An Introduction to Programming and Proving with Dependent Types in Coq

ADAM CHLIPALA
Harvard University

Computer proof assistants vary along many dimensions. Among the mature implementations, the Coq system is distinguished by two key features. First, we have support for programming with dependent types in the tradition of type theory, based on dependent function types and inductive type families. Second, we have a domain-specific language for coding correct-by-construction proof automation. Though the Coq user community has grown quite large, neither of the aspects I highlight is widely used. In this tutorial, I aim to provide a pragmatic introduction to both, showing how they can bring significant improvements in productivity.

1. INTRODUCTION

Today, several computer proof assistants have gone mainstream, as more people are using them who do not specialize in formalized mathematics. Early research laid the foundations for how systems of sufficient expressivity could be built. Thanks to the success of that research, today it is worth turning attention to the pragmatics of building large formal developments. Software engineers have developed techniques that reliably reduce the costs of writing, understanding, and maintaining large programs. Similar insights in proof engineering are accumulating as a wider audience gets involved. This tutorial aims to introduce two such ideas that I have found very useful.

The setting is the Coq proof assistant, which, to me, is distinguished from others by two key properties.

First, Coq is based on dependent type theory. In particular, Coq's implemented logic follows closely the formalism called the Calculus of Inductive Constructions. By programming with dependent types, it is often possible to "prove theorems" without writing anything that looks like a proof. Instead, this work is done via an extension of the familiar ideas of type-checking, with some of the same light-weight feel when compared to classical formal methods. Dependent types are especially useful for encoding "boring" structural invariants of data in a way that prevents the construction of any invalid object.

Second, Coq includes a domain-specific language Ltac for coding proof-finding procedures that are correct by construction. Coq satisfies the de Bruijn Criterion, which says that proofs must be checkable by a small, general-purpose, trustworthy kernel. On top of this simple language of proof terms, it is possible to build a tower of abstraction, where single "proof steps" correspond to high-level inference rules or decision procedures. Each schema of steps is implemented as a program called a tactic. There is no need to worry that a tactic bug will lead to acceptance of a false theorem, as tactics justify their decisions in terms of the core proof language. With Coq's Ltac language, every primitive operation generates a proof term automatically, and many useful combinators are provided, yielding a Turing-

complete language specialized for proof procedures.

Most Coq users are not taking much advantage of these features today. Most definitions of mathematical objects via programming use code that could just as well be written in ML, and most proofs go through much detail step by step. The former decision forces the explicit threading of invariant assertions through proofs in an ad-hoc way, and the latter decision leads to proof scripts that are very brittle. The style of automation I will use in this article goes beyond just making it easier to build the first version of a proof: with sufficient foresight, it is often possible to change a theorem statement without touching a line of proof.

Thus, this tutorial is meant for experienced Coq users and newcomers alike. The material here comes from excerpts of a draft textbook, which can be found online at

http://adam.chlipala.net/cpdt/

All of the examples in this article are rendered from literate Coq code, which can be stepped through interactively in a visual Coq environment. The book Web site includes a link to the full book source, which includes this article's sections among its chapters.

The code in this article is tested with Coq version 8.2pl1, though parts may work with other versions. Coq may be downloaded from

http://coq.inria.fr/

Two main graphical environments for Coq are available: CoqIDE, a standalone program; and Proof General, an Emacs mode. My experience is with Proof General, so I will begin the tutorial with some brief instructions on using it. Proof General can be downloaded from

http://proofgeneral.inf.ed.ac.uk/

Most of the tutorial is interface-independent.

There is a good base of educational material on the basics of Coq, which many people have used with success. In this article, I will focus on the tools that wind up consistently underused, among almost all Coq users. To clear up space for that content, I will avoid introducing some concepts that most Coq users are already learning without much trouble. For instance, I assume an understanding of the Curry-Howard Isomorphism, and I do not spend much time introducing the basics of inductive type definitions. Instead, the content here is all geared toward explaining how to write programs that make effective use of dependent typing, along with how to prove things about those programs. I use an unusually high level of proof automation, but space prevents explaining the building blocks behind it. Think of the automated proofs in this article as an advertisement for a style that is introduced more carefully in the book that these sections are excerpted from.

The next section introduces dependently-typed programs and their proofs, via some examples in simple compiler verification. After that, we take a step back and spend some time with *subset types* and their variations, a kind of dependent typing closer to what most programmers are used to. Next, we return to the full power of *inductive type families*, working through a few examples, including verified program optimization, red-black trees, and regular expression matching. In that

style of development, we often find ourselves wanting to employ the same kinds of rich data structures, much as lists recur in traditional functional programming. A section considers the possibilities for implementing such structures. The final section introduces perhaps the most arcane and least known aspect of dependently-typed programming, reasoning about programs that manipulate equality proofs.

2. ORIENTATION

I will start off by jumping right in to a fully-worked set of examples, building certified compilers from increasingly complicated source languages to stack machines. We will meet a few useful tactics and see how they can be used in manual proofs, and we will also see how easily these proofs can be automated instead. This section is meant as something of a teaser, demonstrating a relatively advanced example that showcases Coq's special advantages. The proofs shown here should give a basic sense of interaction with Coq, but I do not claim to give an introduction to the details needed to *write* those proofs, even elsewhere in the article, for the space reasons mentioned above. In this section, I also use advanced dependent typing techniques that will be introduced more systematically in later sections.

I assume that you have installed Coq and Proof General. As always, you can step through the source file `StackMachine.v` for this section interactively in Proof General. Alternatively, to get a feel for the whole lifecycle of creating a Coq development, you can enter the pieces of source code in this section in a new `.v` file in an Emacs buffer. If you do the latter, include two lines `Require Import` *List* Tactics. and `Set Implicit` *Arguments.* at the start of the file, to match some code hidden in this rendering of the section source, and be sure to run the Coq binary `coqtop` with the command-line argument `-I SRC`, where `SRC` is the path to a directory containing the book source. In either case, you will need to run `make` in the root directory of the book source distribution before getting started. If you have installed Proof General properly, it should start automatically when you visit a `.v` buffer in Emacs.

There are some minor headaches associated with getting Proof General to pass the proper command line arguments to the `coqtop` program. The best way to add settings that will be shared by many source files is to add a custom variable setting to your `.emacs` file, like this:

```
(custom-set-variables
  ...
  '(coq-prog-args '("-I" "SRC"))
  ...
)
```

The extra arguments demonstrated here are the proper choices for working with the code for this book. The ellipses stand for other Emacs customization settings you may already have. It can be helpful to save several alternate sets of flags in your `.emacs` file, with all but one commented out within the `custom-set-variables` block at any given time.

With Proof General, the portion of a buffer that Coq has processed is highlighted in some way, like being given a blue background. You step through Coq source files

by positioning the point at the position you want Coq to run to and pressing C-C C-RET. This can be used both for normal step-by-step coding, by placing the point inside some command past the end of the highlighted region; and for undoing, by placing the point inside the highlighted region.

2.1 Arithmetic Expressions Over Natural Numbers

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

2.1.1 *Source Language.*

We begin with the syntax of the source language.

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

Our first line of Coq code should be unsurprising to ML and Haskell programmers. We define an algebraic datatype **binop** to stand for the binary operators of our source language. There are just two wrinkles 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 garden variety algebraic datatypes, essentially enabling us to encode all of mathematics, though we begin humbly in this section. 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 will see 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 following along in the PDF version: coqdoc, the program used to render this article from Coq source, 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 section are a double right arrow for `=>` and the inverted 'A' symbol for `forall`. When in doubt about the ASCII version of a symbol, you can consult this article's source code.

Now we are ready to say what these programs mean. We will do this by writing an interpreter that can be thought of as a trivial operational or denotational semantics.

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

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 of 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 typing* property, which gives us 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 heuristically 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 preponderance of different languages associated with Coq. The theoretical foundation of Coq is a formal system called the *Calculus of Inductive Constructions (CIC)*, which is an extension of the older *Calculus of Constructions (CoC)*. 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*, meaning that every program (and, more importantly, every proof term) terminates; and *relative consistency* 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.

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 adds many useful features that are not compiled internally to more primitive CIC features. The important metatheorems about CIC have not been extended to the full breadth of these features, but most Coq users do not seem to lose much sleep over this omission.

Commands like `Inductive` and `Definition` are part of *the vernacular*, which includes all sorts of useful queries and requests to the Coq system.

Finally, there is *Ltac*, Coq's domain-specific language for writing proofs and decision procedures. We will see some basic examples of Ltac later in this section, in the form of atomic proof steps similar to what we would expect from single

sentences of informal proofs. The language is Turing-complete, allowing for more complex patterns of automation, a few of which we will see demonstrated in later sections. Details of the art of Ltac programming can be found in the book that this article is excerpted from.

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 old hat for functional programmers.

It is convenient to be able to test definitions before starting to prove things about them. We can verify that our semantics is sensible by evaluating some sample uses.

```
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 will compile our 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'*. :: is the "list cons" operator 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.

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

There is one interesting difference compared to our previous example of a `Fixpoint`. This recursive function takes two arguments, p and s. It is critical for the soundness of Coq that every program terminate, so a shallow syntactic termination check is imposed on every recursive function definition. One of the function parameters must be designated to decrease monotonically across recursive calls. That is, every recursive call must use a version of that argument that has been pulled out of the current argument by some number of `match` expressions. expDenote has only one argument, so we did not need to specify which of its arguments decreases. For progDenote, we resolve the ambiguity by writing {`struct` p} to indicate that argument p decreases structurally.

Recent versions of Coq will also infer a termination argument, so that we may write simply:

```
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.
```

2.1.3 *Translation.* Our compiler itself is now unsurprising. ++ is the list concatenation operator 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 our 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 our 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.
  = 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
```

2.1.4 *Translation Correctness.* We are ready to prove that our compiler is implemented correctly. We can use a new vernacular command `Theorem` to start a correctness proof, in terms of the semantics we 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:

```
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*. We find ourselves staring at this ominous 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 for us. What we are seeing 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.

Next in the output, we see our single subgoal described in full detail. There is a double-dashed line, above which would be our 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*. Let us start out by running one of the most important tactics:

```
induction e.
```

We declare that this proof will proceed by induction on the structure of the expression e. This swaps out our 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 and current subgoal is displayed with the double-dashed line below free variables and hypotheses, while later subgoals are only summarized with their conclusions. We see an example of a free variable in the first subgoal; n is a free variable of type **nat**. The conclusion is the original theorem statement where e has been replaced by Const n. In a similar manner, the second case has e replaced by a generalized invocation of the Binop expression constructor. We can see that 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
============================
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:

```
unfold progDenote at 1.
```

```
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) ((IConst n :: nil) ++ p) s =
  progDenote p (n :: s)
```

This last `unfold` has left us with an anonymous fixpoint version of progDenote, which will generally happen when unfolding recursive definitions. Fortunately, in this case, we can eliminate such complications 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:

```
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)
```

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 us off:

`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)
```

We see our 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.
fold expDenote.
```

Now we arrive at a point where the tactics we have seen so far are insufficient. No further definition unfoldings get us anywhere, so we will 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)
```

What we need is the associative law of list concatenation, available as a theorem `app_ass` in the standard library.

```
Check app_ass.
```

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

We use it to perform a rewrite:

```
rewrite app_ass.
```

changing the conclusion to:

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

Now we can notice that the lefthand side of the equality matches the lefthand 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_ass.
  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 that that ended the proof of the first case.

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

And the proof is completed, as indicated by the message:

```
Proof completed.
```

And there lies our first proof. Already, 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 that are the favorite complaints of 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) to make short work of this lemma. We abort the old proof attempt and start again.

```
Abort.

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

We need only to 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 our 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, where you 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 article 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 proof of our 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)
```

At this point, we want to massage the lefthand side 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. `rewrite` 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 lefthand and righthand sides can be seen to 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.
```

2.2 Typed Expressions

In this section, we will build on the initial example by adding additional expression forms that depend on static typing of terms for safety. We will use Coq's support for *dependent types*, where the type of a term may contain other executable terms. A canonical example is a type of arrays indexed by array sizes, where a function might take arguments `n` and `a`, assigning to `a` the type `array(int,n)` to express that `n` is the size of `a`. In Coq, dependent types arise from *dependent function types* and *inductive type families*, which, in this article, we will introduce implicitly by example.

2.2.1 *Source Language.* We define a trivial language of types to classify our expressions:

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

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
| TLt : 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.

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 to do what we did here, where we, for instance, say that TPlus is a constructor building a **tbinop** whose indices are all fixed at Nat. *Generalized algebraic datatypes (GADTs)* are a popular feature in GHC Haskell 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 like regular programs. Haskell supports a hobbled form of computation in type indices based on multi-parameter type classes, and recent extensions like type functions bring Haskell programming even closer to "real" 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.

```
Inductive texp : type → Set :=
| TNConst : nat → texp Nat
| TBConst : bool → texp Bool
| TBinop : ∀ arg1 arg2 res,
  tbinop arg1 arg2 res → texp arg1 → texp arg2 → texp res.
```

Thanks to our 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 our interpreter approach to defining semantics. We start by defining a function mapping the types of our languages into Coq types:

```
Definition typeDenote (t : type) : Set :=
  match t with
```

```
    | Nat ⇒ nat
    | Bool ⇒ bool
  end.
```

It can take a few moments to come to terms with the fact that `Set`, the type of types of programs, is itself a first-class type, and that we can write functions that return `Set`s. Past that wrinkle, the definition of typeDenote is trivial, relying on the **nat** and **bool** types from the Coq standard library.

We need to define a few auxiliary functions, implementing our boolean binary operators that do not appear with the right types in the standard library. They are entirely standard and ML-like, with the one caveat being that the Coq **nat** type uses a unary representation, where O is zero and S n is the successor of n.

```
Definition eq_bool (b1 b2 : bool) : bool :=
  match b1, b2 with
    | true, true ⇒ true
    | false, false ⇒ true
    | _, _ ⇒ false
  end.

Fixpoint eq_nat (n1 n2 : nat) : bool :=
  match n1, n2 with
    | O, O ⇒ true
    | S n1', S n2' ⇒ eq_nat n1' n2'
    | _, _ ⇒ false
  end.

Fixpoint lt (n1 n2 : nat) : bool :=
  match n1, n2 with
    | O, S _ ⇒ true
    | S n1', S n2' ⇒ lt n1' n2'
    | _, _ ⇒ false
  end.
```

Now we can interpret binary operators:

```
Definition tbinopDenote arg1 arg2 res (b : tbinop arg1 arg2 res)
  : typeDenote arg1 → typeDenote arg2 → typeDenote res :=
  match b in (tbinop arg1 arg2 res)
    return (typeDenote arg1 → typeDenote arg2 → typeDenote res) with
    | TPlus ⇒ plus
    | TTimes ⇒ mult
    | TEq Nat ⇒ eq_nat
    | TEq Bool ⇒ eq_bool
    | TLt ⇒ lt
  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* to come up with a definition of this function that type-checks. In each branch of the `match`, we need to use branch-specific information about the indices to **tbinop**.

General type inference that takes such information into account is undecidable, so it is often necessary to write annotations, like we see above on the line with `match`.

The `in` annotation restates the type of the term being case-analyzed. Though we use the same names for the indices as we use in the type of the original argument binder, these are actually fresh variables, and they are *binding occurrences*. Their scope is the `return` clause. That is, *arg1*, *arg2*, and *arg3* are new bound variables bound only within the return clause typeDenote *arg1* → typeDenote *arg2* → typeDenote *res*. By being explicit about the functional relationship between the type indices and the match result, we regain decidable type inference.

In fact, recent Coq versions use some heuristics that can save us the trouble of writing `match` annotations, and those heuristics get the job done in this case. We can get away with writing just:

```
Definition tbinopDenote arg1 arg2 res (b : tbinop arg1 arg2 res)
  : typeDenote arg1 → typeDenote arg2 → typeDenote res :=
  match b with
    | TPlus ⇒ plus
    | TTimes ⇒ mult
    | TEq Nat ⇒ eq_nat
    | TEq Bool ⇒ eq_bool
    | TLt ⇒ lt
  end.
```

The same tricks suffice to define an expression denotation function in an unsurprising way:

```
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

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

2.2.2 *Target Language.* Now we want to define a suitable stack machine target for compilation. In the example of the untyped language, stack machine programs could encounter stack underflows and "get stuck." This was unfortunate, since we had to deal with this complication even though we proved that our compiler never produced underflowing programs. We could have used dependent types to force all stack machine programs to be underflow-free.

For our new languages, besides underflow, we also have the problem of stack slots with naturals instead of bools or vice versa. This time, we will use indexed typed families to 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 tells us 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.
```

Now, to define the semantics of our new target language, we need a representation for stacks at runtime. We will 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 so that Coq knows to interpret × as Cartesian product rather than multiplication.

This idea of programming with types can take a while to internalize, but it enables a very simple definition of instruction denotation. Our definition is like what you might expect from a Lisp-like version of ML that ignored type information. Nonetheless, the fact that tinstrDenote passes the type-checker guarantees that our stack machine programs can never go wrong.

```
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 ⇒
      match s with
        (arg1, (arg2, s')) ⇒ ((tbinopDenote b) arg1 arg2, s')
      end
  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:

```
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 ⇒
      match s with
        (arg1, (arg2, s')) ⇒ ((tbinopDenote b) arg1 arg2, s')
      end
  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".
```

The text ?119 stands for a unification variable. We can try to help Coq figure out the value of this variable with an explicit annotation on our `match` expression.

```
Definition tinstrDenote ts ts' (i : tinstr ts ts') (s : vstack ts) : vstack ts' :=
  match i in tinstr ts ts' return vstack ts' with
    | TINConst _ n ⇒ (n, s)
    | TIBConst _ b ⇒ (b, s)
    | TIBinop _ _ _ _ b ⇒
      match s with
        (arg1, (arg2, s')) ⇒ ((tbinopDenote b) arg1 arg2, s')
      end
  end.
```

Now the error message changes.

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

Recall from our earlier discussion of `match` annotations that we write the annotations to express to the type-checker the relationship between the type indices of the case object and the result type of the `match`. Coq chooses to assign to the wildcard _ after TINConst the name t, and the type error is telling us that the type checker cannot prove that t is the same as ts. By moving s out of the `match`, we lose the ability to express, with `in` and `return` clauses, the relationship between the shared index ts of s and i.

There *are* reasonably general ways of getting around this problem without pushing binders inside `match`es. However, the alternatives are significantly more involved, and the technique we use here is almost certainly the best choice, whenever it applies.

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.
```

2.2.3 *Translation.* To define our 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 very similarly to how it was before, modulo the use of dependent typing.

```
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 quite 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 of 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. Inference of such

values is far from complete, though; 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 take a 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 our sample programs from above:

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

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 TLt
  (TBinop TPlus (TNConst 2) (TNConst 2)) (TNConst 7)) nil) tt.
  = (true, tt) : vstack (Bool :: nil)
```

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, I will 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" 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 that we settled on in the last section.

```
induction e; crush.
```

We are left with this unproved conclusion:

```
tprogDenote
    (tconcat (tcompile e2 ts)
        (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_ass 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_ass for use by *crush*. We must register tconcat_correct similarly to get the same effect:

```
Hint Rewrite tconcat_correct : cpdt.
```

We ask that the lemma be used for left-to-right rewriting, and we ask for the hint to be added to the hint database called *cpdt*, which is the database used by *crush*. 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' : cpdt.

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

```
Qed.
```

3. SUBSET TYPES AND VARIATIONS

Coq is often used for what we might call "classical program verification." We write programs, write their specifications, and then prove that the programs satisfy their specifications. In that setting, the programs that people write in Coq are normal functional programs that we could just as well have written in Haskell or ML. In this section, we discuss one of the simplest approaches to mixing programming and proving.

3.1 Introducing Subset Types

Let us consider several ways of implementing the natural number predecessor function. We start by displaying the definition from the standard library:

```
Print pred.
pred = fun n : nat ⇒ match n with
                     | 0 ⇒ 0
                     | S u ⇒ u
                     end
     : nat → nat
```

We can use a new command, *Extraction*, to produce an OCaml version of this function.

