinop : type → type → type → Set :=
| TPlus : tbinop Nat Nat Nat
| TTimes : tbinop Nat Nat Nat
| TEq : ∀ t, tbinop t t Bool
| TLe : tbinop Nat Nat Bool.
```

The definition of **tbinop** is different from **binop** in an important way. Where we declared that **binop** has type `Set`, here we declare that **tbinop** has type `type → type → type → Set`. We define **tbinop** as an *indexed type family*. Indexed inductive types are at the heart of Coq's expressive power; almost everything else of interest is defined in terms of them.

The intuitive explanation of **tbinop** is that a **tbinop** *t1 t2 t* is a binary operator whose operands should have types *t1* and *t2*, and whose result has type *t*. For instance, constructor TLe (for less-than-or-equal comparison of numbers) is assigned type **tbinop** Nat Nat Bool, meaning the operator's arguments are naturals and its result is Boolean. The type of TEq introduces a small bit of additional complication via polymorphism: we want to allow equality comparison of any two values of any type, as long as they have the same type.

ML and Haskell have indexed algebraic datatypes. For instance, their list types are indexed by the type of data that the list carries. However, compared to Coq, ML and Haskell 98 place two important restrictions on datatype definitions.

First, the indices of the range of each data constructor must be type variables bound at the top level of the datatype definition. There is no way in those languages to do what we did here, for instance, to say that TPlus is a constructor building a **tbinop** whose indices are all fixed at

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

Nat. *Generalized algebraic datatypes* (GADTs) [50] are a popular feature in GHC Haskell, OCaml 4, and other languages that removes this first restriction.

The second restriction is not lifted by GADTs. In ML and Haskell, indices of types must be types and may not be *expressions*. In Coq, types may be indexed by arbitrary Gallina terms. Type indices can live in the same universe as programs, and we can compute with them just as with regular programs. Haskell supports a hobbled form of computation in type indices based on multiparameter type classes, and recent extensions like type functions bring Haskell programming even closer to functional programming with types, but without dependent typing, there must always be a gap between how one programs with types and how one programs normally.

We can define a similar type family for typed expressions, where a term of type **texp** t can be assigned object language type t. (It is conventional in the world of interactive theorem proving to call the language of the proof assistant the *metalanguage* and a language being formalized the *object language*.)

```
Inductive texp : type → Set :=
| TNConst : nat → texp Nat
| TBConst : bool → texp Bool
| TBinop : ∀ t1 t2 t, tbinop t1 t2 t → texp t1 → texp t2 → texp t.
```

Thanks to the use of dependent types, every well-typed **texp** represents a well-typed source expression, by construction. This turns out to be very convenient for many things we might want to do with expressions. For instance, it is easy to adapt the interpreter approach to defining semantics. We start by defining a function mapping the types of the object language into Coq types.

```
Definition typeDenote (t : type) : Set :=
  match t with
    | Nat ⇒ nat
    | Bool ⇒ bool
  end.
```

Set, the type of types of programs, is itself a first-class type, and we can write functions that return Sets. Beyond that, the definition of typeDenote is trivial, relying on the **nat** and **bool** types from the Coq standard library. We can interpret binary operators by relying on the standard library equality test functions eqb and beq_nat for Booleans and naturals, respectively, along with a less-than-or-equal test leb.

```
Definition tbinopDenote arg1 arg2 res (b : tbinop arg1 arg2 res)
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
  : typeDenote arg1 → typeDenote arg2 → typeDenote res :=
  match b with
    | TPlus ⇒ plus
    | TTimes ⇒ mult
    | TEq Nat ⇒ beq_nat
    | TEq Bool ⇒ eqb
    | TLe ⇒ leb
  end.
```

This function has just a few differences from the denotation functions we saw earlier. First, **tbinop** is an indexed type, so its indices become additional arguments to tbinopDenote. Second, we need to perform a genuine *dependent pattern match*, where the necessary *type* of each case body depends on the *value* that has been matched. At this early stage, I do not go into detail on the many subtle aspects of Gallina that support dependent pattern matching, but the subject is central to Part II of the book.

The same tricks suffice to define an expression denotation function in an unsurprising way. Note that the `type` arguments to the TBinop constructor must be included explicitly in pattern matching, but here we write underscores because we do not need to refer to those arguments directly.

```
Fixpoint texpDenote t (e : texp t) : typeDenote t :=
  match e with
    | TNConst n ⇒ n
    | TBConst b ⇒ b
    | TBinop _ _ _ b e1 e2 ⇒
      (tbinopDenote b) (texpDenote e1) (texpDenote e2)
  end.
```

We can evaluate a few example programs to convince ourselves that this semantics is correct.

