Certified Programming with Dependent Types

Adam Chlipala

December 30, 2009

Contents

Chapter 1

Introduction

1.1 Whence This Book?

We would all like to have programs check that our programs are correct. Due in no small part to some bold but unfulfilled promises in the history of computer science, today most people who write software, practitioners and academics alike, assume that the costs of formal program verification outweigh the benefits. The purpose of this book is to convince you that the technology of program verification is mature enough today that it makes sense to use it in a support role in many kinds of research projects in computer science. Beyond the convincing, I also want to provide a handbook on practical engineering of certified programs with the Coq proof assistant.

There are a good number of (though definitely not "many") tools that are in wide use today for building machine-checked mathematical proofs and machine-certified programs. This is my attempt at an exhaustive list of interactive "proof assistants" satisfying a few criteria. First, the authors of each tool must intend for it to be put to use for software-related applications. Second, there must have been enough engineering effort put into the tool that someone not doing research on the tool itself would feel his time was well spent using it. A third criterion is more of an empirical validation of the second: the tool must have a significant user community outside of its own development team.

ACL2	http://www.cs.utexas.edu/users/moore/acl2/
Coq	http://coq.inria.fr/
Isabelle/HOL	http://isabelle.in.tum.de/
PVS	http://pvs.csl.sri.com/
Twelf	http://www.twelf.org/

Isabelle/HOL, implemented with the "proof assistant development framework" Isabelle, is the most popular proof assistant for the HOL logic. The other implementations of HOL can be considered equivalent for purposes of the discussion here.

1.2 Why Coq?

This book is going to be about certified programming using Coq, and I am convinced that it is the best tool for the job. Coq has a number of very attractive properties, which I will summarize here, mentioning which of the other candidate tools lack each property.

1.2.1 Based on a Higher-Order Functional Programming Language

There is no reason to give up the familiar comforts of functional programming when you start writing certified programs. All of the tools I listed are based on functional programming languages, which means you can use them without their proof-related aspects to write and run regular programs.

ACL2 is notable in this field for having only a *first-order* language at its foundation. That is, you cannot work with functions over functions and all those other treats of functional programming. By giving up this facility, ACL2 can make broader assumptions about how well its proof automation will work, but we can generally recover the same advantages in other proof assistants when we happen to be programming in first-order fragments.

1.2.2 Dependent Types

A language of *dependent types* may include references to programs inside of types. For instance, the type of an array might include a program expression giving the size of the array, making it possible to verify absence of out-of-bounds accesses statically. Dependent types can go even further than this, effectively capturing any correctness property in a type. For instance, later in this book, we will see how to give a Mini-ML compiler a type that guarantees that it maps well-typed source programs to well-typed target programs.

ACL2 and HOL lack dependent types outright. PVS and Twelf each supports a different strict subset of Coq's dependent type language. Twelf's type language is restricted to a bare-bones, monomorphic lambda calculus, which places serious restrictions on how complicated *computations inside types* can be. This restriction is important for the soundness argument behind Twelf's approach to representing and checking proofs.

In contrast, PVS's dependent types are much more general, but they are squeezed inside the single mechanism of *subset types*, where a normal type is refined by attaching a predicate over its elements. Each member of the subset type is an element of the base type that satisfies the predicate.

Dependent types are not just useful because they help you express correctness properties in types. Dependent types also often let you write certified programs *without writing anything that looks like a proof*. Even with subset types, which for many contexts can be used to express any relevant property with enough acrobatics, the human driving the proof assistant usually has to build some proofs explicitly. Writing formal proofs is hard, so we want to avoid it as far as possible, so dependent types are invaluable.

1.2.3 An Easy-to-Check Kernel Proof Language

Scores of automated decision procedures are useful in practical theorem proving, but it is unfortunate to have to trust in the correct implementation of each procedure. Proof assistants satisfying the "de Bruijn criterion" may use complicated and extensible procedures to seek out proofs, but in the end they produce *proof terms* in kernel languages. These core languages have feature complexity on par with what you find in proposals for formal foundations for mathematics. To believe a proof, we can ignore the possibility of bugs during *search* and just rely on a (relatively small) proof-checking kernel that we apply to the *result* of the search.

ACL2 and PVS do not meet the de Bruijn criterion, employing fancy decision procedures that produce no "evidence trails" justifying their results.

1.2.4 Convenient Programmable Proof Automation

A commitment to a kernel proof language opens up wide possibilities for user extension of proof automation systems, without allowing user mistakes to trick the overall system into accepting invalid proofs. Almost any interesting verification problem is undecidable, so it is important to help users build their own procedures for solving the restricted problems that they encounter in particular implementations.

Twelf features no proof automation marked as a bonafide part of the latest release; there is some automation code included for testing purposes. The Twelf style is based on writing out all proofs in full detail. Because Twelf is specialized to the domain of syntactic metatheory proofs about programming languages and logics, it is feasible to use it to write those kinds of proofs manually. Outside that domain, the lack of automation can be a serious obstacle to productivity. Most kinds of program verification fall outside Twelf's forte.