```
Extraction pred.
(** val pred : nat -> nat **)

let pred = function
  | O -> O
  | S u -> u
```

Returning 0 as the predecessor of 0 can come across as somewhat of a hack. In some situations, we might like to be sure that we never try to take the predecessor of 0. We can enforce this by giving pred a stronger, dependent type.

```
Lemma zgtz : 0 > 0 → False.
  crush.
Qed.

Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

We expand the type of pred to include a *proof* that its argument n is greater than 0. When n is 0, we use the proof to derive a contradiction, which we can use to build a value of any type via a vacuous pattern match. When n is a successor, we have no need for the proof and just return the answer. The proof argument

can be said to have a *dependent* type, because its type depends on the *value* of the argument n.

Coq's `Eval` command can execute particular invocations of pred_strong1 just as easily as it can execute more traditional functional programs. Note that Coq has decided that argument n of pred_strong1 can be made *implicit*, since it can be deduced from the type of the second argument, so we need not write n in function calls.

```
Theorem two_gt0 : 2 > 0.
  crush.
Qed.

Eval compute in pred_strong1 two_gt0.
     = 1
     : nat
```

One aspect in particular of the definition of pred_strong1 may be surprising. We took advantage of `Definition`'s syntactic sugar for defining function arguments in the case of n, but we bound the proofs later with explicit `fun` expressions. Let us see what happens if we write this function in the way that at first seems most natural.

```
Definition pred_strong1' (n : nat) (pf : n > 0) : nat :=
  match n with
    | O ⇒ match zgtz pf with end
    | S n' ⇒ n'
  end.

Error: In environment
n : nat
pf : n > 0
The term "pf" has type "n > 0" while it is expected to have type
"0 > 0"
```

The term zgtz *pf* fails to type-check. Somehow the type checker has failed to take into account information that follows from which `match` branch that term appears in. The problem is that, by default, `match` does not let us use such implied information. To get refined typing, we must always rely on `match` annotations, either written explicitly or inferred.

In this case, we must use a `return` annotation to declare the relationship between the *value* of the `match` discriminee and the *type* of the result. There is no annotation that lets us declare a relationship between the discriminee and the type of a variable that is already in scope; hence, we delay the binding of *pf*, so that we can use the `return` annotation to express the needed relationship.

We are lucky that Coq's heuristics infer the `return` clause (specifically, `return` $n > 0 \rightarrow$ **nat**) for us in this case. In general, however, the inference problem is undecidable. The known undecidable problem of *higher-order unification* reduces to the `match` type inference problem. Over time, Coq is enhanced with more and more heuristics to get around this problem, but there must always exist `match`es whose types Coq cannot infer without annotations.

Let us now take a look at the OCaml code Coq generates for pred_strong1.

Extraction pred_strong1.

```
(** val pred_strong1 : nat -> nat **)

let pred_strong1 = function
  | O -> assert false (* absurd case *)
  | S n' -> n'
```

The proof argument has disappeared! We get exactly the OCaml code we would have written manually. This is our first demonstration of the main technically interesting feature of Coq program extraction: proofs are erased automatically. The terms **nat** and $1 < 2$ are both types; the former is inhabited by natural numbers, while the latter is inhabited by proofs of the fact $1 < 2$. In Coq, every type itself has a type, and, from the perspective of extraction, the crucial difference between **nat** and $1 < 2$ is in their types.

```
Check nat.
  nat : Set


Check 1 < 2.
  1 < 2 : Prop
```

Extraction will erase exactly those values whose types have type `Prop`.

We can reimplement our dependently-typed pred based on *subset types*, defined in the standard library with the type family *sig*. This type family uses Coq's mechanism for inductive type definitions, which extend the familiar algebraic datatype definitions from Haskell and ML. This definition has two parameters, A and P, which are like the single type parameter of a polymorphic list type. The *sig* type has one constructor, exist, and we give the type of exist in full, rather than just giving the types of its arguments, as we do with algebraic datatypes. The more explicit form is necessary because, in Coq, the return type of a constructor is allowed to depend on the *values* of its arguments.

```
Print sig.
Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀ x : A, P x → sig P
For sig: Argument A is implicit
For exist: Argument A is implicit
```

sig is a Curry-Howard twin of existential quantification in logic, except that *sig* is in `Type`, while *ex* (the type family for $\exists$) is in `Prop`. That means that *sig* values can survive extraction, while *ex* proofs will always be erased. The actual details of extraction of *sig*s are more subtle, as we will see shortly.

We rewrite pred_strong1, using some syntactic sugar for subset types.

```
Locate "{ _: _| _}".
Notation Scope
"{ x : A | P }" := sig (fun x : A ⇒ P)
```

```
                    : type_scope
                    (default interpretation)
Definition pred_strong2 (s : {n : nat | n > 0}) : nat :=
  match s with
    | exist O pf ⇒ match zgtz pf with end
    | exist (S n') _ ⇒ n'
  end.
```

To build a value of a subset type, we use the exist constructor, and the details of how to do that follow from the output of our earlier `Print` *sig* command.

```
Eval compute in pred_strong2 (exist _ 2 two_gt0).
     = 1
     : nat
```

Extraction pred_strong2.

```
(** val pred_strong2 : nat -> nat **)

let pred_strong2 = function
  | O -> assert false (* absurd case *)
  | S n' -> n'
```

We arrive at the same OCaml code as was extracted from pred_strong1, which may seem surprising at first. The reason is that a value of *sig* is a pair of two pieces, a value and a proof about it. Extraction erases the proof, which reduces the constructor exist of *sig* to taking just a single argument. An optimization eliminates uses of datatypes with single constructors taking single arguments, and we arrive back where we started.

We can continue on in the process of refining pred's type. Let us change its result type to capture that the output is really the predecessor of the input.

```
Definition pred_strong3 (s : {n : nat | n > 0})
  : {m : nat | proj1_sig s = S m} :=
  match s return {m : nat | proj1_sig s = S m} with
    | exist 0 pf ⇒ match zgtz pf with end
    | exist (S n') pf ⇒ exist _ n' (refl_equal _)
  end.

Eval compute in pred_strong3 (exist _ 2 two_gt0).
     = exist (fun m : nat ⇒ 2 = S m) 1 (refl_equal 2)
     : {m : nat | proj1_sig (exist (lt 0) 2 two_gt0) = S m}
```

The function proj1_sig extracts the base value from a subset type. Besides the use of that function, the only other new thing is the use of the exist constructor to build a new *sig* value, and the details of how to do that follow from the output of our earlier `Print` command. The body of the *sig* type is an equality, so we can use the reflexivity proof rule refl_equal explicitly. It also turns out that we need to include an explicit `return` clause here, since Coq's heuristics are not smart enough to propagate the result type that we wrote earlier.

By now, the reader is probably ready to believe that the new *pred_strong* leads to the same OCaml code as we have seen several times so far, and Coq does not disappoint.

Extraction pred_strong3.

```
(** val pred_strong3 : nat -> nat **)

let pred_strong3 = function
  | O -> assert false (* absurd case *)
  | S n' -> n'
```

We have managed to reach a type that is, in a formal sense, the most expressive possible for pred. Any other implementation of the same type must have the same input-output behavior. However, there is still room for improvement in making this kind of code easier to write. Here is a version that takes advantage of tactic-based theorem proving. We switch back to passing a separate proof argument instead of using a subset type for the function's input, because this leads to cleaner code.

```
Definition pred_strong4 (n : nat) : n > 0 → {m : nat | n = S m}.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ False_rec _ _
      | S n' ⇒ fun _ ⇒ exist _ n' _
    end).
```

We build pred_strong4 using tactic-based proving, beginning with a `Definition` command that ends in a period before a definition is given. Such a command enters the interactive proving mode, with the type given for the new identifier as our proof goal. We do most of the work with the `refine` tactic, to which we pass a partial "proof" of the type we are trying to prove. There may be some pieces left to fill in, indicated by underscores. We make use of the combinator False_rec, which will return a value of any type when given a proof of **False**.

Any underscore that Coq cannot reconstruct with type inference is added as a proof subgoal. In this case, we have two subgoals:

```
2 subgoals

  n : nat
  _ : 0 > 0
  ============================
   False

subgoal 2 is:
 S n' = S n'
```

We can see that the first subgoal comes from the second underscore passed to False_rec, and the second subgoal comes from the second underscore passed to exist. In the first case, we see that, though we bound the proof variable with an underscore, it is still available in our proof context. It is hard to refer to underscore-

named variables in manual proofs, but automation makes short work of them. Both subgoals are easy to discharge that way, so let us back up and ask to prove all subgoals automatically.

```
  Undo.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ False_rec _ _
      | S n' ⇒ fun _ ⇒ exist _ n' _
    end); crush.
Defined.
```

We end the "proof" with `Defined` instead of `Qed`, so that the definition we constructed remains visible. This contrasts to the case of ending a proof with `Qed`, where the details of the proof are hidden afterward. Let us see what our proof script constructed.

```
Print pred_strong4.
pred_strong4 =
fun n : nat ⇒
match n as n0 return (n0 > 0 → {m : nat | n0 = S m}) with
| 0 ⇒
    fun _ : 0 > 0 ⇒
    False_rec {m : nat | 0 = S m}
      (Bool.diff_false_true
         (Bool.absurd_eq_true false
            (Bool.diff_false_true
               (Bool.absurd_eq_true false (pred_strong4_subproof n _)))))
| S n' ⇒
    fun _ : S n' > 0 ⇒
    exist (fun m : nat ⇒ S n' = S m) n' (refl_equal (S n'))
end
     : ∀ n : nat, n > 0 → {m : nat | n = S m}
```

We see the code we entered, with some proofs filled in. The first proof obligation, the second argument to False_rec, is filled in with a nasty-looking proof term that we can be glad we did not enter by hand. The second proof obligation is a simple reflexivity proof.

```
Eval compute in pred_strong4 two_gt0.
     = exist (fun m : nat ⇒ 2 = S m) 1 (refl_equal 2)
     : {m : nat | 2 = S m}
```

We are almost done with the ideal implementation of dependent predecessor. We can use Coq's syntax extension facility to arrive at code with almost no complexity beyond a Haskell or ML program with a complete specification in a comment.

```
Notation "!" := (False_rec _ _).
Notation "[ e ]" := (exist _ e _).

Definition pred_strong5 (n : nat) : n > 0 → {m : nat | n = S m}.
```

```
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ !
      | S n' ⇒ fun _ ⇒ [n']
    end); crush.
Defined.
```

By default, notations are also used in pretty-printing terms, including results of evaluation.

```
Eval compute in pred_strong5 two_gt0.
     = [1]
     : {m : nat | 2 = S m}
```

One other alternative is worth demonstrating. Recent Coq versions include a facility called *Program* that streamlines this style of definition. Here is a complete implementation using *Program*.

```
Obligation Tactic := crush.

Program Definition pred_strong6 (n : nat) (_ : n > 0)
  : {m : nat | n = S m} :=
  match n with
    | O ⇒ _
    | S n' ⇒ n'
  end.
```

Printing the resulting definition of pred_strong6 yields a term very similar to what we built with `refine`. *Program* can save time in writing programs that use subset types. Nonetheless, `refine` is often just as effective, and `refine` gives more control over the form the final term takes, which can be useful when you want to prove additional theorems about your definition. *Program* will sometimes insert type casts that can complicate theorem-proving.

```
Eval compute in pred_strong6 two_gt0.
     = [1]
     : {m : nat | 2 = S m}
```

3.2 Decidable Proposition Types

There is another type in the standard library which captures the idea of program values that indicate which of two propositions is true.

```
Print sumbool.
Inductive sumbool (A : Prop) (B : Prop) : Set :=
    left : A → {A} + {B} | right : B → {A} + {B}
For left: Argument A is implicit
For right: Argument B is implicit
```

We can define some notations to make working with *sumbool* more convenient.

```
Notation "'Yes'" := (left _ _).
```

```
Notation "'No'" := (right _ _).
Notation "'Reduce' x" := (if x then Yes else No) (at level 50).
```

The *Reduce* notation is notable because it demonstrates how `if` is overloaded in Coq. The `if` form actually works when the test expression has any two-constructor inductive type. Moreover, in the `then` and `else` branches, the appropriate constructor arguments are bound. This is important when working with *sumbool*s, when we want to have the proof stored in the test expression available when proving the proof obligations generated in the appropriate branch.

Now we can write `eq_nat_dec`, which compares two natural numbers, returning either a proof of their equality or a proof of their inequality. The code we pass to `refine` uses an anonymous recursive function built with a *fix* expression. Our proof script removes the recursive function f from the context before finding the proofs, so that we do not accidentally mention a non-structurally-recursive call to f in a proof rule.

```
Definition eq_nat_dec (n m : nat) : {n = m} + {n ≠ m}.
  refine (fix f (n m : nat) : {n = m} + {n ≠ m} :=
    match n, m with
      | O, O ⇒ Yes
      | S n', S m' ⇒ Reduce (f n' m')
      | _, _ ⇒ No
    end); clear f; crush.
Defined.
```

Our definition extracts to reasonable OCaml code.

Extraction eq_nat_dec.

```
(** val eq_nat_dec : nat -> nat -> sumbool **)

let rec eq_nat_dec n m =
  match n with
    | O -> (match m with
              | O -> Left
              | S n0 -> Right)
    | S n' -> (match m with
                 | O -> Right
                 | S m' -> eq_nat_dec n' m')
```

Proving this kind of decidable equality result is so common that Coq comes with a tactic for automating it.

```
Definition eq_nat_dec' (n m : nat) : {n = m} + {n ≠ m}.
  decide equality.
Defined.
```

Curious readers can verify that the *decide equality* version extracts to the same OCaml code as our more manual version does. That OCaml code had one undesirable property, which is that it uses `Left` and `Right` constructors instead of the boolean values built into OCaml. We can fix this, by using Coq's facility for mapping Coq inductive types to OCaml variant types.

Extract `Inductive` *sumbool* ⇒ "bool" ["true" "false"].
Extraction eq_nat_dec'.

```
(** val eq_nat_dec' : nat -> nat -> bool **)

let rec eq_nat_dec' n m0 =
  match n with
    | O -> (match m0 with
              | O -> true
              | S n0 -> false)
    | S n0 -> (match m0 with
                 | O -> false
                 | S n1 -> eq_nat_dec' n0 n1)
```

We can build "smart" versions of the usual boolean operators and put them to good use in certified programming. For instance, here is a *sumbool* version of boolean "or."

`Notation` "x || y" := (`if` *x* `then` *Yes* `else Reduce` *y*).

Let us use it for building a function that decides list membership, in terms of the standard library list membership function In.

`Print` *In.*

```
In =
fun A : Type ⇒
fix In (a : A) (l : list A) {struct l} : Prop :=
  match l with
  | nil ⇒ False
  | b :: m ⇒ b = a ∨ In a m
  end
     : ∀ A : Type, A → list A → Prop
```

We need to assume the existence of an equality decision procedure for the type of list elements. The Coq section mechanism lets us scope local assumptions over particular definitions.

```
Section In_dec.
  Variable A : Set.
  Variable A_eq_dec : ∀ x y : A, {x = y} + {x ≠ y}.
```

The final function is easy to write using the techniques we have developed so far.

```
  Definition In_dec : ∀ (x : A) (ls : list A), {In x ls} + {¬ In x ls}.
    refine (fix f (x : A) (ls : list A) : {In x ls} + {¬ In x ls} :=
      match ls with
        | nil ⇒ No
        | x' :: ls' ⇒ A_eq_dec x x' || f x ls'
      end); crush.
  Qed.
```

```
End In_dec.
```

In_dec has a reasonable extraction to OCaml.

Extraction In_dec.

```
(** val in_dec : ('a1 -> 'a1 -> bool) -> 'a1 -> 'a1 list -> bool **)

let rec in_dec a_eq_dec x = function
  | Nil -> false
  | Cons (x', ls') ->
      (match a_eq_dec x x' with
         | true -> true
         | false -> in_dec a_eq_dec x ls')
```

3.3 Partial Subset Types

Our final implementation of dependent predecessor used a very specific argument type to ensure that execution could always complete normally. Sometimes we want to allow execution to fail, and we want a more principled way of signaling that than returning a default value, as pred does for 0. One approach is to define this type family **maybe**, which is a version of *sig* that allows obligation-free failure.

```
Inductive maybe (A : Set) (P : A → Prop) : Set :=
| Unknown : maybe P
| Found : ∀ x : A, P x → maybe P.
```

We can define some new notations, analogous to those we defined for subset types.

```
Notation "{{ x | P }}" := (maybe (fun x ⇒ P)).
Notation "??" := (Unknown _).
Notation "[[ x ]]" := (Found _ x _).
```

Now our next version of pred is trivial to write.

```
Definition pred_strong7 (n : nat) : {{m | n = S m}}.
  refine (fun n ⇒
    match n with
      | O ⇒ ??
      | S n' ⇒ [[n']]
    end); trivial.
Defined.
```

Because we used **maybe**, one valid implementation of the type we gave pred_strong7 would return ?? in every case. We can strengthen the type to rule out such vacuous implementations, and the type family *sumor* from the standard library provides the easiest starting point. For type A and proposition B, $A + \{B\}$ desugars to *sumor* A B, whose values are either values of A or proofs of B.

```
Print sumor.
Inductive sumor (A : Type) (B : Prop) : Type :=
    inleft : A → A + {B} | inright : B → A + {B}
For inleft: Argument A is implicit
```

For inright: *Argument B is implicit*

We add notations for easy use of the *sumor* constructors. The second notation is specialized to *sumor*s whose A parameters are instantiated with regular subset types, since this is how we will use *sumor* below.

```
Notation "!!" := (inright _ _).
Notation "[[[ x ]]]" := (inleft _ [x]).
```

Now we are ready to give the final version of possibly-failing predecessor. The *sumor*-based type that we use is maximally expressive; any implementation of the type has the same input-output behavior.

```
Definition pred_strong8 (n : nat) : {m : nat | n = S m} + {n = 0}.
  refine (fun n ⇒
    match n with
      | O ⇒ !!
      | S n' ⇒ [[[n']]]
    end); trivial.
Defined.
```

3.4 Monadic Notations

We can treat **maybe** like a monad, in the same way that the Haskell *Maybe* type is interpreted as a failure monad. Our **maybe** has the wrong type to be a literal monad, but a "bind"-like notation will still be helpful.

```
Notation "x ← e1 ; e2" := (match e1 with
                           | Unknown ⇒ ??
                           | Found x _ ⇒ e2
                           end)
(right associativity, at level 60).
```

The meaning of $x \leftarrow e1;\ e2$ is: First run *e1*. If it fails to find an answer, then announce failure for our derived computation, too. If *e1 does* find an answer, pass that answer on to *e2* to find the final result. The variable x can be considered bound in *e2*.