```
Eval simpl in texpDenote (TNConst 42).
  = 42 : typeDenote Nat

Eval simpl in texpDenote (TBConst true).
  = true : typeDenote Bool

Eval simpl in texpDenote (TBinop TTimes (TBinop TPlus (TNConst 2)
  (TNConst 2)) (TNConst 7)).
  = 28 : typeDenote Nat

Eval simpl in texpDenote (TBinop (TEq Nat)
  (TBinop TPlus (TNConst 2) (TNConst 2)) (TNConst 7)).
  = false : typeDenote Bool
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
Eval simpl in texpDenote (TBinop TLe (TBinop TPlus (TNConst 2)
  (TNConst 2)) (TNConst 7)).
  = true : typeDenote Bool
```

Now we are ready to define a suitable stack machine target for compilation.

2.2.2 Target Language

In the example of the untyped language, stack machine programs could encounter stack underflows and get stuck. We had to deal with this complication even though we proved that the compiler never produced underflowing programs. We could have used dependent types to force all stack machine programs to be underflow-free.

For the new languages, besides underflow, we also have the problem of stack slots with naturals instead of Booleans, or vice versa. Using indexed typed families will avoid the need to reason about potential failures.

We start by defining stack types, which classify sets of possible stacks.

```
Definition tstack := list type.
```

Any stack classified by a tstack must have exactly as many elements, and each stack element must have the type found in the same position of the stack type.

We can define instructions in terms of stack types, where every instruction's type indicates what initial stack type it expects and what final stack type it will produce.

```
Inductive tinstr : tstack → tstack → Set :=
| TiNConst : ∀ s, nat → tinstr s (Nat :: s)
| TiBConst : ∀ s, bool → tinstr s (Bool :: s)
| TiBinop : ∀ arg1 arg2 res s,
  tbinop arg1 arg2 res
  → tinstr (arg1 :: arg2 :: s) (res :: s).
```

Stack machine programs must be a similar inductive family, since if we again used the list type family, we would not be able to guarantee that intermediate stack types match within a program.

```
Inductive tprog : tstack → tstack → Set :=
| TNil : ∀ s, tprog s s
| TCons : ∀ s1 s2 s3,
  tinstr s1 s2
  → tprog s2 s3
  → tprog s1 s3.
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

Now, to define the semantics of the new target language, we need a representation for stacks at run-time. We again take advantage of type information to define types of value stacks that, by construction, contain the right number and types of elements.

```
Fixpoint vstack (ts : tstack) : Set :=
  match ts with
    | nil ⇒ unit
    | t :: ts' ⇒ typeDenote t × vstack ts'
  end%type.
```

This is another `Set`-valued function. This time it is recursive, which is perfectly valid, since `Set` is not treated specially in determining which functions may be written. We say that the value stack of an empty stack type is any value of type **unit**, which has just a single value, `tt`. A nonempty stack type leads to a value stack that is a pair, whose first element has the proper type and whose second element follows the representation for the remainder of the stack type. We write %*type* as an instruction to Coq's extensible parser. In particular, this directive applies to the whole `match` expression, which we ask to be parsed as though it were a type, so that the operator × is interpreted as Cartesian product instead of, say, multiplication. (Note that this use of *type* has no connection to the inductive type `type` that was defined earlier.)

This idea of programming with types can take a while to internalize, but it enables a very simple definition of instruction denotation. The definition is similar to what might be expected from a Lisp-like version of ML that ignored type information. Nonetheless, the fact that tinstrDenote passes the type checker guarantees that the stack machine programs can never go wrong. We use a special form of `let` to destructure a multilevel tuple.

```
Definition tinstrDenote ts ts' (i : tinstr ts ts')
  : vstack ts → vstack ts' :=
  match i with
    | TiNConst _ n ⇒ fun s ⇒ (n, s)
    | TiBConst _ b ⇒ fun s ⇒ (b, s)
    | TiBinop _ _ _ _ b ⇒ fun s ⇒
      let '(arg1, (arg2, s')) := s in
        ((tbinopDenote b) arg1 arg2, s')
  end.
```

Why do we choose to use an anonymous function to bind the initial stack in every case of the `match`? Consider this well-intentioned but invalid alternative version:

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
Definition tinstrDenote ts ts' (i : tinstr ts ts') (s : vstack ts)
  : vstack ts' :=
  match i with
    | TiNConst _ n ⇒ (n, s)
    | TiBConst _ b ⇒ (b, s)
    | TiBinop _ _ _ _ b ⇒
      let '(arg1, (arg2, s')) := s in
        ((tbinopDenote b) arg1 arg2, s')
  end.
```

The Coq type checker complains that

```
The term "(n, s)" has type "(nat * vstack ts)%type"
 while it is expected to have type "vstack ?119".
```

This and other mysteries of Coq dependent typing are explained in Part II of the book, which clarifies why it is often useful to push inside of `match` branches those function parameters whose types depend on the type of the value being matched. (That more complete treatment of Gallina's typing rules explains why this helps.)

We finish the semantics with a straightforward definition of program denotation.