This notation is very helpful for composing richly-typed procedures. For instance, here is a very simple implementation of a function to take the predecessors of two naturals at once.

```
Definition doublePred (n1 n2 : nat) : {{p | n1 = S (fst p) ∧ n2 = S (snd p)}}.
  refine (fun n1 n2 ⇒
    m1 ← pred_strong7 n1;
    m2 ← pred_strong7 n2;
    [[(m1, m2)]]); tauto.
Defined.
```

We can build a *sumor* version of the "bind" notation and use it to write a similarly straightforward version of this function.

```
Notation "x ←← e1 ; e2" := (match e1 with
                            | inright _ ⇒ !!
```

```
                                    | inleft (exist x _) ⇒ e2
                                  end)
```
(*right associativity*, `at` *level* 60).

```
Definition doublePred' (n1 n2 : nat)
  : {p : nat × nat | n1 = S (fst p) ∧ n2 = S (snd p)}
  + {n1 = 0 ∨ n2 = 0}.
  refine (fun n1 n2 ⇒
    m1 ⟵ pred_strong8 n1;
    m2 ⟵ pred_strong8 n2;
    [[[(m1, m2)]]]); tauto.
Defined.
```

3.5 A Type-Checking Example

We can apply these specification types to build a certified type-checker for a simple expression language.

```
Inductive exp : Set :=
| Nat : nat → exp
| Plus : exp → exp → exp
| Bool : bool → exp
| And : exp → exp → exp.
```

We define a simple language of types and its typing rules. The same inductive type definition mechanism that we have seen several times so far may also be used to define inductive predicates by giving their inference rules. That is the technique we use to specify the typing judgment.

```
Inductive type : Set := TNat | TBool.
```

```
Inductive hasType : exp → type → Prop :=
| HtNat : ∀ n,
  hasType (Nat n) TNat
| HtPlus : ∀ e1 e2,
  hasType e1 TNat
  → hasType e2 TNat
  → hasType (Plus e1 e2) TNat
| HtBool : ∀ b,
  hasType (Bool b) TBool
| HtAnd : ∀ e1 e2,
  hasType e1 TBool
  → hasType e2 TBool
  → hasType (And e1 e2) TBool.
```

It will be helpful to have a function for comparing two types. We build one using *decide equality*.

```
Definition eq_type_dec : ∀ t1 t2 : type, {t1 = t2} + {t1 ≠ t2}.
  decide equality.
Defined.
```

Another notation complements the monadic notation for **maybe** that we defined earlier. Sometimes we want to include "assertions" in our procedures. That is, we want to run a decision procedure and fail if it fails; otherwise, we want to continue, with the proof that it produced made available to us. This infix notation captures that idea, for a procedure that returns an arbitrary two-constructor type.

```
Notation "e1 ;; e2" := (if e1 then e2 else ??)
  (right associativity, at level 60).
```

With that notation defined, we can implement a typeCheck function, whose code is only more complex than what we would write in ML because it needs to include some extra type annotations. Every [[e]] expression adds a **hasType** proof obligation, and *crush* makes short work of them when we add **hasType**'s constructors as hints.

```
Definition typeCheck (e : exp) : {{t | hasType e t}}.
  Hint Constructors hasType.

  refine (fix F (e : exp) : {{t | hasType e t}} :=
    match e with
      | Nat _ ⇒ [[TNat]]
      | Plus e1 e2 ⇒
        t1 ← F e1;
        t2 ← F e2;
        eq_type_dec t1 TNat;;
        eq_type_dec t2 TNat;;
        [[TNat]]
      | Bool _ ⇒ [[TBool]]
      | And e1 e2 ⇒
        t1 ← F e1;
        t2 ← F e2;
        eq_type_dec t1 TBool;;
        eq_type_dec t2 TBool;;
        [[TBool]]
    end); crush.
Defined.
```

Despite manipulating proofs, our type checker is easy to run.

```
Eval simpl in typeCheck (Nat 0).
     = [[TNat]]
     : {{t | hasType (Nat 0) t}}
Eval simpl in typeCheck (Plus (Nat 1) (Nat 2)).
     = [[TNat]]
     : {{t | hasType (Plus (Nat 1) (Nat 2)) t}}
Eval simpl in typeCheck (Plus (Nat 1) (Bool false)).
     = ??
     : {{t | hasType (Plus (Nat 1) (Bool false)) t}}
```

The type-checker also extracts to some reasonable OCaml code.

```
Extraction typeCheck.
```

```
(** val typeCheck : exp -> type0 maybe **)

let rec typeCheck = function
  | Nat n -> Found TNat
  | Plus (e1, e2) ->
      (match typeCheck e1 with
         | Unknown -> Unknown
         | Found t1 ->
             (match typeCheck e2 with
                | Unknown -> Unknown
                | Found t2 ->
                    (match eq_type_dec t1 TNat with
                       | true ->
                           (match eq_type_dec t2 TNat with
                              | true -> Found TNat
                              | false -> Unknown)
                       | false -> Unknown)))
  | Bool b -> Found TBool
  | And (e1, e2) ->
      (match typeCheck e1 with
         | Unknown -> Unknown
         | Found t1 ->
             (match typeCheck e2 with
                | Unknown -> Unknown
                | Found t2 ->
                    (match eq_type_dec t1 TBool with
                       | true ->
                           (match eq_type_dec t2 TBool with
                              | true -> Found TBool
                              | false -> Unknown)
                       | false -> Unknown)))
```

We can adapt this implementation to use *sumor*, so that we know our typechecker only fails on ill-typed inputs. First, we define an analogue to the "assertion" notation.

```
Notation "e1 ;;; e2" := (if e1 then e2 else !!)
  (right associativity, at level 60).
```

Next, we prove a helpful lemma, which states that a given expression can have at most one type.

```
Lemma hasType_det : ∀ e t1,
  hasType e t1
  → ∀ t2, hasType e t2
    → t1 = t2.
  induction 1; inversion 1; crush.
Qed.
```

Now we can define the type-checker. Its type expresses that it only fails on untypable expressions.

```
Definition typeCheck' (e : exp)
  : {t : type | hasType e t} + {forall t, ~ hasType e t}.
```

`Hint` *Constructors hasType*.

We register all of the typing rules as hints.

```
Hint Resolve hasType_det.
```

hasType_det will also be useful for proving proof obligations with contradictory contexts. Since its statement includes ∀-bound variables that do not appear in its conclusion, normal *crush* will not apply this hint. Instead, we use the more expensive `eauto`, which does Prolog-style backtracking search with unification variables.

Finally, the implementation of typeCheck can be transcribed literally, simply switching notations as needed.

```
  refine (fix F (e : exp) : {t : type | hasType e t} + {forall t, ~ hasType e t} :=
    match e with
      | Nat _ => [[[TNat]]]
      | Plus e1 e2 =>
        t1 <-- F e1;
        t2 <-- F e2;
        eq_type_dec t1 TNat;;;
        eq_type_dec t2 TNat;;;
        [[[TNat]]]
      | Bool _ => [[[TBool]]]
      | And e1 e2 =>
        t1 <-- F e1;
        t2 <-- F e2;
        eq_type_dec t1 TBool;;;
        eq_type_dec t2 TBool;;;
        [[[TBool]]]
    end); clear F; crush' tt hasType; eauto.
```

We clear F, the local name for the recursive function, to avoid strange proofs that refer to recursive calls that we never make. The *crush* variant *crush'* helps us by performing automatic inversion on instances of the predicates specified in its second argument. Once we throw in `eauto` to apply hasType_det for us, we have discharged all the subgoals.

```
Defined.
```

The short implementation here hides just how time-saving automation is. Every use of one of the notations adds a proof obligation, giving us 12 in total. Most of these obligations require multiple inversions and either uses of hasType_det or applications of **hasType** rules.

The results of simplifying calls to typeCheck' look deceptively similar to the results for typeCheck, but now the types of the results provide more information.

```
Eval simpl in typeCheck' (Nat 0).
    = [[[TNat]]]
```

```
    : {t : type | hasType (Nat 0) t} +
      {(∀ t : type, ¬ hasType (Nat 0) t)}
Eval simpl in typeCheck' (Plus (Nat 1) (Nat 2)).
    = [[[TNat]]]
    : {t : type | hasType (Plus (Nat 1) (Nat 2)) t} +
      {(∀ t : type, ¬ hasType (Plus (Nat 1) (Nat 2)) t)}
Eval simpl in typeCheck' (Plus (Nat 1) (Bool false)).
    = !!
    : {t : type | hasType (Plus (Nat 1) (Bool false)) t} +
      {(∀ t : type, ¬ hasType (Plus (Nat 1) (Bool false)) t)}
```

4. MORE DEPENDENT TYPES

Subset types and their relatives help us integrate verification with programming. Though they reorganize the certified programmer's workflow, they tend not to have deep effects on proofs. We write largely the same proofs as we would for classical verification, with some of the structure moved into the programs themselves. It turns out that, when we use dependent types to their full potential, we warp the development and proving process even more than that, picking up "free theorems" to the extent that often a certified program is hardly more complex than its uncertified counterpart in Haskell or ML.

In particular, outside of the teaser from the first section of examples, we have only scratched the tip of the iceberg that is Coq's inductive definition mechanism. The inductive types we have seen so far have their counterparts in the other popular proof assistants. This section explores the strange new world of dependent inductive datatypes (that is, dependent inductive types outside `Prop`), a possibility which sets Coq apart from all of the competition not based on type theory.

4.1 Length-Indexed Lists

Many introductions to dependent types start out by showing how to use them to eliminate array bounds checks. When the type of an array tells you how many elements it has, your compiler can detect out-of-bounds dereferences statically. Since we are working in a pure functional language, the next best thing is length-indexed lists, which the following code defines.

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).
```

We see that, within its section, **ilist** is given type **nat** → `Set`. Previously, every inductive type we have seen has either had plain `Set` as its type or has been a predicate with some type ending in `Prop`. The full generality of inductive definitions lets us integrate the expressivity of predicates directly into our normal programming.

The **nat** argument to **ilist** tells us the length of the list. The types of **ilist**'s constructors tell us that a Nil list has length O and that a Cons list has length one

greater than the length of its sublist. We may apply **ilist** to any natural number, even natural numbers that are only known at runtime. It is this breaking of the *phase distinction* that characterizes **ilist** as *dependently typed*.

In expositions of list types, we usually see the length function defined first, but here that would not be a very productive function to code. Instead, let us implement list concatenation.

```
Fixpoint app n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2) :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons _ x ls1' ⇒ Cons x (app ls1' ls2)
  end.
```

In Coq version 8.1 and earlier, this definition leads to an error message:

```
The term "ls2" has type "ilist n2" while it is expected to have type
 "ilist (?14 + n2)"
```

In Coq's core language, without explicit annotations, Coq does not enrich our typing assumptions in the branches of a `match` expression. It is clear that the unification variable ?14 should be resolved to 0 in this context, so that we have $0 + n2$ reducing to $n2$, but Coq does not realize that. We cannot fix the problem using just the simple `return` clauses we applied in the last section. We need to combine a `return` clause with an `in` clause, as we saw in Section 2. This is exactly what the inference heuristics do in Coq 8.2 and later.

Specifically, Coq infers the following definition from the simpler one.

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2) :=
  match ls1 in (ilist n1) return (ilist (n1 + n2)) with
    | Nil ⇒ ls2
    | Cons _ x ls1' ⇒ Cons x (app' ls1' ls2)
  end.
```

Our app function could be typed in so-called *stratified* type systems, which avoid true dependency. We could consider the length indices to lists to live in a separate, compile-time-only universe from the lists themselves. Our next example would be harder to implement in a stratified system. We write an injection function from regular lists to length-indexed lists. A stratified implementation would need to duplicate the definition of lists across compile-time and run-time versions, and the run-time versions would need to be indexed by the compile-time versions.

```
Fixpoint inject (ls : list A) : ilist (length ls) :=
  match ls with
    | nil ⇒ Nil
    | h :: t ⇒ Cons h (inject t)
  end.
```

We can define an inverse conversion and prove that it really is an inverse.

```
Fixpoint unject n (ls : ilist n) : list A :=
  match ls with
    | Nil ⇒ nil
```

```
    | Cons _ h t ⇒ h :: unject t
  end.

Theorem inject_inverse : ∀ ls, unject (inject ls) = ls.
  induction ls; crush.
Qed.
```

Now let us attempt a function that is surprisingly tricky to write. In ML, the list head function raises an exception when passed an empty list. With length-indexed lists, we can rule out such invalid calls statically, and here is a first attempt at doing so.

```
Definition hd n (ls : ilist (S n)) : A :=
  match ls with
    | Nil ⇒ whatGoesHere?
    | Cons _ h _ ⇒ h
  end.
```

It is not clear what to write for the Nil case, so we are stuck before we even turn our function over to the type checker. We could try omitting the Nil case:

```
Definition hd n (ls : ilist (S n)) : A :=
  match ls with
    | Cons _ h _ ⇒ h
  end.

Error: Non exhaustive pattern-matching: no clause found for pattern Nil
```

Unlike in ML, we cannot use inexhaustive pattern matching, because there is no conception of a `Match` exception to be thrown. We might try using an `in` clause somehow.

```
Definition hd n (ls : ilist (S n)) : A :=
  match ls in (ilist (S n)) with
    | Cons _ h _ ⇒ h
  end.

Error: The reference n was not found in the current environment
```

In this and other cases, we feel like we want `in` clauses with type family arguments that are not variables. Unfortunately, Coq only supports variables in those positions. A completely general mechanism could only be supported with a solution to the problem of higher-order unification, which is undecidable. There *are* useful heuristics for handling non-variable indices which are gradually making their way into Coq, but we will spend some time on effective pattern matching on dependent types using only the primitive `match` annotations.

Our final, working attempt at hd uses an auxiliary function and a surprising `return` annotation.

```
Definition hd' n (ls : ilist n) :=
  match ls in (ilist n) return (match n with O ⇒ unit | S _ ⇒ A end) with
    | Nil ⇒ tt
```

```
    | Cons _ h _ ⇒ h
  end.

Definition hd n (ls : ilist (S n)) : A := hd' ls.
```

We annotate our main `match` with a type that is itself a `match`. We write that the function hd' returns **unit** when the list is empty and returns the carried type A in all other cases. In the definition of hd, we just call hd'. Because the index of ls is known to be nonzero, the type checker reduces the `match` in the type of hd' to A.

```
End ilist.
```

4.2 A Tagless Interpreter

A favorite example for motivating the power of functional programming is implementation of a simple expression language interpreter. In ML and Haskell, such interpreters are often implemented using an algebraic datatype of values, where at many points it is checked that a value was built with the right constructor of the value type. With dependent types, we can implement a *tagless* interpreter that both removes this source of runtime inefficiency and gives us more confidence that our implementation is correct.

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

Inductive exp : type → Set :=
| NConst : nat → exp Nat
| Plus : exp Nat → exp Nat → exp Nat
| Eq : exp Nat → exp Nat → exp Bool

| BConst : bool → exp Bool
| And : exp Bool → exp Bool → exp Bool
| If : ∀ t, exp Bool → exp t → exp t → exp t

| Pair : ∀ t1 t2, exp t1 → exp t2 → exp (Prod t1 t2)
| Fst : ∀ t1 t2, exp (Prod t1 t2) → exp t1
| Snd : ∀ t1 t2, exp (Prod t1 t2) → exp t2.
```

We have a standard algebraic datatype **type**, defining a type language of naturals, booleans, and product (pair) types. Then we have the indexed inductive type **exp**, where the argument to **exp** tells us the encoded type of an expression. In effect, we are defining the typing rules for expressions simultaneously with the syntax.

We can give types and expressions semantics in a new style, based critically on the chance for *type-level computation*.

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

```
    | Prod t1 t2 ⇒ typeDenote t1 × typeDenote t2
  end%type.
```

typeDenote compiles types of our object language into "native" Coq types. It is deceptively easy to implement. We can define a function expDenote that is typed in terms of typeDenote.

```
Fixpoint expDenote t (e : exp t) : typeDenote t :=
  match e with
    | NConst n ⇒ n
    | Plus e1 e2 ⇒ expDenote e1 + expDenote e2
    | Eq e1 e2 ⇒ if eq_nat_dec (expDenote e1) (expDenote e2)
      then true else false

    | BConst b ⇒ b
    | And e1 e2 ⇒ expDenote e1 && expDenote e2
    | If _ e' e1 e2 ⇒ if expDenote e' then expDenote e1 else expDenote e2

    | Pair _ _ e1 e2 ⇒ (expDenote e1, expDenote e2)
    | Fst _ _ e' ⇒ fst (expDenote e')
    | Snd _ _ e' ⇒ snd (expDenote e')
  end.
```

Despite the fancy type, the function definition is routine. In fact, it is less complicated than what we would write in ML or Haskell 98, since we do not need to worry about pushing final values in and out of an algebraic datatype. The only unusual thing is the use of an expression of the form `if` E `then true else false` in the Eq case. Remember that eq_nat_dec has a rich dependent type, rather than a simple boolean type. Coq's native `if` is overloaded to work on a test of any two-constructor type, so we can use `if` to build a simple boolean from the *sumbool* that eq_nat_dec returns.

We can implement a constant folding function and prove it correct. It will be useful to write a function pairOut that checks if an **exp** of Prod type is a pair, returning its two components if so. Unsurprisingly, a first attempt leads to a type error.

```
Definition pairOut t1 t2 (e : exp (Prod t1 t2)) : option (exp t1 × exp t2) :=
  match e in (exp (Prod t1 t2)) return option (exp t1 × exp t2) with
    | Pair _ _ e1 e2 ⇒ Some (e1, e2)
    | _ ⇒ None
  end.
```

```
Error: The reference t2 was not found in the current environment
```

We run again into the problem of not being able to specify non-variable arguments in `in` clauses. The problem would just be hopeless without a use of an `in` clause, though, since the result type of the `match` depends on an argument to **exp**. Our solution will be to use a more general type, as we did for hd. First, we define a type-valued function to use in assigning a type to pairOut.

```
Definition pairOutType (t : type) :=
```

```
  match t with
    | Prod t1 t2 ⇒ option (exp t1 × exp t2)
    | _ ⇒ unit
  end.
```

When passed a type that is a product, pairOutType returns our final desired type. On any other input type, pairOutType returns **unit**, since we do not care about extracting components of non-pairs. Now we can write another helper function to provide the default behavior of pairOut, which we will apply for inputs that are not literal pairs.

```
Definition pairOutDefault (t : type) :=
  match t return (pairOutType t) with
    | Prod _ _ ⇒ None
    | _ ⇒ tt
  end.
```

Now pairOut is deceptively easy to write.