```
Fixpoint tprogDenote ts ts' (p : tprog ts ts')
  : vstack ts → vstack ts' :=
  match p with
    | TNil _ ⇒ fun s ⇒ s
    | TCons _ _ _ i p' ⇒ fun s ⇒ tprogDenote p' (tinstrDenote i s)
  end.
```

The same argument-postponing trick is crucial for this definition.

2.2.3 Translation

To define the compilation, it is useful to have an auxiliary function for concatenating two stack machine programs.

```
Fixpoint tconcat ts ts' ts'' (p : tprog ts ts')
  : tprog ts' ts'' → tprog ts ts'' :=
  match p with
    | TNil _ ⇒ fun p' ⇒ p'
    | TCons _ _ _ i p1 ⇒ fun p' ⇒ TCons i (tconcat p1 p')
  end.
```

With that function in place, the compilation is defined similarly to the way it was earlier, modulo the use of dependent typing.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
Fixpoint tcompile t (e : texp t) (ts : tstack) : tprog ts (t :: ts) :=
  match e with
    | TNConst n ⇒ TCons (TiNConst _ n) (TNil _)
    | TBConst b ⇒ TCons (TiBConst _ b) (TNil _)
    | TBinop _ _ _ b e1 e2 ⇒ tconcat (tcompile e2 _)
      (tconcat (tcompile e1 _) (TCons (TiBinop _ b) (TNil _)))
  end.
```

One interesting feature of the definition is the underscores appearing to the right of ⇒ arrows. Haskell and ML programmers are familiar with compilers that infer type parameters to polymorphic values. In Coq, it is possible to go even further and ask the system to infer arbitrary terms, by writing underscores in place of specific values. You may have noticed that we have been calling functions without specifying all their arguments. For instance, the recursive calls here to tcompile omit the t argument. Coq's *implicit argument* mechanism automatically inserts underscores for arguments that it will probably be able to infer, but inference of such values is far from complete; generally, it only works in cases similar to those encountered with polymorphic type instantiation in Haskell and ML.

The underscores here are being filled in with stack types. That is, the Coq type inferencer is, in a sense, inferring something about the flow of control in the translated programs. We can look at exactly which values are filled in.

```
Print tcompile.
```

```
tcompile =
fix tcompile (t : type) (e : texp t) (ts : tstack) {struct e} :
  tprog ts (t :: ts) :=
  match e in (texp t0) return (tprog ts (t0 :: ts)) with
  | TNConst n ⇒ TCons (TiNConst ts n) (TNil (Nat :: ts))
  | TBConst b ⇒ TCons (TiBConst ts b) (TNil (Bool :: ts))
  | TBinop arg1 arg2 res b e1 e2 ⇒
      tconcat (tcompile arg2 e2 ts)
        (tconcat (tcompile arg1 e1 (arg2 :: ts))
          (TCons (TiBinop ts b) (TNil (res :: ts))))
  end
     : ∀ t : type, texp t → ∀ ts : tstack, tprog ts (t :: ts)
```

We can check that the compiler generates programs that behave appropriately on the earlier sample programs.

```
Eval simpl in tprogDenote (tcompile (TNConst 42) nil) tt.
  = (42, tt) : vstack (Nat :: nil)
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
Eval simpl in tprogDenote (tcompile (TBConst true) nil) tt.
  = (true, tt) : vstack (Bool :: nil)

Eval simpl in tprogDenote (tcompile (TBinop TTimes
  (TBinop TPlus (TNConst 2) (TNConst 2)) (TNConst 7)) nil) tt.
  = (28, tt) : vstack (Nat :: nil)

Eval simpl in tprogDenote (tcompile (TBinop (TEq Nat)
  (TBinop TPlus (TNConst 2) (TNConst 2)) (TNConst 7)) nil) tt.
  = (false, tt) : vstack (Bool :: nil)

Eval simpl in tprogDenote (tcompile (TBinop TLe
  (TBinop TPlus (TNConst 2) (TNConst 2)) (TNConst 7)) nil) tt.
  = (true, tt) : vstack (Bool :: nil)
```

The compiler seems to be working, so let us turn to proving that it *always* works.