```
Definition pairOut t (e : exp t) :=
  match e in (exp t) return (pairOutType t) with
    | Pair _ _ e1 e2 ⇒ Some (e1, e2)
    | _ ⇒ pairOutDefault _
  end.
```

There is one important subtlety in this definition. Coq allows us to use convenient ML-style pattern matching notation, but, internally and in proofs, we see that patterns are expanded out completely, matching one level of inductive structure at a time. Thus, the default case in the `match` above expands out to one case for each constructor of **exp** besides Pair, and the underscore in pairOutDefault _ is resolved differently in each case. From an ML or Haskell programmer's perspective, what we have here is type inference determining which code is run (returning either None or tt), which goes beyond what is possible with type inference guiding parametric polymorphism in Hindley-Milner languages, but is similar to what goes on with Haskell type classes.

With pairOut available, we can write cfold in a straightforward way. There are really no surprises beyond that Coq verifies that this code has such an expressive type, given the small annotation burden. In some places, we see that Coq's `match` annotation inference is too smart for its own good, and we have to turn that inference off by writing `return _`.

```
Fixpoint cfold t (e : exp t) : exp t :=
  match e with
    | NConst n ⇒ NConst n
    | Plus e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
      match e1', e2' return _ with
        | NConst n1, NConst n2 ⇒ NConst (n1 + n2)
        | _, _ ⇒ Plus e1' e2'
      end
```

```
  | Eq e1 e2 ⇒
    let e1' := cfold e1 in
    let e2' := cfold e2 in
    match e1', e2' return _ with
      | NConst n1, NConst n2 ⇒ BConst (if eq_nat_dec n1 n2
        then true else false)
      | _, _ ⇒ Eq e1' e2'
    end

  | BConst b ⇒ BConst b
  | And e1 e2 ⇒
    let e1' := cfold e1 in
    let e2' := cfold e2 in
    match e1', e2' return _ with
      | BConst b1, BConst b2 ⇒ BConst (b1 && b2)
      | _, _ ⇒ And e1' e2'
    end
  | If _ e e1 e2 ⇒
    let e' := cfold e in
    match e' with
      | BConst true ⇒ cfold e1
      | BConst false ⇒ cfold e2
      | _ ⇒ If e' (cfold e1) (cfold e2)
    end

  | Pair _ _ e1 e2 ⇒ Pair (cfold e1) (cfold e2)
  | Fst _ _ e ⇒
    let e' := cfold e in
    match pairOut e' with
      | Some p ⇒ fst p
      | None ⇒ Fst e'
    end
  | Snd _ _ e ⇒
    let e' := cfold e in
    match pairOut e' with
      | Some p ⇒ snd p
      | None ⇒ Snd e'
    end
end.
```

The correctness theorem for cfold turns out to be easy to prove, once we get over one serious hurdle.

```
Theorem cfold_correct : ∀ t (e : exp t), expDenote e = expDenote (cfold e).
  induction e; crush.
```

The first remaining subgoal is:

```
  expDenote (cfold e1) + expDenote (cfold e2) =
```

```
expDenote
  match cfold e1 with
  | NConst n1 ⇒
      match cfold e2 with
      | NConst n2 ⇒ NConst (n1 + n2)
      | Plus _ _ ⇒ Plus (cfold e1) (cfold e2)
      | Eq _ _ ⇒ Plus (cfold e1) (cfold e2)
      | BConst _ ⇒ Plus (cfold e1) (cfold e2)
      | And _ _ ⇒ Plus (cfold e1) (cfold e2)
      | If _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      | Pair _ _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      | Fst _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      | Snd _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      end
  | Plus _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Eq _ _ ⇒ Plus (cfold e1) (cfold e2)
  | BConst _ ⇒ Plus (cfold e1) (cfold e2)
  | And _ _ ⇒ Plus (cfold e1) (cfold e2)
  | If _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Pair _ _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Fst _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Snd _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  end
```

We would like to do a case analysis on cfold *e1*, and we attempt that in the way that has worked so far.

```
  destruct (cfold e1).
User error: e1 is used in hypothesis e
```

Coq gives us another cryptic error message. Like so many others, this one basically means that Coq is not able to build some proof about dependent types. It is hard to generate helpful and specific error messages for problems like this, since that would require some kind of understanding of the dependency structure of a piece of code. We will encounter many examples of case-specific tricks for recovering from errors like this one.

For our current proof, we can use a tactic *dep_destruct* defined in the book Tactics module. General elimination/inversion of dependently-typed hypotheses is undecidable, since it must be implemented with `match` expressions that have the restriction on `in` clauses that we have already discussed. *dep_destruct* makes a best effort to handle some common cases, relying upon the more primitive `dependent destruction` tactic that comes with Coq. In a later section, we will learn about the explicit manipulation of equality proofs that is behind *dep_destruct*'s implementation in Ltac, but for now, we treat it as a useful black box.

```
  dep_destruct (cfold e1).
```

This successfully breaks the subgoal into 5 new subgoals, one for each constructor of **exp** that could produce an **exp** Nat. Note that *dep_destruct* is successful in ruling

out the other cases automatically, in effect automating some of the work that we have done manually in implementing functions like hd and pairOut.

Now we can back up and give a short, automated proof, taking advantage of some of Ltac's programming features. We can use the tactic form `repeat` t to implement a loop, executing the tactic t repeatedly until it fails. We can also use `match` tactics to perform pattern-matching on proof goals, much as we perform pattern-matching on algebraic datatypes in ML or Haskell. Some convenient pattern forms are available, including *context*[p], which matches a term that has a subterm which matches pattern p.

For a more thorough introduction to Ltac programming, see the book that this article is excerpted from. For now, we only mean to give a taste of what Ltac makes possible. The only non-trivial inconvenience in this particular automated proof is that we cannot write a pattern that matches a Gallina `match` without including a case for every constructor of the inductive type we match over.

```
Restart.
induction e; crush;
  repeat (match goal with
            | [ ⊢ context[match cfold ?E with NConst _ ⇒ _ | Plus _ _ ⇒ _
                                            | Eq _ _ ⇒ _ | BConst _ ⇒ _ | And _ _ ⇒ _
                                            | If _ _ _ ⇒ _ | Pair _ _ _ ⇒ _
                                            | Fst _ _ ⇒ _ | Snd _ _ ⇒ _ end] ] ⇒
              dep_destruct (cfold E)
            | [ ⊢ context[match pairOut (cfold ?E) with Some _ ⇒ _
                                            | None ⇒ _ end] ] ⇒
              dep_destruct (cfold E)
            | [ ⊢ (if ?E then _ else _) = _ ] ⇒ destruct E
          end; crush).
Qed.
```

4.3 Dependently-Typed Red-Black Trees

Red-black trees are a favorite purely-functional data structure with an interesting invariant. We can use dependent types to enforce that operations on red-black trees preserve the invariant. For simplicity, we specialize our red-black trees to represent sets of **nat**s.

```
Inductive color : Set := Red | Black.

Inductive rbtree : color → nat → Set :=
| Leaf : rbtree Black 0
| RedNode : ∀ n, rbtree Black n → nat → rbtree Black n → rbtree Red n
| BlackNode : ∀ c1 c2 n, rbtree c1 n → nat → rbtree c2 n → rbtree Black (S n).
```

A value of type **rbtree** c d is a red-black tree node whose root has color c and that has black depth d. The latter property means that there are no more than d black-colored nodes on any path from the root to a leaf.

At first, it can be unclear that this choice of type indices tracks any useful property. To convince ourselves, we will prove that every red-black tree is balanced. We will phrase our theorem in terms of a depth calculating function that ignores

the extra information in the types. It will be useful to parameterize this function over a combining operation, so that we can re-use the same code to calculate the minimum or maximum height among all paths from root to leaf.

```
Require Import Max Min.

Section depth.
  Variable f : nat → nat → nat.

  Fixpoint depth c n (t : rbtree c n) : nat :=
    match t with
      | Leaf ⇒ 0
      | RedNode _ t1 _ t2 ⇒ S (f (depth t1) (depth t2))
      | BlackNode _ _ _ t1 _ t2 ⇒ S (f (depth t1) (depth t2))
    end.
End depth.
```

Our proof of balanced-ness decomposes naturally into a lower bound and an upper bound. We prove the lower bound first. Unsurprisingly, a tree's black depth provides such a bound on the minimum path length. We use the richly-typed procedure min_dec to do case analysis on whether min X Y equals X or Y.

```
Theorem depth_min : ∀ c n (t : rbtree c n), depth min t ≥ n.
  induction t; crush;
    match goal with
      | [ ⊢ context[min ?X ?Y] ] ⇒ destruct (min_dec X Y)
    end; crush.
Qed.
```

There is an analogous upper-bound theorem based on black depth. Unfortunately, a symmetric proof script does not suffice to establish it.

```
Theorem depth_max : ∀ c n (t : rbtree c n), depth max t ≤ 2 × n + 1.
  induction t; crush;
    match goal with
      | [ ⊢ context[max ?X ?Y] ] ⇒ destruct (max_dec X Y)
    end; crush.
```

Two subgoals remain. One of them is:

```
n : nat
t1 : rbtree Black n
n0 : nat
t2 : rbtree Black n
IHt1 : depth max t1 ≤ n + (n + 0) + 1
IHt2 : depth max t2 ≤ n + (n + 0) + 1
e : max (depth max t1) (depth max t2) = depth max t1
============================
 S (depth max t1) ≤ n + (n + 0) + 1
```

We see that *IHt1* is *almost* the fact we need, but it is not quite strong enough. We will need to strengthen our induction hypothesis to get the proof to go through.

```
Abort.
```

In particular, we prove a lemma that provides a stronger upper bound for trees with black root nodes. We got stuck above in a case about a red root node. Since red nodes have only black children, our IH strengthening will enable us to finish the proof.

```
Lemma depth_max' : ∀ c n (t : rbtree c n),
  match c with
    | Red ⇒ depth max t ≤ 2 × n + 1
    | Black ⇒ depth max t ≤ 2 × n
  end.
  induction t; crush;
    match goal with
      | [ ⊢ context[max ?X ?Y] ] ⇒ destruct (max_dec X Y)
    end; crush;
    repeat (match goal with
              | [ H : context[match ?C with Red ⇒ _ | Black ⇒ _ end] ⊢ _ ] ⇒
                destruct C
            end; crush).
Qed.
```

The original theorem follows easily from the lemma. We use the tactic `generalize` *pf*, which, when *pf* proves the proposition P, changes the goal from Q to $P \to Q$. It is useful to do this because it makes the truth of P manifest syntactically, so that automation machinery can rely on P, even if that machinery is not smart enough to establish P on its own.

```
Theorem depth_max : ∀ c n (t : rbtree c n), depth max t ≤ 2 × n + 1.
  intros; generalize (depth_max' t); destruct c; crush.
Qed.
```

The final balance theorem establishes that the minimum and maximum path lengths of any tree are within a factor of two of each other.

```
Theorem balanced : ∀ c n (t : rbtree c n), 2 × depth min t + 1 ≥ depth max t.
  intros; generalize (depth_min t); generalize (depth_max t); crush.
Qed.
```

Now we are ready to implement an example operation on our trees, insertion. Insertion can be thought of as breaking the tree invariants locally but then rebalancing. In particular, in intermediate states we find red nodes that may have red children. The type **rtree** captures the idea of such a node, continuing to track black depth as a type index.

```
Inductive rtree : nat → Set :=
| RedNode' : ∀ c1 c2 n, rbtree c1 n → nat → rbtree c2 n → rtree n.
```

Before starting to define insert, we define predicates capturing when a data value is in the set represented by a normal or possibly-invalid tree.

```
Section present.
  Variable x : nat.

  Fixpoint present c n (t : rbtree c n) : Prop :=
    match t with
```

```
    | Leaf ⇒ False
    | RedNode _ a y b ⇒ present a ∨ x = y ∨ present b
    | BlackNode _ _ _ a y b ⇒ present a ∨ x = y ∨ present b
  end.

Definition rpresent n (t : rtree n) : Prop :=
  match t with
    | RedNode' _ _ _ a y b ⇒ present a ∨ x = y ∨ present b
  end.
End present.
```

Insertion relies on two balancing operations. It will be useful to give types to these operations using a relative of the subset types from last section. While subset types let us pair a value with a proof about that value, here we want to pair a value with another non-proof dependently-typed value. The *sigT* type fills this role.

```
Locate "{ _: _& _}".

Notation Scope
"{ x : A & P }" := sigT (fun x : A ⇒ P)

Print sigT.

Inductive sigT (A : Type) (P : A → Type) : Type :=
    existT : ∀ x : A, P x → sigT P
```

It will be helpful to define a concise notation for the constructor of *sigT*.

```
Notation "{< x >}" := (existT _ _ x).
```

Each balance function is used to construct a new tree whose keys include the keys of two input trees, as well as a new key. One of the two input trees may violate the red-black alternation invariant (that is, it has an **rtree** type), while the other tree is known to be valid. Crucially, the two input trees have the same black depth.

A balance operation may return a tree whose root is of either color. Thus, we use a *sigT* type to package the result tree with the color of its root. Here is the definition of the first balance operation, which applies when the possibly-invalid **rtree** belongs to the left of the valid **rbtree**.

```
Definition balance1 n (a : rtree n) (data : nat) c2 :=
  match a in rtree n return rbtree c2 n
    → { c : color & rbtree c (S n) } with
    | RedNode' _ _ _ t1 y t2 ⇒
      match t1 in rbtree c n return rbtree _ n → rbtree c2 n
        → { c : color & rbtree c (S n) } with
        | RedNode _ a x b ⇒ fun c d ⇒
          {< RedNode (BlackNode a x b) y (BlackNode c data d) >}
        | t1' ⇒ fun t2 ⇒
          match t2 in rbtree c n return rbtree _ n → rbtree c2 n
            → { c : color & rbtree c (S n) } with
            | RedNode _ b x c ⇒ fun a d ⇒
              {< RedNode (BlackNode a y b) x (BlackNode c data d) >}
```

```
          | b ⇒ fun a t ⇒ {< BlackNode (RedNode a y b) data t >}
        end t1'
    end t2
end.
```

We apply a trick that I call the *convoy pattern*. Recall that `match` annotations only make it possible to describe a dependence of a `match` *result type* on the discriminee. There is no automatic refinement of the types of free variables. However, it is possible to effect such a refinement by finding a way to encode free variable type dependencies in the `match` result type, so that a `return` clause can express the connection.

In particular, we can extend the `match` to return *functions over the free variables whose types we want to refine*. In the case of balance1, we only find ourselves wanting to refine the type of one tree variable at a time. We match on one subtree of a node, and we want the type of the other subtree to be refined based on what we learn. We indicate this with a `return` clause starting like **rbtree** _ n → ..., where n is bound in an `in` pattern. Such a `match` expression is applied immediately to the "old version" of the variable to be refined, and the type checker is happy.

After writing this code, even I do not understand the precise details of how balancing works. I consulted Chris Okasaki's paper "Red-Black Trees in a Functional Setting" and transcribed the code to use dependent types. Luckily, the details are not so important here; types alone will tell us that insertion preserves balanced-ness, and we will prove that insertion produces trees containing the right keys.

Here is the symmetric function balance2, for cases where the possibly-invalid tree appears on the right rather than on the left.

```
Definition balance2 n (a : rtree n) (data : nat) c2 :=
  match a in rtree n return rbtree c2 n → { c : color & rbtree c (S n) } with
    | RedNode' _ _ _ t1 z t2 ⇒
      match t1 in rbtree c n return rbtree _ n → rbtree c2 n
        → { c : color & rbtree c (S n) } with
        | RedNode _ b y c ⇒ fun d a ⇒
          {< RedNode (BlackNode a data b) y (BlackNode c z d) >}
        | t1' ⇒ fun t2 ⇒
          match t2 in rbtree c n return rbtree _ n → rbtree c2 n
            → { c : color & rbtree c (S n) } with
            | RedNode _ c z' d ⇒ fun b a ⇒
              {< RedNode (BlackNode a data b) z (BlackNode c z' d) >}
            | b ⇒ fun a t ⇒ {< BlackNode t data (RedNode a z b) >}
          end t1'
      end t2
  end.
```

Now we are almost ready to get down to the business of writing an insert function. First, we enter a section that declares a variable x, for the key we want to insert.

```
Section insert.
  Variable x : nat.
```

Most of the work of insertion is done by a helper function ins, whose return types are expressed using a type-level function insResult.

```
Definition insResult c n :=
  match c with
    | Red ⇒ rtree n
    | Black ⇒ { c' : color & rbtree c' n }
  end.
```

That is, inserting into a tree with root color c and black depth n, the variety of tree we get out depends on c. If we started with a red root, then we get back a possibly-invalid tree of depth n. If we started with a black root, we get back a valid tree of depth n with a root node of an arbitrary color.

Here is the definition of ins. Again, we do not want to dwell on the functional details.

```
Fixpoint ins c n (t : rbtree c n) : insResult c n :=
  match t with
    | Leaf ⇒ {< RedNode Leaf x Leaf >}
    | RedNode _ a y b ⇒
      if le_lt_dec x y
        then RedNode' (projT2 (ins a)) y b
        else RedNode' a y (projT2 (ins b))
    | BlackNode c1 c2 _ a y b ⇒
      if le_lt_dec x y
        then
          match c1 return insResult c1 _ → _ with
            | Red ⇒ fun ins_a ⇒ balance1 ins_a y b
            | _ ⇒ fun ins_a ⇒ {< BlackNode (projT2 ins_a) y b >}
          end (ins a)
        else
          match c2 return insResult c2 _ → _ with
            | Red ⇒ fun ins_b ⇒ balance2 ins_b y a
            | _ ⇒ fun ins_b ⇒ {< BlackNode a y (projT2 ins_b) >}
          end (ins b)
  end.
```

The one new trick is a variation of the convoy pattern. In each of the last two pattern matches, we want to take advantage of the typing connection between the trees a and b. We might naively apply the convoy pattern directly on a in the first `match` and on b in the second. This satisfies the type checker per se, but it does not satisfy the termination checker. Inside each `match`, we would be calling ins recursively on a locally-bound variable. The termination checker is not smart enough to trace the dataflow into that variable, so the checker does not know that this recursive argument is smaller than the original argument. We make this fact clearer by applying the convoy pattern on *the result of a recursive call*, rather than just on that call's argument.

Finally, we are in the home stretch of our effort to define insert. We just need a few more definitions of non-recursive functions. First, we need to give the final characterization of insert's return type. Inserting into a red-rooted tree gives a

black-rooted tree where black depth has increased, and inserting into a black-rooted tree gives a tree where black depth has stayed the same and where the root is an arbitrary color.

```
Definition insertResult c n :=
  match c with
    | Red ⇒ rbtree Black (S n)
    | Black ⇒ { c' : color & rbtree c' n }
  end.
```

A simple clean-up procedure translates insResults into insertResults.

```
Definition makeRbtree c n : insResult c n → insertResult c n :=
  match c with
    | Red ⇒ fun r ⇒
      match r with
        | RedNode' _ _ _ a x b ⇒ BlackNode a x b
      end
    | Black ⇒ fun r ⇒ r
  end.
```

We modify Coq's default choice of implicit arguments for makeRbtree, so that we do not need to specify the c and n arguments explicitly in later calls.

```
Implicit Arguments makeRbtree [c n].
```

Finally, we define insert as a simple composition of ins and makeRbtree.

```
Definition insert c n (t : rbtree c n) : insertResult c n :=
  makeRbtree (ins t).
```

As we noted earlier, the type of insert guarantees that it outputs balanced trees whose depths have not increased too much. We also want to know that insert operates correctly on trees interpreted as finite sets, so we finish this section with a proof of that fact.

```
Section present.
  Variable z : nat.
```

The variable z stands for an arbitrary key. We will reason about z's presence in particular trees. As usual, outside the section the theorems we prove will quantify over all possible keys, giving us the facts we wanted.

We start by proving the correctness of the balance operations. It is useful to define a custom tactic *present_balance* that encapsulates the reasoning common to the two proofs. We use the keyword `Ltac` to assign a name to a proof script. This particular script just iterates between *crush* and identification of a tree that is being pattern-matched on and should be destructed.

```
  Ltac present_balance :=
    crush;
    repeat (match goal with
              | [ H : context[match ?T with
                               | Leaf ⇒ _
                               | RedNode _ _ _ _ ⇒ _
                               | BlackNode _ _ _ _ _ _ ⇒ _
```

```
                                end] ⊢ _ ] ⇒ dep_destruct T
    | [ ⊢ context[match ?T with
                  | Leaf ⇒ _
                  | RedNode _ _ _ _ ⇒ _
                  | BlackNode _ _ _ _ _ _ ⇒ _
                  end] ] ⇒ dep_destruct T
  end; crush).
```

The balance correctness theorems are simple first-order logic equivalences, where we use the function projT2 to project the payload of a $sigT$ value.

```
Lemma present_balance1 : ∀ n (a : rtree n) (y : nat) c2 (b : rbtree c2 n),
  present z (projT2 (balance1 a y b))
  ↔ rpresent z a ∨ z = y ∨ present z b.
  destruct a; present_balance.
Qed.

Lemma present_balance2 : ∀ n (a : rtree n) (y : nat) c2 (b : rbtree c2 n),
  present z (projT2 (balance2 a y b))
  ↔ rpresent z a ∨ z = y ∨ present z b.
  destruct a; present_balance.
Qed.
```

To state the theorem for ins, it is useful to define a new type-level function, since ins returns different result types based on the type indices passed to it. Recall that x is the section variable standing for the key we are inserting.

```
Definition present_insResult c n :=
  match c return (rbtree c n → insResult c n → Prop) with
    | Red ⇒ fun t r ⇒ rpresent z r ↔ z = x ∨ present z t
    | Black ⇒ fun t r ⇒ present z (projT2 r) ↔ z = x ∨ present z t
  end.
```

Now the statement and proof of the ins correctness theorem are straightforward, if verbose. We proceed by induction on the structure of a tree, followed by finding case analysis opportunities on expressions we see being analyzed in `if` or `match` expressions. After that, we pattern-match to find opportunities to use the theorems we proved about balancing. Finally, we identify two variables that are asserted by some hypothesis to be equal, and we use that hypothesis to replace one variable with the other everywhere.

```
Theorem present_ins : ∀ c n (t : rbtree c n),
  present_insResult t (ins t).
  induction t; crush;
    repeat (match goal with
              | [ H : context[if ?E then _ else _] ⊢ _ ] ⇒ destruct E
              | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
              | [ H : context[match ?C with Red ⇒ _ | Black ⇒ _ end]
                  ⊢ _ ] ⇒ destruct C
            end; crush);
    try match goal with
          | [ H : context[balance1 ?A ?B ?C] ⊢ _ ] ⇒
```

```
            generalize (present_balance1 A B C)
        end;
      try match goal with
            | [ H : context[balance2 ?A ?B ?C] ⊢ _ ] ⇒
              generalize (present_balance2 A B C)
        end;
      try match goal with
            | [ ⊢ context[balance1 ?A ?B ?C] ] ⇒
              generalize (present_balance1 A B C)
        end;
      try match goal with
            | [ ⊢ context[balance2 ?A ?B ?C] ] ⇒
              generalize (present_balance2 A B C)
        end;
      crush;
        match goal with
          | [ z : nat, x : nat ⊢ _ ] ⇒
            match goal with
              | [ H : z = x ⊢ _ ] ⇒ rewrite H in *; clear H
            end
        end;
        tauto.
  Qed.
```

The hard work is done. The most readable way to state correctness of insert involves splitting the property into two color-specific theorems. We write a tactic to encapsulate the reasoning steps that work to establish both facts.

```
  Ltac present_insert :=
    unfold insert; intros n t; inversion t;
      generalize (present_ins t); simpl;
        dep_destruct (ins t); tauto.

  Theorem present_insert_Red : ∀ n (t : rbtree Red n),
    present z (insert t)
    ↔ (z = x ∨ present z t).
    present_insert.
  Qed.

  Theorem present_insert_Black : ∀ n (t : rbtree Black n),
    present z (projT2 (insert t))
    ↔ (z = x ∨ present z t).
    present_insert.
  Qed.
End present.
End insert.
```

4.4 A Certified Regular Expression Matcher

Another interesting example is regular expressions with dependent types that express which predicates over strings particular regexps implement. We can then assign a dependent type to a regular expression matching function, guaranteeing that it always decides the string property that we expect it to decide.

Before defining the syntax of expressions, it is helpful to define an inductive type capturing the meaning of the Kleene star. We use Coq's string support, which comes through a combination of the *Strings* library and some parsing notations built into Coq. Operators like ++ and functions like length that we know from lists are defined again for strings. Notation scopes help us control which versions we want to use in particular contexts.

```
Require Import Ascii String.
Open Scope string_scope.

Section star.
  Variable P : string → Prop.

  Inductive star : string → Prop :=
  | Empty : star ""
  | Iter : ∀ s1 s2,
    P s1
    → star s2
    → star (s1 ++ s2).
End star.
```

Now we can make our first attempt at defining a **regexp** type that is indexed by predicates on strings. Here is a reasonable-looking definition that is restricted to constant characters and concatenation. We use Coq's string type, which is essentially a specialized list type with "nil" constructor "" and "cons" constructor String.

```
Inductive regexp : (string → Prop) → Set :=
| Char : ∀ ch : ascii,
  regexp (fun s ⇒ s = String ch "")
| Concat : ∀ (P1 P2 : string → Prop) (r1 : regexp P1) (r2 : regexp P2),
  regexp (fun s ⇒ ∃ s1, ∃ s2, s = s1 ++ s2 ∧ P1 s1 ∧ P2 s2).
```

```
User error: Large non-propositional inductive types must be in Type
```

What is a large inductive type? In Coq, it is an inductive type that has a constructor which quantifies over some type of type `Type`. We have not worked with `Type` very much to this point. Every term of CIC has a type, including `Set` and `Prop`, which are assigned type `Type`. The type **string** → `Prop` from the failed definition also has type `Type`.

It turns out that allowing large inductive types in `Set` leads to contradictions when combined with certain kinds of classical logic reasoning. Thus, by default, such types are ruled out. There is a simple fix for our **regexp** definition, which is to place our new type in `Type`. While fixing the problem, we also expand the list of constructors to cover the remaining regular expression operators.

```
Inductive regexp : (string → Prop) → Type :=
| Char : ∀ ch : ascii,
  regexp (fun s ⇒ s = String ch "")
| Concat : ∀ P1 P2 (r1 : regexp P1) (r2 : regexp P2),
  regexp (fun s ⇒ ∃ s1, ∃ s2, s = s1 ++ s2 ∧ P1 s1 ∧ P2 s2)
| Or : ∀ P1 P2 (r1 : regexp P1) (r2 : regexp P2),
  regexp (fun s ⇒ P1 s ∨ P2 s)
| Star : ∀ P (r : regexp P),
  regexp (star P).
```

Many theorems about strings are useful for implementing a certified regexp matcher, and few of them are in the *Strings* library. The source to this section includes statements, proofs, and hint commands for a handful of such omitted theorems. Since they are orthogonal to our use of dependent types, we hide them in the rendered version.

A few auxiliary functions help us in our final matcher definition. The function `split` will be used to implement the regexp concatenation case.

```
Section split.
  Variables P1 P2 : string → Prop.
  Variable P1_dec : ∀ s, {P1 s} + {¬ P1 s}.
  Variable P2_dec : ∀ s, {P2 s} + {¬ P2 s}.
```

We require a choice of two arbitrary string predicates and functions for deciding them.

```
  Variable s : string.
```

Our computation will take place relative to a single fixed string, so it is easiest to make it a `Variable`, rather than an explicit argument to our functions.

split' is the workhorse behind `split`. It searches through the possible ways of splitting s into two pieces, checking the two predicates against each such pair. split' progresses right-to-left, from splitting all of s into the first piece to splitting all of s into the second piece. It takes an extra argument, n, which specifies how far along we are in this search process.

```
  Definition split' (n : nat) : n ≤ length s
    → {∃ s1, ∃ s2, length s1 ≤ n ∧ s1 ++ s2 = s ∧ P1 s1 ∧ P2 s2}
    + {∀ s1 s2, length s1 ≤ n → s1 ++ s2 = s → ¬ P1 s1 ∨ ¬ P2 s2}.
    refine (fix F (n : nat) : n ≤ length s
      → {∃ s1, ∃ s2, length s1 ≤ n ∧ s1 ++ s2 = s ∧ P1 s1 ∧ P2 s2}
      + {∀ s1 s2, length s1 ≤ n → s1 ++ s2 = s → ¬ P1 s1 ∨ ¬ P2 s2} :=
      match n with
        | O ⇒ fun _ ⇒ Reduce (P1_dec "" && P2_dec s)
        | S n' ⇒ fun _ ⇒ (P1_dec (substring 0 (S n') s)
            && P2_dec (substring (S n') (length s - S n') s))
          || F n' _
      end); clear F; crush; eauto 7;
    match goal with
      | [ _ : length ?S ≤ 0 ⊢ _ ] ⇒ destruct S
      | [ _ : length ?S' ≤ S ?N ⊢ _ ] ⇒
```

```
      generalize (eq_nat_dec (length S') (S N)); destruct 1
  end; crush.
Defined.
```

There is one subtle point in the split' code that is worth mentioning. The main body of the function is a `match` on n. In the case where n is known to be S n', we write S n' in several places where we might be tempted to write n. However, without further work to craft proper `match` annotations, the type-checker does not use the equality between n and S n'. Thus, it is common to see patterns repeated in `match` case bodies in dependently-typed Coq code. We can at least use a `let` expression to avoid copying the pattern more than once, replacing the second case body with:

```
| S n' ⇒ fun _ ⇒ let n := S n' in
  (P1_dec (substring 0 n s)
    && P2_dec (substring n (length s - n) s))
  || F n' _
```

split itself is trivial to implement in terms of split'. We just ask split' to begin its search with n = length s.

```
Definition split : {∃ s1, ∃ s2, s = s1 ++ s2 ∧ P1 s1 ∧ P2 s2}
  + {∀ s1 s2, s = s1 ++ s2 → ¬ P1 s1 ∨ ¬ P2 s2}.
  refine (Reduce (split' (n := length s) _)); crush; eauto.
Defined.
End split.

Implicit Arguments split [P1 P2].
```

One more helper function will come in handy: dec_star, for implementing another linear search through ways of splitting a string, this time for implementing the Kleene star.

```
Section dec_star.
  Variable P : string → Prop.
  Variable P_dec : ∀ s, {P s} + {¬ P s}.
```

Some new lemmas and hints about the **star** type family are useful here. We omit them here; they are included in the source at this point.

The function dec_star'' implements a single iteration of the star. That is, it tries to find a string prefix matching P, and it calls a parameter function on the remainder of the string.

```
  Section dec_star''.
    Variable n : nat.
```

n is the length of the prefix of s that we have already processed.

```
    Variable P' : string → Prop.
    Variable P'_dec : ∀ n' : nat, n' > n
      → {P' (substring n' (length s - n') s)}
      + {¬ P' (substring n' (length s - n') s)}.
```

When we use dec_star", we will instantiate *P'_dec* with a function for continuing the search for more instances of P in s.

Now we come to dec_star" itself. It takes as an input a natural l that records how much of the string has been searched so far, as we did for split'. The return type expresses that dec_star" is looking for an index into s that splits s into a nonempty prefix and a suffix, such that the prefix satisfies P and the suffix satisfies P'.

```
Definition dec_star" (l : nat)
  : {∃ l', S l' ≤ l
    ∧ P (substring n (S l') s)
    ∧ P' (substring (n + S l') (length s - (n + S l')) s)}
  + {∀ l', S l' ≤ l
    → ¬ P (substring n (S l') s)
    ∨ ¬ P' (substring (n + S l') (length s - (n + S l')) s)}.
  refine (fix F (l : nat) : {∃ l', S l' ≤ l
    ∧ P (substring n (S l') s)
    ∧ P' (substring (n + S l') (length s - (n + S l')) s)}
  + {∀ l', S l' ≤ l
    → ¬ P (substring n (S l') s)
    ∨ ¬ P' (substring (n + S l') (length s - (n + S l')) s)} :=
  match l with
    | O ⇒ _
    | S l' ⇒
      (P_dec (substring n (S l') s) && P'_dec (n' := n + S l') _)
      || F l'
  end); clear F; crush; eauto 7;
  match goal with
    | [ H : ?X ≤ S ?Y ⊢ _ ] ⇒ destruct (eq_nat_dec X (S Y)); crush
  end.
Defined.
End dec_star".
```

The work of dec_star" is nested inside another linear search by dec_star', which provides the final functionality we need, but for arbitrary suffixes of s, rather than just for s overall.

```
Definition dec_star' (n n' : nat) : length s - n' ≤ n
  → {star P (substring n' (length s - n') s)}
  + {¬ star P (substring n' (length s - n') s)}.
  refine (fix F (n n' : nat) : length s - n' ≤ n
    → {star P (substring n' (length s - n') s)}
    + {¬ star P (substring n' (length s - n') s)} :=
    match n with
      | O ⇒ fun _ ⇒ Yes
      | S n" ⇒ fun _ ⇒
        le_gt_dec (length s) n'
        || dec_star" (n := n') (star P)
          (fun n0 _ ⇒ Reduce (F n" n0 _)) (length s - n')
```

```
    end); clear F; crush; eauto;
  match goal with
    | [ H : star _ _ ⊢ _ ] ⇒ apply star_substring_inv in H; crush; eauto
  end;
  match goal with
    | [ H1 : _ < _ - _, H2 : ∀ l' : nat, _ ≤ _ - _ → _ ⊢ _ ] ⇒
      generalize (H2 _ (lt_le_S _ _ H1)); tauto
  end.
Defined.
```

Finally, we have dec_star, with a straightforward implementation.

```
Definition dec_star : {star P s} + {¬ star P s}.
  refine (Reduce (dec_star' (n := length s) 0 _)); crush.
Defined.
End dec_star.
```

With these helper functions completed, the implementation of our matches function is refreshingly straightforward. We only need one small piece of specific tactic work beyond what *crush* does for us.

```
Definition matches P (r : regexp P) s : {P s} + {¬ P s}.
  refine (fix F P (r : regexp P) s : {P s} + {¬ P s} :=
    match r with
      | Char ch ⇒ string_dec s (String ch "")
      | Concat _ _ r1 r2 ⇒ Reduce (split (F _ r1) (F _ r2) s)
      | Or _ _ r1 r2 ⇒ F _ r1 s || F _ r2 s
      | Star _ r ⇒ dec_star _ _ _
    end); crush;
  match goal with
    | [ H : _ ⊢ _ ] ⇒ generalize (H _ _ (refl_equal _))
  end; tauto.
Defined.
```

5. DEPENDENT DATA STRUCTURES

Our red-black tree example from the last section illustrated how dependent types enable static enforcement of data structure invariants. To find interesting uses of dependent data structures, however, we need not look to the favorite examples of data structures and algorithms textbooks. More basic examples like length-indexed and heterogeneous lists come up again and again as the building blocks of dependent programs. There is a surprisingly large design space for this class of data structure, and we will spend this section exploring it.

5.1 More Length-Indexed Lists

We begin with a deeper look at the length-indexed lists that began the last section.

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
```

```
| Nil : ilist O
| Cons : ∀ n, A → ilist n → ilist (S n).
```

We might like to have a certified function for selecting an element of an **ilist** by position. We could do this using subset types and explicit manipulation of proofs, but dependent types let us do it more directly. It is helpful to define a type family **fin**, where **fin** n is isomorphic to $\{m : \textbf{nat} \mid m < n\}$. The type family name stands for "finite."

```
Inductive fin : nat → Set :=
| First : ∀ n, fin (S n)
| Next : ∀ n, fin n → fin (S n).
```

fin essentially makes a more richly-typed copy of the natural numbers. Every element is a First iterated through applying `Next` a number of times that indicates which number is being selected.

Now it is easy to pick a `Prop`-free type for a selection function. As usual, our first implementation attempt will not convince the type checker, and we will attack the deficiencies one at a time.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒ whatGoesHere?
    | Cons _ x ls' ⇒ fun idx ⇒
      match idx with
        | First _ ⇒ x
        | Next _ idx' ⇒ get ls' idx'
      end
  end.
```

We apply the usual wisdom of delaying arguments in `Fixpoint`s so that they may be included in `return` clauses. This still leaves us with a quandary in each of the `match` cases. First, we need to figure out how to take advantage of the contradiction in the Nil case. Every **fin** has a type of the form S n, which cannot unify with the O value that we learn for n in the Nil case. The solution we adopt is another case of `match`-within-`return`.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ _ ⇒ tt
      end
    | Cons _ x ls' ⇒ fun idx ⇒
      match idx with
```

```
      | First _ ⇒ x
      | Next _ idx' ⇒ get ls' idx'
    end
  end.
```

Now the first `match` case type-checks, and we see that the problem with the Cons case is that the pattern-bound variable *idx'* does not have an apparent type compatible with *ls'*. We need to use `match` annotations to make the relationship explicit. Unfortunately, the usual trick of postponing argument binding will not help us here. We need to match on both *ls* and *idx*; one or the other must be matched first. To get around this, we apply the convoy pattern that we met last section. This application is a little more clever than those we saw before; we use the natural number predecessor function pred to express the relationship between the types of these variables.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ _ ⇒ tt
      end
    | Cons _ x ls' ⇒ fun idx ⇒
      match idx in fin n' return ilist (pred n') → A with
        | First _ ⇒ fun _ ⇒ x
        | Next _ idx' ⇒ fun ls' ⇒ get ls' idx'
      end ls'
  end.
```

There is just one problem left with this implementation. Though we know that the local *ls'* in the `Next` case is equal to the original *ls'*, the type-checker is not satisfied that the recursive call to get does not introduce non-termination. We solve the problem by convoy-binding the partial application of get to *ls'*, rather than *ls'* by itself.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ _ ⇒ tt
```

```
    end
  | Cons _ x ls' ⇒ fun idx ⇒
    match idx in fin n' return (fin (pred n') → A) → A with
      | First _ ⇒ fun _ ⇒ x
      | Next _ idx' ⇒ fun get_ls' ⇒ get_ls' idx'
    end (get ls')
end.

End ilist.

Implicit Arguments Nil [A].
Implicit Arguments First [n].
```

A few examples show how to make use of these definitions.

```
Check Cons 0 (Cons 1 (Cons 2 Nil)).
  Cons 0 (Cons 1 (Cons 2 Nil))
    : ilist nat 3

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.
    = 0
    : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).
    = 1
    : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).
    = 2
    : nat
```

Of course, it is critical that invalid programs fail to type-check at all. For instance, here is an "out-of-bounds dereference" error caught statically.