2.2.4 Translation Correctness

We can state a correctness theorem similar to the last one.

```
Theorem tcompile_correct : ∀ t (e : texp t),
  tprogDenote (tcompile e nil) tt = (texpDenote e, tt).
```

Again, we need to strengthen the theorem statement so that the induction will go through. This time, to provide an excuse to demonstrate different tactics, I develop an alternative approach to this kind of proof, stating the key lemma as

```
Lemma tcompile_correct' : ∀ t (e : texp t) ts (s : vstack ts),
  tprogDenote (tcompile e ts) s = (texpDenote e, s).
```

While lemma compile_correct' quantified over a program that is the continuation [39] for the expression we are considering, here we avoid drawing in any extra syntactic elements. In addition to the source expression and its type, we also quantify over an initial stack type and a stack compatible with it. Running the compilation of the program starting from that stack, we should arrive at a stack that differs only in having the program's denotation pushed onto it.

Let us try to prove this theorem in the same way as in the last section.

`induction` *e*; *crush*.

We are left with this unproved conclusion:

```
tprogDenote
    (tconcat (tcompile e2 ts)
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

```
      (tconcat (tcompile e1 (arg2 :: ts))
         (TCons (TiBinop ts t) (TNil (res :: ts)))))) s =
  (tbinopDenote t (texpDenote e1) (texpDenote e2), s)
```

We need an analogue to the app_assoc_reverse theorem that we used to rewrite the goal in the last section. We can abort this proof and prove such a lemma about tconcat.

```
Abort.

Lemma tconcat_correct : ∀ ts ts' ts'' (p : tprog ts ts')
  (p' : tprog ts' ts'') (s : vstack ts),
  tprogDenote (tconcat p p') s
  = tprogDenote p' (tprogDenote p s).
  induction p; crush.
Qed.
```

This one goes through completely automatically.

Some code behind the scenes registers app_assoc_reverse for use by *crush*. We must register tconcat_correct similarly to get the same effect.

```
Hint Rewrite tconcat_correct.
```

Here we meet the pervasive concept of a *hint*. Many proofs can be found through exhaustive enumerations of combinations of possible proof steps; hints provide the set of steps to consider. The tactic *crush* is applying brute force search silently, and it will consider more possibilities as we add more hints. This particular hint asks that the lemma be used for left-to-right rewriting.

Now we are ready to return to tcompile_correct', proving it automatically this time.

```
Lemma tcompile_correct' : ∀ t (e : texp t) ts (s : vstack ts),
  tprogDenote (tcompile e ts) s = (texpDenote e, s).
  induction e; crush.
Qed.
```

We can register this main lemma as another hint, allowing us to prove the final theorem trivially.

```
Hint Rewrite tcompile_correct'.

Theorem tcompile_correct : ∀ t (e : texp t),
  tprogDenote (tcompile e nil) tt = (texpDenote e, tt).
  crush.
Qed.
```

It is worth emphasizing that we are doing more than building mathematical models. The compilers are functional programs that can be

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

executed efficiently. One strategy for doing so is based on *program extraction*, which generates OCaml code from Coq developments. For instance, we run a command to output the OCaml version of tcompile.

```
Extraction tcompile.
```

```
let rec tcompile t e ts =
  match e with
  | TNConst n ->
    TCons (ts, (Cons (Nat, ts)), (Cons (Nat, ts)),
      (TiNConst (ts, n)), (TNil (Cons (Nat, ts))))
  | TBConst b ->
    TCons (ts, (Cons (Bool, ts)), (Cons (Bool, ts)),
      (TiBConst (ts, b)), (TNil (Cons (Bool, ts))))
  | TBinop (t1, t2, t0, b, e1, e2) ->
    tconcat ts (Cons (t2, ts)) (Cons (t0, ts))
      (tcompile t2 e2 ts) (tconcat (Cons (t2, ts))
      (Cons (t1, (Cons (t2, ts)))) (Cons (t0, ts))
      (tcompile t1 e1 (Cons (t2, ts))) (TCons ((Cons (t1,
      (Cons (t2, ts)))), (Cons (t0, ts)), (Cons (t0, ts)),
      (TiBinop (t1, t2, t0, ts, b)),
      (TNil (Cons (t0, ts))))))
```

We can compile this code with the usual OCaml compiler and obtain an executable program with reasonable performance.

This chapter has given two examples of the style of Coq development that I advocate. Parts II and III of the book focus on the key elements of that style, namely, dependent types and scripted proof automation. Part I deals with more standard foundational material, but it may still be of interest to seasoned Coq hackers, since I follow the highly automated proof style even at that early stage.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/268856/9780262317870_cab.pdf by guest on 10 August 2022