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next (Next First))).

Error: The term "Next (Next (Next First))" has type
"fin (S (S (S ?69))))" while it is expected to have type
"fin 3".
```

Our get function is also quite easy to reason about. We show how with a short example about an analogue to the list *map* function.

```
Section ilist_map.
  Variables A B : Set.
  Variable f : A → B.

  Fixpoint imap n (ls : ilist A n) : ilist B n :=
    match ls with
      | Nil ⇒ Nil
      | Cons _ x ls' ⇒ Cons (f x) (imap ls')
    end.
```

It is easy to prove that get "distributes over" imap calls. The only tricky bit is remembering to use the *dep_destruct* tactic in place of plain `destruct` when faced with a baffling tactic error message.

```
  Theorem get_imap : ∀ n (idx : fin n) (ls : ilist A n),
    get (imap ls) idx = f (get ls idx).
    induction ls; dep_destruct idx; crush.
  Qed.
End ilist_map.
```

With a few basic functions defined over **ilist**s, we can write further useful dependently-typed functions with minimal hassle. For instance, we might want to mix in Section 3's subset type functionality, in building a richly-typed equality test for **ilist**s.

```
Section ilist_eq_dec.
  Variable A : Set.
  Variable A_eq_dec : ∀ a b : A, {a = b} + {a ≠ b}.
```

We parameterize over the carried type A and a decidable equality function for it.

The function itself is easy to write, using the hd function we wrote in the last section, plus the dual tl function, whose implementation we leave as an exercise for the reader. Coding ilist_eq_dec involves little more than repeating the function type inside a `refine` and applying the convenient notations from Section 3. An automated proof destructs **ilist**s of known or partially-known size and leaves the rest to *crush*.

```
  Definition ilist_eq_dec : ∀ (n : nat) (l1 l2 : ilist A n), {l1 = l2} + {l1 ≠ l2}.
    refine (fix F (n : nat) : ∀ l1 l2 : ilist A n, {l1 = l2} + {l1 ≠ l2} :=
      match n with
        | O ⇒ fun _ _ ⇒ Yes
        | S n' ⇒ fun l1 l2 ⇒
          Reduce (A_eq_dec (hd l1) (hd l2) && F n' (tl l1) (tl l2))
      end);
    repeat (match goal with
              | [ l : ilist _ O ⊢ _ ] ⇒ dep_destruct l
              | [ l : ilist _ (S _) ⊢ _ ] ⇒ dep_destruct l
            end; crush).
  Defined.
End ilist_eq_dec.
```

5.2 Heterogeneous Lists

Programmers who move to statically-typed functional languages from "scripting languages" often complain about the requirement that every element of a list have the same type. With fancy type systems, we can partially lift this requirement. We can index a list type with a "type-level" list that explains what type each element of the list should have. This has been done in a variety of ways in Haskell using type classes, and we can do it much more cleanly and directly in Coq.

```
Section hlist.
  Variable A : Type.
  Variable B : A → Type.
```

We parameterize our heterogeneous lists by a type A and an A-indexed type B.

```
Inductive hlist : list A → Type :=
| MNil : hlist nil
| MCons : ∀ (x : A) (ls : list A), B x → hlist ls → hlist (x :: ls).
```

We can implement a variant of the last section's get function for **hlist**s. To get the dependent typing to work out, we will need to index our element selectors by the types of data that they point to.

```
Variable elm : A.

Inductive member : list A → Type :=
| MFirst : ∀ ls, member (elm :: ls)
| MNext : ∀ x ls, member ls → member (x :: ls).
```

Because the element *elm* that we are "searching for" in a list does not change across the constructors of **member**, we simplify our definitions by making *elm* a local variable. In the definition of **member**, we say that *elm* is found in any list that begins with *elm*, and, if removing the first element of a list leaves *elm* present, then *elm* is present in the original list, too. The form looks much like a predicate for list membership, but we purposely define **member** in Type so that we may decompose its values to guide computations.

We can use **member** to adapt our definition of get to *hlists*. The same basic match tricks apply. In the MCons case, we form a two-element convoy, passing both the data element *x* and the recursor for the sublist *mls'* to the result of the inner match. We did not need to do that in get's definition because the types of list elements were not dependent there.

```
Fixpoint hget ls (mls : hlist ls) : member ls → B elm :=
  match mls with
    | MNil ⇒ fun mem ⇒
      match mem in member ls' return (match ls' with
                                        | nil ⇒ B elm
                                        | _ :: _ ⇒ unit
                                      end) with
        | MFirst _ ⇒ tt
        | MNext _ _ _ ⇒ tt
      end
    | MCons _ _ x mls' ⇒ fun mem ⇒
      match mem in member ls'
        return (match ls' with
                  | nil ⇒ Empty_set
                  | x' :: ls'' ⇒ B x' → (member ls'' → B elm) → B elm
                end) with
        | MFirst _ ⇒ fun x _ ⇒ x
        | MNext _ _ mem' ⇒ fun _ get_mls' ⇒ get_mls' mem'
      end x (hget mls')
  end.
End hlist.

Implicit Arguments MNil [A B].
Implicit Arguments MCons [A B x ls].
```

```
Implicit Arguments MFirst [A elm ls].
Implicit Arguments MNext [A elm x ls].
```

By putting the parameters A and B in `Type`, we allow some very higher-order uses. For instance, one use of **hlist** is for the simple heterogeneous lists that we referred to earlier.

```
Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set ⇒ T) someTypes :=
  MCons 5 (MCons true MNil).

Eval simpl in hget someValues MFirst.
    = 5
    : (fun T : Set ⇒ T) nat

Eval simpl in hget someValues (MNext MFirst).
    = true
    : (fun T : Set ⇒ T) bool
```

We can also build indexed lists of pairs in this way.

```
Example somePairs : hlist (fun T : Set ⇒ T × T)%type someTypes :=
  MCons (1, 2) (MCons (true, false) MNil).
```

5.2.1 *A Lambda Calculus Interpreter.* Heterogeneous lists are very useful in implementing interpreters for functional programming languages. Using the types and operations we have already defined, it is trivial to write an interpreter for simply-typed lambda calculus. Our interpreter can alternatively be thought of as a denotational semantics.

We start with an algebraic datatype for types.

```
Inductive type : Set :=
| Unit : type
| Arrow : type → type → type.
```

Now we can define a type family for expressions. An **exp** *ts t* will stand for an expression that has type *t* and whose free variables have types in the list *ts*. We effectively use the de Bruijn variable representation. Variables are represented as **member** values; that is, a variable is more or less a constructive proof that a particular type is found in the type environment.

```
Inductive exp : list type → type → Set :=
| Const : ∀ ts, exp ts Unit
| Var : ∀ ts t, member t ts → exp ts t
| App : ∀ ts dom ran, exp ts (Arrow dom ran) → exp ts dom → exp ts ran
| Abs : ∀ ts dom ran, exp (dom :: ts) ran → exp ts (Arrow dom ran).

Implicit Arguments Const [ts].
```

We write a simple recursive function to translate **type**s into `Set`s.

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Unit ⇒ unit
    | Arrow t1 t2 ⇒ typeDenote t1 → typeDenote t2
```

```
  end.
```

Now it is straightforward to write an expression interpreter. The type of the function, expDenote, tells us that we translate expressions into functions from properly-typed environments to final values. An environment for a free variable list *ts* is simply a **hlist** typeDenote *ts*. That is, for each free variable, the heterogeneous list that is the environment must have a value of the variable's associated type. We use hget to implement the Var case, and we use MCons to extend the environment in the Abs case.

```
Fixpoint expDenote ts t (e : exp ts t) : hlist typeDenote ts → typeDenote t :=
  match e with
    | Const _ ⇒ fun _ ⇒ tt

    | Var _ _ mem ⇒ fun s ⇒ hget s mem
    | App _ _ _ e1 e2 ⇒ fun s ⇒ (expDenote e1 s) (expDenote e2 s)
    | Abs _ _ _ e' ⇒ fun s ⇒ fun x ⇒ expDenote e' (MCons x s)
  end.
```

Like for previous examples, our interpreter is easy to run with `simpl`.

```
Eval simpl in expDenote Const MNil.
     = tt
      : typeDenote Unit

Eval simpl in expDenote (Abs (dom := Unit) (Var MFirst)) MNil.
      = fun x : unit ⇒ x
      : typeDenote (Arrow Unit Unit)

Eval simpl in expDenote (Abs (dom := Unit)
  (Abs (dom := Unit) (Var (MNext MFirst)))) MNil.
      = fun x _ : unit ⇒ x
      : typeDenote (Arrow Unit (Arrow Unit Unit))

Eval simpl in
  expDenote (Abs (dom := Unit) (Abs (dom := Unit) (Var MFirst))) MNil.
      = fun _ x0 : unit ⇒ x0
      : typeDenote (Arrow Unit (Arrow Unit Unit))

Eval simpl in expDenote (App (Abs (Var MFirst)) Const) MNil.
      = tt
      : typeDenote Unit
```

We are starting to develop the tools behind dependent typing's amazing advantage over alternative approaches in several important areas. Here, we have implemented complete syntax, typing rules, and evaluation semantics for simply-typed lambda calculus without even needing to define a syntactic substitution operation. We did it all without a single line of proof, and our implementation is manifestly executable. More common approaches to language formalization often state and prove explicit theorems about type safety of languages. In the above example, we got type safety, termination, and other meta-theorems for free, by reduction to CIC, which we know has those properties.

5.3 Recursive Type Definitions

There is another style of datatype definition that leads to much simpler definitions of the get and hget definitions above. Because Coq supports "type-level computation," we can redo our inductive definitions as *recursive* definitions.

```
Section filist.
  Variable A : Set.

  Fixpoint filist (n : nat) : Set :=
    match n with
      | O ⇒ unit
      | S n' ⇒ A × filist n'
    end%type.
```

We say that a list of length 0 has no contents, and a list of length S *n'* is a pair of a data value and a list of length *n'*.

```
  Fixpoint ffin (n : nat) : Set :=
    match n with
      | O ⇒ Empty_set
      | S n' ⇒ option (ffin n')
    end.
```

We express that there are no index values when $n = \mathsf{O}$, by defining such indices as type **Empty_set**; and we express that, at $n = \mathsf{S}\ n'$, there is a choice between picking the first element of the list (represented as None) or choosing a later element (represented by Some *idx*, where *idx* is an index into the list tail).

```
  Fixpoint fget (n : nat) : filist n → ffin n → A :=
    match n with
      | O ⇒ fun _ idx ⇒ match idx with end
      | S n' ⇒ fun ls idx ⇒
        match idx with
          | None ⇒ fst ls
          | Some idx' ⇒ fget n' (snd ls) idx'
        end
    end.
```

Our new get implementation needs only one dependent `match`, and its annotation is inferred for us. Our choices of data structure implementations lead to just the right typing behavior for this new definition to work out.

```
End filist.
```

Heterogeneous lists are a little trickier to define with recursion, but we then reap similar benefits in simplicity of use.

```
Section fhlist.
  Variable A : Type.
  Variable B : A → Type.

  Fixpoint fhlist (ls : list A) : Type :=
    match ls with
      | nil ⇒ unit
```

```
    | x :: ls' ⇒ B x × fhlist ls'
  end%type.
```

The definition of fhlist follows the definition of filist, with the added wrinkle of dependently-typed data elements.

```
Variable elm : A.

Fixpoint fmember (ls : list A) : Type :=
  match ls with
    | nil ⇒ Empty_set
    | x :: ls' ⇒ (x = elm) + fmember ls'
  end%type.
```

The definition of fmember follows the definition of ffin. Empty lists have no members, and member types for nonempty lists are built by adding one new option to the type of members of the list tail. While for *index* we needed no new information associated with the option that we add, here we need to know that the head of the list equals the element we are searching for. We express that with a sum type whose left branch is the appropriate equality proposition. Since we define fmember to live in `Type`, we can insert `Prop` types as needed, because `Prop` is a subtype of `Type`.

We know all of the tricks needed to write a first attempt at a get function for fhlists.

```
Fixpoint fhget (ls : list A) : fhlist ls → fmember ls → B elm :=
  match ls with
    | nil ⇒ fun _ idx ⇒ match idx with end
    | _ :: ls' ⇒ fun mls idx ⇒
      match idx with
        | inl _ ⇒ fst mls
        | inr idx' ⇒ fhget ls' (snd mls) idx'
      end
  end.
```

Only one problem remains. The expression fst *mls* is not known to have the proper type. To demonstrate that it does, we need to use the proof available in the inl case of the inner `match`.

```
Fixpoint fhget (ls : list A) : fhlist ls → fmember ls → B elm :=
  match ls with
    | nil ⇒ fun _ idx ⇒ match idx with end
    | _ :: ls' ⇒ fun mls idx ⇒
      match idx with
        | inl pf ⇒ match pf with
                     | refl_equal ⇒ fst mls
                   end
        | inr idx' ⇒ fhget ls' (snd mls) idx'
      end
  end.
```

By pattern-matching on the equality proof *pf*, we make that equality known to the type-checker. Exactly why this works can be seen by studying the definition of equality.

```
Print eq.
Inductive eq (A : Type) (x : A) : A → Prop := refl_equal : x = x
```

In a proposition $x = y$, we see that x is a parameter and y is a regular argument. The type of the constructor refl_equal shows that y can only ever be instantiated to x. Thus, within a pattern-match with refl_equal, occurrences of y can be replaced with occurrences of x for typing purposes.

```
End fhlist.

Implicit Arguments fhget [A B elm ls].
```

5.4 Data Structures as Index Functions

Indexed lists can be useful in defining other inductive types with constructors that take variable numbers of arguments. In this section, we consider parameterized trees with arbitrary branching factor.

```
Section tree.
  Variable A : Set.

  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, ilist tree n → tree.
End tree.
```

Every Node of a **tree** has a natural number argument, which gives the number of child trees in the second argument, typed with **ilist**. We can define two operations on trees of naturals: summing their elements and incrementing their elements. It is useful to define a generic fold function on **ilist**s first.

```
Section ifoldr.
  Variables A B : Set.
  Variable f : A → B → B.
  Variable i : B.

  Fixpoint ifoldr n (ls : ilist A n) : B :=
    match ls with
      | Nil ⇒ i
      | Cons _ x ls' ⇒ f x (ifoldr ls')
    end.
End ifoldr.

Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ ls ⇒ ifoldr (fun t' n ⇒ sum t' + n) O ls
  end.

Fixpoint inc (t : tree nat) : tree nat :=
```

```
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ ls ⇒ Node (imap inc ls)
  end.
```

Now we might like to prove that inc does not decrease a tree's sum.

```
Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  induction t; crush.

  n : nat
  i : ilist (tree nat) n
  ============================
   ifoldr (fun (t' : tree nat) (n0 : nat) ⇒ sum t' + n0) 0 (imap inc i) ≥
   ifoldr (fun (t' : tree nat) (n0 : nat) ⇒ sum t' + n0) 0 i
```

We are left with a single subgoal which does not seem provable directly. The problem is that Coq generates induction principles using a best-effort heuristic; sometimes more permissive principles are possible, and the principle we are getting here for **tree** is very weak.

```
  Check tree_ind.
  tree_ind
      : ∀ (A : Set) (P : tree A → Prop),
          (∀ a : A, P (Leaf a)) →
          (∀ (n : nat) (i : ilist (tree A) n), P (Node i)) →
          ∀ t : tree A, P t
```

For the Node case, we get no inductive hypothesis. We could prove and apply our own induction principle, which is an informative exercise, but there is an easier way, if we are willing to alter the definition of **tree**.

```
Abort.
```

Reset tree.

First, let us try using our recursive definition of **ilist**s instead of the inductive version.

```
Section tree.
  Variable A : Set.

  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, filist tree n → tree.

Error: Non strictly positive occurrence of "tree" in
 "forall n : nat, filist tree n -> tree"
```

Coq inductive definitions support a special-case rule for *nested* datatypes, where one datatype is defined in terms of an instantiation of another. Unfortunately, this only works with nested uses of other *inductive* types, which could be replaced with uses of new mutually-inductive types. We defined filist recursively, so it may not be used for nested recursion.

Our final solution uses a technique called reflexive types, where an inductive type is defined in terms of functions returning values in that same type. Instead of merely using **fin** to get elements out of **ilist**, we can *define* **ilist** in terms of **fin**. For the reasons outlined above, it turns out to be easier to work with ffin in place of **fin**.

```
Inductive tree : Set :=
| Leaf : A → tree
| Node : ∀ n, (ffin n → tree) → tree.
```

A Node is indexed by a natural number n, and the node's n children are represented as a function from ffin n to trees, which is isomorphic to the **ilist**-based representation that we used above.

```
End tree.

Implicit Arguments Node [A n].
```

We can redefine sum and inc for our new **tree** type. Again, it is useful to define a generic fold function first. This time, it takes in a function whose range is some ffin type, and it folds another function over the results of calling the first function at every possible ffin value.

```
Section rifoldr.
  Variables A B : Set.
  Variable f : A → B → B.
  Variable i : B.

  Fixpoint rifoldr (n : nat) : (ffin n → A) → B :=
    match n with
      | O ⇒ fun _ ⇒ i
      | S n' ⇒ fun get ⇒ f (get None) (rifoldr n' (fun idx ⇒ get (Some idx)))
    end.
End rifoldr.

Implicit Arguments rifoldr [A B n].

Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ f ⇒ rifoldr plus O (fun idx ⇒ sum (f idx))
  end.

Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ f ⇒ Node (fun idx ⇒ inc (f idx))
  end.
```

Now we are ready to prove the theorem where we got stuck before. We will not need to define any new induction principle, but it *will* be helpful to prove some lemmas.

```
Lemma plus_ge : ∀ x1 y1 x2 y2,
  x1 ≥ x2
```

```
  → y1 ≥ y2
  → x1 + y1 ≥ x2 + y2.
  crush.
Qed.

Lemma sum_inc' : ∀ n (f1 f2 : ffin n → nat),
  (∀ idx, f1 idx ≥ f2 idx)
  → rifoldr plus 0 f1 ≥ rifoldr plus 0 f2.
  Hint Resolve plus_ge.

  induction n; crush.
Qed.

Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  Hint Resolve sum_inc'.

  induction t; crush.
Qed.
```

Even if Coq would generate complete induction principles automatically for nested inductive definitions like the one we started with, there would still be advantages to using this style of reflexive encoding. We see one of those advantages in the definition of inc, where we did not need to use any kind of auxiliary function. In general, reflexive encodings often admit direct implementations of operations that would require recursion if performed with more traditional inductive data structures.

5.4.1 *Another Interpreter Example.* We develop another example of variable-arity constructors, in the form of optimization of a small expression language with a construct like Scheme's `cond`. Each of our conditional expressions takes a list of pairs of boolean tests and bodies. The value of the conditional comes from the body of the first test in the list to evaluate to true. To simplify the interpreter we will write, we force each conditional to include a final, default case.

```
Inductive type' : Type := Nat | Bool.

Inductive exp' : type' → Type :=
| NConst : nat → exp' Nat
| Plus : exp' Nat → exp' Nat → exp' Nat
| Eq : exp' Nat → exp' Nat → exp' Bool

| BConst : bool → exp' Bool
| Cond : ∀ n t, (ffin n → exp' Bool)
  → (ffin n → exp' t) → exp' t → exp' t.
```

A Cond is parameterized by a natural n, which tells us how many cases this conditional has. The test expressions are represented with a function of type ffin n → **exp'** Bool, and the bodies are represented with a function of type ffin n → **exp'** t, where t is the overall type. The final **exp'** t argument is the default case.

We start implementing our interpreter with a standard type denotation function.

```
Definition type'Denote (t : type') : Set :=
  match t with
```

```
    | Nat ⇒ nat
    | Bool ⇒ bool
  end.
```

To implement the expression interpreter, it is useful to have the following function that implements the functionality of Cond without involving any syntax.

```
Section cond.
  Variable A : Set.
  Variable default : A.

  Fixpoint cond (n : nat) : (ffin n → bool) → (ffin n → A) → A :=
    match n with
      | O ⇒ fun _ _ ⇒ default
      | S n' ⇒ fun tests bodies ⇒
        if tests None
          then bodies None
          else cond n'
            (fun idx ⇒ tests (Some idx))
            (fun idx ⇒ bodies (Some idx))
    end.
End cond.

Implicit Arguments cond [A n].
```

Now the expression interpreter is straightforward to write.

```
Fixpoint exp'Denote t (e : exp' t) : type'Denote t :=
  match e with
    | NConst n ⇒ n
    | Plus e1 e2 ⇒ exp'Denote e1 + exp'Denote e2
    | Eq e1 e2 ⇒
      if eq_nat_dec (exp'Denote e1) (exp'Denote e2) then true else false

    | BConst b ⇒ b
    | Cond _ _ tests bodies default ⇒
      cond
      (exp'Denote default)
      (fun idx ⇒ exp'Denote (tests idx))
      (fun idx ⇒ exp'Denote (bodies idx))
  end.
```

We will implement a constant-folding function that optimizes conditionals, removing cases with known-false tests and cases that come after known-true tests. A function cfoldCond implements the heart of this logic. The convoy pattern is used again near the end of the implementation.

```
Section cfoldCond.
  Variable t : type'.
  Variable default : exp' t.

  Fixpoint cfoldCond (n : nat)
    : (ffin n → exp' Bool) → (ffin n → exp' t) → exp' t :=
```

```
    match n with
      | O ⇒ fun _ _ ⇒ default
      | S n' ⇒ fun tests bodies ⇒
        match tests None return _ with
          | BConst true ⇒ bodies None
          | BConst false ⇒ cfoldCond n'
            (fun idx ⇒ tests (Some idx))
            (fun idx ⇒ bodies (Some idx))
          | _ ⇒
            let e := cfoldCond n'
              (fun idx ⇒ tests (Some idx))
              (fun idx ⇒ bodies (Some idx)) in
            match e in exp' t return exp' t → exp' t with
              | Cond n _ tests' bodies' default' ⇒ fun body ⇒
                Cond
                (S n)
                (fun idx ⇒ match idx with
                              | None ⇒ tests None
                              | Some idx ⇒ tests' idx
                            end)
                (fun idx ⇒ match idx with
                              | None ⇒ body
                              | Some idx ⇒ bodies' idx
                            end)
                default'
              | e ⇒ fun body ⇒
                Cond
                1
                (fun _ ⇒ tests None)
                (fun _ ⇒ body)
                e
            end (bodies None)
        end
    end.
End cfoldCond.

Implicit Arguments cfoldCond [t n].
```

Like for the interpreters, most of the action was in this helper function, and cfold itself is easy to write.

```
Fixpoint cfold t (e : exp' t) : exp' t :=
  match e with
    | NConst n ⇒ NConst n
    | Plus e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
      match e1', e2' return _ with
        | NConst n1, NConst n2 ⇒ NConst (n1 + n2)
```

```
      | _, _ ⇒ Plus e1' e2'
    end
  | Eq e1 e2 ⇒
    let e1' := cfold e1 in
    let e2' := cfold e2 in
    match e1', e2' return _ with
      | NConst n1, NConst n2 ⇒
        BConst (if eq_nat_dec n1 n2 then true else false)
      | _, _ ⇒ Eq e1' e2'
    end

  | BConst b ⇒ BConst b
  | Cond _ _ tests bodies default ⇒
    cfoldCond
    (cfold default)
    (fun idx ⇒ cfold (tests idx))
    (fun idx ⇒ cfold (bodies idx))
end.
```

To prove our final correctness theorem, it is useful to know that cfoldCond preserves expression meanings. This lemma formalizes that property. The proof is a standard mostly-automated one, with the only wrinkle being a guided instantiation of the quantifiers in the induction hypothesis.

```
Lemma cfoldCond_correct : ∀ t (default : exp' t)
  n (tests : ffin n → exp' Bool) (bodies : ffin n → exp' t),
  exp'Denote (cfoldCond default tests bodies)
  = exp'Denote (Cond n tests bodies default).
  induction n; crush;
    match goal with
      | [ IHn : ∀ tests bodies, _, tests : _ → _, bodies : _ → _ ⊢ _ ] ⇒
        specialize (IHn (fun idx ⇒ tests (Some idx))
          (fun idx ⇒ bodies (Some idx)))
    end;
    repeat (match goal with
              | [ ⊢ context[match ?E with
                             | NConst _ ⇒ _
                             | Plus _ _ ⇒ _
                             | Eq _ _ ⇒ _
                             | BConst _ ⇒ _
                             | Cond _ _ _ _ _ ⇒ _
                           end] ] ⇒ dep_destruct E
              | [ ⊢ context[if ?B then _ else _] ] ⇒ destruct B
            end; crush).
Qed.
```

It is also useful to know that the result of a call to cond is not changed by substituting new tests and bodies functions, so long as the new functions have the same input-output behavior as the old. It turns out that, in Coq, it is not

possible to prove in general that functions related in this way are equal. It suffices to prove that the particular function cond is *extensional*; that is, it is unaffected by substitution of functions with input-output equivalents.

```
Lemma cond_ext : ∀ (A : Set) (default : A) n (tests tests' : ffin n → bool)
  (bodies bodies' : ffin n → A),
  (∀ idx, tests idx = tests' idx)
  → (∀ idx, bodies idx = bodies' idx)
  → cond default tests bodies
  = cond default tests' bodies'.
  induction n; crush;
    match goal with
      | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
    end; crush.
Qed.
```

Now the final theorem is easy to prove. We add our two lemmas as hints and perform standard automation with pattern-matching of subterms to destruct.

```
Theorem cfold_correct : ∀ t (e : exp' t),
  exp'Denote (cfold e) = exp'Denote e.
  Hint Rewrite cfoldCond_correct : cpdt.
  Hint Resolve cond_ext.

  induction e; crush;
    repeat (match goal with
              | [ ⊢ context[cfold ?E] ] ⇒ dep_destruct (cfold E)
            end; crush).
Qed.
```

5.5 Choosing Between Representations

It is not always clear which of these representation techniques to apply in a particular situation, but I will try to summarize the pros and cons of each.

Inductive types are often the most pleasant to work with, after someone has spent the time implementing some basic library functions for them, using fancy `match` annotations. Many aspects of Coq's logic and tactic support are specialized to deal with inductive types, and you may miss out if you use alternate encodings.

Recursive types usually involve much less initial effort, but they can be less convenient to use with proof automation. For instance, the `simpl` tactic (which is among the ingredients in *crush*) will sometimes be overzealous in simplifying uses of functions over recursive types. Consider a call get l f, where variable l has type filist A (S n). The type of l would be simplified to an explicit pair type. In a proof involving many recursive types, this kind of unhelpful "simplification" can lead to rapid bloat in the sizes of subgoals. Even worse, it can prevent syntactic pattern-matching, like in cases where filist is expected but a pair type is found in the "simplified" version.

Another disadvantage of recursive types is that they only apply to type families whose indices determine their "skeletons." This is not true for all data structures; a good counterexample comes from the richly-typed programming language syntax

types we have used several times so far. The fact that a piece of syntax has type Nat tells us nothing about the tree structure of that syntax.

Reflexive encodings of data types are seen relatively rarely. As our examples demonstrated, manipulating index values manually can lead to hard-to-read code. A normal inductive type is generally easier to work with, once someone has gone through the trouble of implementing an induction principle manually. For small developments, avoiding that kind of coding can justify the use of reflexive data structures. There are also some useful instances of co-inductive definitions with nested data structures (e.g., lists of values in the co-inductive type) that can only be deconstructed effectively with reflexive encoding of the nested structures.

6. REASONING ABOUT EQUALITY PROOFS

In traditional mathematics, the concept of equality is usually taken as a given. On the other hand, in type theory, equality is a very contentious subject. There are at least three different notions of equality that are important, and researchers are actively investigating new definitions of what it means for two terms to be equal. Even once we fix a notion of equality, there are inevitably tricky issues that arise in proving properties of programs that manipulate equality proofs explicitly. In this section, I will focus on design patterns for circumventing these tricky issues, and I will introduce the different notions of equality as they are germane.

6.1 The Definitional Equality

We have seen many examples so far where proof goals follow "by computation." That is, we apply computational reduction rules to reduce the goal to a normal form, at which point it follows trivially. Exactly when this works and when it does not depends on the details of Coq's *definitional equality*. This is an untyped binary relation appearing in the formal metatheory of CIC. CIC contains a typing rule allowing the conclusion $E : T$ from the premise $E : T'$ and a proof that T and T' are definitionally equal.

The *cbv* tactic will help us illustrate the rules of Coq's definitional equality. We redefine the natural number predecessor function in a somewhat convoluted way and construct a manual proof that it returns 0 when applied to 1.

```
Definition pred' (x : nat) :=
  match x with
    | O ⇒ O
    | S n' ⇒ let y := n' in y
  end.

Theorem reduce_me : pred' 1 = 0.
```

CIC follows the traditions of lambda calculus in associating reduction rules with Greek letters. Coq can certainly be said to support the familiar alpha reduction rule, which allows capture-avoiding renaming of bound variables, but we never need to apply alpha explicitly, since Coq uses a de Bruijn representation that encodes terms canonically.

The delta rule is for unfolding global definitions. We can use it here to unfold the definition of pred'. We do this with the *cbv* tactic, which takes a list of reduction

rules and makes as many call-by-value reduction steps as possible, using only those rules. There is an analogous tactic *lazy* for call-by-need reduction.

cbv delta.

```
============================
 (fun x : nat ⇒ match x with
                | 0 ⇒ 0
                | S n' ⇒ let y := n' in y
                end) 1 = 0
```

At this point, we want to apply the famous beta reduction of lambda calculus, to simplify the application of a known function abstraction.

cbv beta.

```
============================
 match 1 with
 | 0 ⇒ 0
 | S n' ⇒ let y := n' in y
 end = 0
```

Next on the list is the iota reduction, which simplifies a single `match` term by determining which pattern matches.

cbv iota.

```
============================
 (fun n' : nat ⇒ let y := n' in y) 0 = 0
```

Now we need another beta reduction.

cbv beta.

```
============================
 (let y := 0 in y) = 0
```

The final reduction rule is zeta, which replaces a `let` expression by its body with the appropriate term substituted.

cbv zeta.

```
============================
 0 = 0
```

```
reflexivity.
Qed.
```

The standard *eq* relation is critically dependent on the definitional equality. *eq* is often called a *propositional equality*, because it reifies definitional equality as a proposition that may or may not hold. Standard axiomatizations of an equality

predicate in first-order logic define equality in terms of properties it has, like reflexivity, symmetry, and transitivity. In contrast, for *eq* in Coq, those properties are implicit in the properties of the definitional equality, which are built into CIC's metatheory and the implementation of Gallina. We could add new rules to the definitional equality, and *eq* would keep its definition and methods of use.

This all may make it sound like the choice of *eq*'s definition is unimportant. To the contrary, in this section, we will see examples where alternate definitions may simplify proofs. Before that point, I will introduce proof methods for goals that use proofs of the standard propositional equality "as data."

6.2 Heterogeneous Lists Revisited

One of our example dependent data structures from the last section was heterogeneous lists and their associated "cursor" type. The recursive version poses some special challenges related to equality proofs, since it uses such proofs in its definition of **member** types.

```
Section fhlist.
  Variable A : Type.
  Variable B : A → Type.

  Fixpoint fhlist (ls : list A) : Type :=
    match ls with
      | nil ⇒ unit
      | x :: ls' ⇒ B x × fhlist ls'
    end%type.

  Variable elm : A.

  Fixpoint fmember (ls : list A) : Type :=
    match ls with
      | nil ⇒ Empty_set
      | x :: ls' ⇒ (x = elm) + fmember ls'
    end%type.

  Fixpoint fhget (ls : list A) : fhlist ls → fmember ls → B elm :=
    match ls return fhlist ls → fmember ls → B elm with
      | nil ⇒ fun _ idx ⇒ match idx with end
      | _ :: ls' ⇒ fun mls idx ⇒
        match idx with
          | inl pf ⇒ match pf with
                       | refl_equal ⇒ fst mls
                     end
          | inr idx' ⇒ fhget ls' (snd mls) idx'
        end
    end.
End fhlist.

Implicit Arguments fhget [A B elm ls].
```

We can define a *map*-like function for fhlists.

```
Section fhlist_map.
```

```
Variables A : Type.
Variables B C : A → Type.
Variable f : ∀ x, B x → C x.

Fixpoint fhmap (ls : list A) : fhlist B ls → fhlist C ls :=
  match ls return fhlist B ls → fhlist C ls with
    | nil ⇒ fun _ ⇒ tt
    | _ :: _ ⇒ fun hls ⇒ (f (fst hls), fhmap _ (snd hls))
  end.

Implicit Arguments fhmap [ls].
```

For the inductive versions of the **ilist** definitions, we proved a lemma about the interaction of get and imap. It was a strategic choice not to attempt such a proof for the definitions that we just gave, because that sets us on a collision course with the problems that are the subject of this section.

```
Variable elm : A.

Theorem get_imap : ∀ ls (mem : fmember elm ls) (hls : fhlist B ls),
  fhget (fhmap hls) mem = f (fhget hls mem).
  induction ls; crush.
```

Part of our single remaining subgoal is:

```
a0 : a = elm
============================
 match a0 in (_ = a2) return (C a2) with
 | refl_equal ⇒ f a1
 end = f match a0 in (_ = a2) return (B a2) with
         | refl_equal ⇒ a1
         end
```

This seems like a trivial enough obligation. The equality proof *a0* must be refl_equal, since that is the only constructor of *eq*. Therefore, both the matches reduce to the point where the conclusion follows by reflexivity.

```
    destruct a0.
User error: Cannot solve a second-order unification problem
```

This is one of Coq's standard error messages for informing us that its heuristics for attempting an instance of an undecidable problem about dependent typing have failed. We might try to nudge things in the right direction by stating the lemma that we believe makes the conclusion trivial.

```
    assert (a0 = refl_equal _).
The term "refl_equal ?98" has type "?98 = ?98"
 while it is expected to have type "a = elm"
```

In retrospect, the problem is not so hard to see. Reflexivity proofs only show $x = x$ for particular values of x, whereas here we are thinking in terms of a proof of

$a = elm$, where the two sides of the equality are not equal syntactically. Thus, the essential lemma we need does not even type-check!

Is it time to throw in the towel? Luckily, the answer is "no." In this section, we will see several useful patterns for proving obligations like this.

For this particular example, the solution is surprisingly straightforward. `destruct` has a simpler sibling `case` which should behave identically for any inductive type with one constructor of no arguments.

```
  case a0.

============================
 f a1 = f a1
```

It seems that `destruct` was trying to be too smart for its own good.

```
  reflexivity.
Qed.
```

It will be helpful to examine the proof terms generated by this sort of strategy. A simpler example illustrates what is going on.

```
Lemma lemma1 : ∀ x (pf : x = elm), O = match pf with refl_equal ⇒ O end.
  simple destruct pf; reflexivity.
Qed.
```

simple `destruct` *pf* is a convenient form for applying `case`. It runs `intro` to bring into scope all quantified variables up to its argument.

```
Print lemma1.
lemma1 =
fun (x : A) (pf : x = elm) ⇒
match pf as e in (_ = y) return (0 = match e with
                                     | refl_equal ⇒ 0
                                     end) with
| refl_equal ⇒ refl_equal 0
end
     : ∀ (x : A) (pf : x = elm), 0 = match pf with
                                     | refl_equal ⇒ 0
                                     end
```

Using what we know about shorthands for `match` annotations, we can write this proof in shorter form manually.

```
Definition lemma1' :=
  fun (x : A) (pf : x = elm) ⇒
    match pf return (0 = match pf with
                         | refl_equal ⇒ 0
                         end) with
      | refl_equal ⇒ refl_equal 0
    end.
```

Surprisingly, what seems at first like a *simpler* lemma is harder to prove.

```
Lemma lemma2 : ∀ (x : A) (pf : x = x), O = match pf with refl_equal ⇒ O end.

  simple destruct pf.
User error: Cannot solve a second-order unification problem

Abort.
```

Nonetheless, we can adapt the last manual proof to handle this theorem.

```
Definition lemma2 :=
  fun (x : A) (pf : x = x) ⇒
    match pf return (0 = match pf with
                           | refl_equal ⇒ 0
                         end) with
      | refl_equal ⇒ refl_equal 0
    end.
```

We can try to prove a lemma that would simplify proofs of many facts like lemma2:

```
Lemma lemma3 : ∀ (x : A) (pf : x = x), pf = refl_equal x.

  simple destruct pf.
User error: Cannot solve a second-order unification problem

Abort.
```

This time, even our manual attempt fails.

```
Definition lemma3' :=
  fun (x : A) (pf : x = x) ⇒
    match pf as pf' in (_ = x') return (pf' = refl_equal x') with
      | refl_equal ⇒ refl_equal _
    end.
The term "refl_equal x'" has type "x' = x'" while it is expected to have type
 "x = x'"
```

The type error comes from our `return` annotation. In that annotation, the `as`-bound variable *pf'* has type $x = x'$, referring to the `in`-bound variable *x'*. To do a dependent `match`, we *must* choose a fresh name for the second argument of *eq*. We are just as constrained to use the "real" value x for the first argument. Thus, within the `return` clause, the proof we are matching on *must* equate two non-matching terms, which makes it impossible to equate that proof with reflexivity.

Nonetheless, it turns out that, with one catch, we *can* prove this lemma.

```
Lemma lemma3 : ∀ (x : A) (pf : x = x), pf = refl_equal x.
  intros; apply UIP_refl.
Qed.

Check UIP_refl.
UIP_refl
     : ∀ (U : Type) (x : U) (p : x = x), p = refl_equal x
```

UIP_refl comes from the *Eqdep* module of the standard library. Do the Coq authors know of some clever trick for building such proofs that we have not seen yet? If they do, they did not use it for this proof. Rather, the proof is based on an *axiom*.

```
  Print eq_rect_eq.
eq_rect_eq =
fun U : Type ⇒ Eq_rect_eq.eq_rect_eq U
     : ∀ (U : Type) (p : U) (Q : U → Type) (x : Q p) (h : p = p),
       x = eq_rect p Q x p h
```

eq_rect_eq states a "fact" that seems like common sense, once the notation is deciphered. *eq_rect* is the automatically-generated recursion principle for *eq*. Calling *eq_rect* is another way of `match`ing on an equality proof. The proof we match on is the argument *h*, and *x* is the body of the `match`. *eq_rect_eq* just says that `match`es on proofs of $p = p$, for any *p*, are superfluous and may be removed.

Perhaps surprisingly, we cannot prove *eq_rect_eq* from within Coq. This proposition is introduced as an axiom; that is, a proposition asserted as true without proof. We cannot assert just any statement without proof. Adding **False** as an axiom would allow us to prove any proposition, for instance, defeating the point of using a proof assistant. In general, we need to be sure that we never assert *inconsistent* sets of axioms. A set of axioms is inconsistent if its conjunction implies **False**. For the case of *eq_rect_eq*, consistency has been verified outside of Coq via "informal" metatheory.

This axiom is equivalent to another that is more commonly known and mentioned in type theory circles.

```
  Print Streicher_K.
Streicher_K =
fun U : Type ⇒ UIP_refl__Streicher_K U (UIP_refl U)
     : ∀ (U : Type) (x : U) (P : x = x → Prop),
       P (refl_equal x) → ∀ p : x = x, P p
```

This is the unfortunately-named "Streicher's axiom K," which says that a predicate on properly-typed equality proofs holds of all such proofs if it holds of reflexivity.

```
End fhlist_map.
```

6.3 Type-Casts in Theorem Statements

Sometimes we need to use tricks with equality just to state the theorems that we care about. To illustrate, we start by defining a concatenation function for fhlists.

```
Section fhapp.
  Variable A : Type.
  Variable B : A → Type.

  Fixpoint fhapp (ls1 ls2 : list A)
    : fhlist B ls1 → fhlist B ls2 → fhlist B (ls1 ++ ls2) :=
```

```
    match ls1 with
      | nil ⇒ fun _ hls2 ⇒ hls2
      | _ :: _ ⇒ fun hls1 hls2 ⇒ (fst hls1, fhapp _ _ (snd hls1) hls2)
    end.

  Implicit Arguments fhapp [ls1 ls2].
```

We might like to prove that fhapp is associative.

```
  Theorem fhapp_ass : ∀ ls1 ls2 ls3
    (hls1 : fhlist B ls1) (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3) = fhapp (fhapp hls1 hls2) hls3.
The term
 "fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3) (fhapp (ls1:=ls1) (ls2:=ls2) hls1 hls2)
    hls3" has type "fhlist B ((ls1 ++ ls2) ++ ls3)"
 while it is expected to have type "fhlist B (ls1 ++ ls2 ++ ls3)"
```

This first cut at the theorem statement does not even type-check. We know that the two fhlist types appearing in the error message are always equal, by associativity of normal list append, but this fact is not apparent to the type checker. This stems from the fact that Coq's equality is *intensional*, in the sense that type equality theorems can never be applied after the fact to get a term to type-check. Instead, we need to make use of equality explicitly in the theorem statement.

```
  Theorem fhapp_ass : ∀ ls1 ls2 ls3
    (pf : (ls1 ++ ls2) ++ ls3 = ls1 ++ (ls2 ++ ls3))
    (hls1 : fhlist B ls1) (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3)
    = match pf in (_ = ls) return fhlist _ ls with
        | refl_equal ⇒ fhapp (fhapp hls1 hls2) hls3
      end.
    induction ls1; crush.
```

The first remaining subgoal looks trivial enough:

```
  ============================
   fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3 =
   match pf in (_ = ls) return (fhlist B ls) with
   | refl_equal ⇒ fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3
   end
```

We can try what worked in previous examples.

```
    case pf.
User error: Cannot solve a second-order unification problem
```

It seems we have reached another case where it is unclear how to use a dependent match to implement case analysis on our proof. The UIP_refl theorem can come to our rescue again.

```
    rewrite (UIP_refl _ _ pf).
```

```
============================
 fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3 =
 fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3


  reflexivity.
```

Our second subgoal is trickier.

```
pf : a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3
============================
 (a0,
 fhapp (ls1:=ls1) (ls2:=ls2 ++ ls3) b
   (fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3)) =
 match pf in (_ = ls) return (fhlist B ls) with
 | refl_equal ⇒
     (a0,
     fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
       (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3)
 end


   rewrite (UIP_refl _ _ pf).
The term "pf" has type "a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3"
 while it is expected to have type "?556 = ?556"
```

We can only apply UIP_refl on proofs of equality with syntactically equal operands, which is not the case of *pf* here. We will need to manipulate the form of this subgoal to get us to a point where we may use UIP_refl. A first step is obtaining a proof suitable to use in applying the induction hypothesis. Inversion on the structure of *pf* is sufficient for that.

```
   injection pf; intro pf'.

pf : a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3
pf' : (ls1 ++ ls2) ++ ls3 = ls1 ++ ls2 ++ ls3
============================
 (a0,
 fhapp (ls1:=ls1) (ls2:=ls2 ++ ls3) b
   (fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3)) =
 match pf in (_ = ls) return (fhlist B ls) with
 | refl_equal ⇒
     (a0,
     fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
       (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3)
 end
```

Now we can rewrite using the inductive hypothesis.

```
   rewrite (IHls1 _ _ pf').
```

```
============================
 (a0,
 match pf' in (_ = ls) return (fhlist B ls) with
 | refl_equal ⇒
     fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
       (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3
 end) =
 match pf in (_ = ls) return (fhlist B ls) with
 | refl_equal ⇒
     (a0,
     fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
       (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3)
 end
```

We have made an important bit of progress, as now only a single call to fhapp appears in the conclusion, repeated twice. Trying case analysis on our proofs still will not work, but there is a move we can make to enable it. Not only does just one call to fhapp matter to us now, but it also *does not matter what the result of the call is*. In other words, the subgoal should remain true if we replace this fhapp call with a fresh variable. The `generalize` tactic helps us do exactly that.

```
generalize (fhapp (fhapp b hls2) hls3).
```

```
∀ f : fhlist B ((ls1 ++ ls2) ++ ls3),
(a0,
match pf' in (_ = ls) return (fhlist B ls) with
| refl_equal ⇒ f
end) =
match pf in (_ = ls) return (fhlist B ls) with
| refl_equal ⇒ (a0, f)
end
```

The conclusion has gotten markedly simpler. It seems counterintuitive that we can have an easier time of proving a more general theorem, but that is exactly the case here and for many other proofs that use dependent types heavily. Speaking informally, the reason why this kind of activity helps is that `match` annotations only support variables in certain positions. By reducing more elements of a goal to variables, built-in tactics can have more success building `match` terms under the hood.

In this case, it is helpful to generalize over our two proofs as well.

```
generalize pf pf'.
```

```
∀ (pf0 : a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3)
  (pf'0 : (ls1 ++ ls2) ++ ls3 = ls1 ++ ls2 ++ ls3)
  (f : fhlist B ((ls1 ++ ls2) ++ ls3)),
(a0,
```

```
match pf'0 in (_ = ls) return (fhlist B ls) with
| refl_equal ⇒ f
end) =
match pf0 in (_ = ls) return (fhlist B ls) with
| refl_equal ⇒ (a0, f)
end
```

To an experienced dependent types hacker, the appearance of this goal term calls for a celebration. The formula has a critical property that indicates that our problems are over. To get our proofs into the right form to apply UIP_refl, we need to use associativity of list append to rewrite their types. We could not do that before because other parts of the goal require the proofs to retain their original types. In particular, the call to fhapp that we generalized must have type $(ls1 ++ ls2) ++ ls3$, for some values of the list variables. If we rewrite the type of the proof used to type-cast this value to something like $ls1 ++ ls2 ++ ls3 = ls1 ++ ls2 ++ ls3$, then the lefthand side of the equality would no longer match the type of the term we are trying to cast.

However, now that we have generalized over the fhapp call, the type of the term being type-cast appears explicitly in the goal and *may be rewritten as well*. In particular, the final masterstroke is rewriting everywhere in our goal using associativity of list append.

```
  rewrite app_ass.

============================
 ∀ (pf0 : a :: ls1 ++ ls2 ++ ls3 = a :: ls1 ++ ls2 ++ ls3)
   (pf'0 : ls1 ++ ls2 ++ ls3 = ls1 ++ ls2 ++ ls3)
   (f : fhlist B (ls1 ++ ls2 ++ ls3)),
 (a0,
 match pf'0 in (_ = ls) return (fhlist B ls) with
 | refl_equal ⇒ f
 end) =
 match pf0 in (_ = ls) return (fhlist B ls) with
 | refl_equal ⇒ (a0, f)
 end
```

We can see that we have achieved the crucial property: the type of each generalized equality proof has syntactically equal operands. This makes it easy to finish the proof with UIP_refl.

```
  intros.
  rewrite (UIP_refl _ _ pf0).
  rewrite (UIP_refl _ _ pf'0).
  reflexivity.
 Qed.
End fhapp.

Implicit Arguments fhapp [A B ls1 ls2].
```

6.4 Heterogeneous Equality

There is another equality predicate, defined in the **JMeq** module of the standard library, implementing *heterogeneous equality*.

```
Print JMeq.
Inductive JMeq (A : Type) (x : A) : ∀ B : Type, B → Prop :=
    JMeq_refl : JMeq x x
```

JMeq stands for "John Major equality," a name coined by Conor McBride as a sort of pun about British politics. **JMeq** starts out looking a lot like *eq*. The crucial difference is that we may use **JMeq** *on arguments of different types*. For instance, a lemma that we failed to establish before is trivial with **JMeq**. It makes for prettier theorem statements to define some syntactic shorthand first.

```
Infix "==" := JMeq (at level 70, no associativity).

Definition UIP_refl' (A : Type) (x : A) (pf : x = x) : pf == refl_equal x :=
  match pf return (pf == refl_equal _) with
    | refl_equal ⇒ JMeq_refl _
  end.
```

There is no quick way to write such a proof by tactics, but the underlying proof term that we want is trivial.

Suppose that we want to use UIP_refl' to establish another lemma of the kind we have run into several times so far.

```
Lemma lemma4 : ∀ (A : Type) (x : A) (pf : x = x),
  O = match pf with refl_equal ⇒ O end.
  intros; rewrite (UIP_refl' pf); reflexivity.
Qed.
```

All in all, refreshingly straightforward, but there really is no such thing as a free lunch. The use of `rewrite` is implemented in terms of an axiom:

```
Check JMeq_eq.
JMeq_eq
     : ∀ (A : Type) (x y : A), x == y → x = y
```

It may be surprising that we cannot prove that heterogeneous equality implies normal equality. The difficulties are the same kind we have seen so far, based on limitations of `match` annotations.

We can redo our fhapp associativity proof based around **JMeq**.

```
Section fhapp'.
  Variable A : Type.
  Variable B : A → Type.
```

This time, the naive theorem statement type-checks, which demonstrates the main advantage of **JMeq** over *eq*.

```
  Theorem fhapp_ass' : ∀ ls1 ls2 ls3
    (hls1 : fhlist B ls1) (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3) == fhapp (fhapp hls1 hls2) hls3.
```

```
induction ls1; crush.
```

Even better, *crush* discharges the first subgoal automatically. The second subgoal is:

```
============================
 (a0,
 fhapp (B:=B) (ls1:=ls1) (ls2:=ls2 ++ ls3) b
   (fhapp (B:=B) (ls1:=ls2) (ls2:=ls3) hls2 hls3)) =
 (a0,
 fhapp (B:=B) (ls1:=ls1 ++ ls2) (ls2:=ls3)
   (fhapp (B:=B) (ls1:=ls1) (ls2:=ls2) b hls2) hls3)
```

It looks like one rewrite with the inductive hypothesis should be enough to make the goal trivial.

```
rewrite IHls1.
```

```
Error: Impossible to unify "fhlist B ((ls1 ++ ?1572) ++ ?1573)" with
 "fhlist B (ls1 ++ ?1572 ++ ?1573)"
```

We see that **JMeq** is not a silver bullet. We can use it to simplify the statements of equality facts, but the Coq type-checker uses non-trivial heterogeneous equality facts no more readily than it uses standard equality facts. Here, the problem is that the form (*e1*, *e2*) is syntactic sugar for an explicit application of a constructor of an inductive type. That application mentions the type of each tuple element explicitly, and our `rewrite` tries to change one of those elements without updating the corresponding type argument.

We can get around this problem by another multiple use of `generalize`. We want to bring into the goal the proper instance of the inductive hypothesis, and we also want to generalize the two relevant uses of fhapp.

```
generalize (fhapp b (fhapp hls2 hls3))
  (fhapp (fhapp b hls2) hls3)
  (IHls1 _ _ b hls2 hls3).
```

```
============================
 ∀ (f : fhlist B (ls1 ++ ls2 ++ ls3))
   (f0 : fhlist B ((ls1 ++ ls2) ++ ls3)), f == f0 → (a0, f) == (a0, f0)
```

Now we can rewrite with append associativity, as before.

```
rewrite app_ass.
```

```
============================
 ∀ f f0 : fhlist B (ls1 ++ ls2 ++ ls3), f == f0 → (a0, f) == (a0, f0)
```

From this point, the goal is trivial.

```
intros f f0 H; rewrite H; reflexivity.
Qed.
```

```
End fhapp'.
```

6.5 Equivalence of Equality Axioms

Assuming axioms (like axiom K and *JMeq_eq*) is a hazardous business. The due diligence associated with it is necessarily global in scope, since two axioms may be consistent alone but inconsistent together. It turns out that all of the major axioms proposed for reasoning about equality in Coq are logically equivalent, so that we only need to pick one to assert without proof. In this subsection, we demonstrate this by showing how each of the previous two subsections' approaches reduces to the other logically.

To show that **JMeq** and its axiom let us prove UIP_refl, we start from the lemma UIP_refl' from the previous subsection. The rest of the proof is trivial.

```
Lemma UIP_refl'' : ∀ (A : Type) (x : A) (pf : x = x), pf = refl_equal x.
  intros; rewrite (UIP_refl' pf); reflexivity.
Qed.
```

The other direction is perhaps more interesting. Assume that we only have the axiom of the *Eqdep* module available. We can define **JMeq** in a way that satisfies the same interface as the combination of the **JMeq** module's inductive definition and axiom.

```
Definition JMeq' (A : Type) (x : A) (B : Type) (y : B) : Prop :=
  ∃ pf : B = A, x = match pf with refl_equal ⇒ y end.

Infix "===" := JMeq' (at level 70, no associativity).
```

We say that, by definition, x and y are equal if and only if there exists a proof pf that their types are equal, such that x equals the result of casting y with pf. This statement can look strange from the standpoint of classical math, where we almost never mention proofs explicitly with quantifiers in formulas, but it is perfectly legal Coq code.

We can easily prove a theorem with the same type as that of the JMeq_refl constructor of **JMeq**.

```
Theorem JMeq_refl' : ∀ (A : Type) (x : A), x === x.
  intros; unfold JMeq'; exists (refl_equal A); reflexivity.
Qed.
```

The proof of an analogue to *JMeq_eq* is a little more interesting, but most of the action is in appealing to UIP_refl.

```
Theorem JMeq_eq' : ∀ (A : Type) (x y : A),
  x === y → x = y.
  unfold JMeq'; intros.

  H : ∃ pf : A = A,
        x = match pf in (_ = T) return T with
            | refl_equal ⇒ y
            end
  ============================
```

```
 x = y

destruct H.

x0 : A = A
H : x = match x0 in (_ = T) return T with
        | refl_equal ⇒ y
        end
============================
 x = y

rewrite H.

x0 : A = A
============================
 match x0 in (_ = T) return T with
 | refl_equal ⇒ y
 end = y

rewrite (UIP_refl _ _ x0); reflexivity.
Qed.
```

We see that, in a very formal sense, we are free to switch back and forth between the two styles of proofs about equality proofs. One style may be more convenient than the other for some proofs, but we can always interconvert between our results. The style that does not use heterogeneous equality may be preferable in cases where many results do not require the tricks of this section, since then the use of axioms is avoided altogether for the simple cases, and a wider audience will be able to follow those "simple" proofs. On the other hand, heterogeneous equality often makes for shorter and more readable theorem statements.

It is worth remarking that it is possible to avoid axioms altogether for equalities on types with decidable equality. The *Eqdep_dec* module of the standard library contains a parametric proof of UIP_refl for such cases.

6.6 Equality of Functions

The following seems like a reasonable theorem to want to hold, and it does hold in set theory.

```
Theorem S_eta : S = (fun n ⇒ S n).
```

Unfortunately, this theorem is not provable in CIC without additional axioms. None of the definitional equality rules force function equality to be *extensional*. That is, the fact that two functions return equal results on equal inputs does not imply that the functions are equal. We *can* assert function extensionality as an axiom.

```
Axiom ext_eq : ∀ A B (f g : A → B),
  (∀ x, f x = g x)
```

```
  → f = g.
```

This axiom has been verified metatheoretically to be consistent with CIC and the two equality axioms we considered previously. With it, the proof of S_eta is trivial.

```
Theorem S_eta : S = (fun n ⇒ S n).
  apply ext_eq; reflexivity.
Qed.
```

The same axiom can help us prove equality of types, where we need to "reason under quantifiers."

```
Theorem forall_eq : (∀ x : nat, match x with
                                  | O ⇒ True
                                  | S _ ⇒ True
                                end)
  = (∀ _ : nat, True).
```

There are no immediate opportunities to apply *ext_eq*, but we can use *change* to fix that, by replacing the conclusion with another that is definitionally equal.

```
  change ((∀ x : nat, (fun x ⇒ match x with
                                 | 0 ⇒ True
                                 | S _ ⇒ True
                               end) x) = (nat → True)).
  rewrite (ext_eq (fun x ⇒ match x with
                             | 0 ⇒ True
                             | S _ ⇒ True
                           end) (fun _ ⇒ True)).
```

2 *subgoals*

```
  ============================
   (nat → True) = (nat → True)

subgoal 2 is:
 ∀ x : nat, match x with
              | 0 ⇒ True
              | S _ ⇒ True
              end = True
```

```
  reflexivity.
  destruct x; constructor.
Qed.
```

7. CONCLUSION

This article has been a brisk introduction to two underappreciated elements of the Coq proof assistant.

With dependent types, we can often prove important theorems implicitly, with little additional effort beyond that needed to implement a running program. Coq's

minimalist type theory lacks some conveniences that similar dependently-typed languages offer. However, a few design patterns make it relatively pleasant to use dependent types to simplify Coq developments. The bulk of this article has been a presentation of the main design patterns by example.

I have also hinted at the power of Ltac, Coq's domain-specific language for building proof procedures. An Ltac program that succeeds at proving a theorem will always generate a proof term in Coq's minimal logic, by construction. Ltac contains both the usual abstraction tools of functional programming and a few new ones targeted especially at proof search. For a more thorough, bottom-up presentation of Ltac, I point the reader to the full book that this article is excerpted from:

http://adam.chlipala.net/cpdt/