Of the remaining tools, all can support user extension with new decision procedures by hacking directly in the tool's implementation language (such as OCaml for Coq). Since ACL2 and PVS do not satisfy the de Bruijn criterion, overall correctness is at the mercy of the authors of new procedures.

Isabelle/HOL and Coq both support coding new proof manipulations in ML in ways that cannot lead to the acceptance of invalid proofs. Additionally, Coq includes a domain-specific language for coding decision procedures in normal Coq source code, with no need to break out into ML. This language is called Ltac, and I think of it as the unsung hero of the proof assistant world. Not only does Ltac prevent you from making fatal mistakes, it also includes a number of novel programming constructs which combine to make a "proof by decision procedure" style very pleasant. We will meet these features in the chapters to come.

1.2.5 Proof by Reflection

A surprising wealth of benefits follow from choosing a proof language that integrates a rich notion of computation. Coq includes programs and proof terms in the same syntactic class. This makes it easy to write programs that compute proofs. With rich enough dependent

types, such programs are *certified decision procedures*. In such cases, these certified procedures can be put to good use *without ever running them*! Their types guarantee that, if we did bother to run them, we would receive proper "ground" proofs.

The critical ingredient for this technique, many of whose instances are referred to as *proof by reflection*, is a way of inducing non-trivial computation inside of logical propositions during proof checking. Further, most of these instances require dependent types to make it possible to state the appropriate theorems. Of the proof assistants I listed, only Coq really provides this support.

1.3 Why Not a Different Dependently-Typed Language?

The logic and programming language behind Coq belongs to a type-theory ecosystem with a good number of other thriving members. Agda[1] and Epigram[2] are the most developed tools among the alternatives to Coq, and there are others that are earlier in their lifecycles. All of the languages in this family feel sort of like different historical offshoots of Latin. The hardest conceptual epiphanies are, for the most part, portable among all the languages. Given this, why choose Coq for certified programming?

I think the answer is simple. None of the competition has well-developed systems for tactic-based theorem proving. Agda and Epigram are designed and marketed more as programming languages than proof assistants. Dependent types are great, because they often help you prove deep theorems without doing anything that feels like proving. Nonetheless, almost any interesting certified programming project will benefit from some activity that deserves to be called proving, and many interesting projects absolutely require semi-automated proving, if the sanity of the programmer is to be safeguarded. Informally, proving is unavoidable when any correctness proof for a program has a structure that does not mirror the structure of the program itself. An example is a compiler correctness proof, which probably proceeds by induction on program execution traces, which have no simple relationship with the structure of the compiler or the structure of the programs it compiles. In building such proofs, a mature system for scripted proof automation is invaluable.

On the other hand, Agda, Epigram, and similar tools have less implementation baggage associated with them, and so they tend to be the default first homes of innovations in practical type theory. Some significant kinds of dependently-typed programs are much easier to write in Agda and Epigram than in Coq. The former tools may very well be superior choices for projects that do not involve any "proving." Anecdotally, I have gotten the impression that manual proving is orders of magnitudes more costly than manual coping with Coq's lack of programming bells and whistles. In this book, I will devote significant time to patterns for programming with dependent types in Coq as it is today. We can hope that the type theory community is tending towards convergence on the right set of features for practical programming with dependent types, and that we will eventually have a single tool embodying

[1] http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php

[2] http://www.e-pig.org/

those features.

1.4 Engineering with a Proof Assistant

In comparisons with its competitors, Coq is often derided for promoting unreadable proofs. It is very easy to write proof scripts that manipulate proof goals imperatively, with no structure to aid readers. Such developments are nightmares to maintain, and they certainly do not manage to convey "why the theorem is true" to anyone but the original author. One additional (and not insignificant) purpose of this book is to show why it is unfair and unproductive to dismiss Coq based on the existence of such developments.

I will go out on a limb and guess that the reader is a dedicated fan of some functional programming language or another, and that he may even have been involved in teaching that language to undergraduates. I want to propose an analogy between two attitudes: coming to a negative conclusion about Coq after reading common Coq developments in the wild, and coming to a negative conclusion about Your Favorite Language after looking at the programs undergraduates write in it in the first week of class. The pragmatics of mechanized proving and program verification have been under serious study for much less time than the pragmatics of programming have been. The computer theorem proving community is still developing the key insights that correspond to those that functional programming texts and instructors impart to their students, to help those students get over that critical hump where using the language stops being more trouble than it is worth. Most of the insights for Coq are barely even disseminated among the experts, let alone set down in a tutorial form. I hope to use this book to go a long way towards remedying that.

If I do that job well, then this book should be of interest even to people who have participated in classes or tutorials specifically about Coq. The book should even be useful to people who have been using Coq for years but who are mystified when their Coq developments prove impenetrable by colleagues. The crucial angle in this book is that there are "design patterns" for reliably avoiding the really grungy parts of theorem proving, and consistent use of these patterns can get you over the hump to the point where it is worth your while to use Coq to prove your theorems and certify your programs, even if formal verification is not your main concern in a project. We will follow this theme by pursuing two main methods for replacing manual proofs with more understandable artifacts: dependently-typed functions and custom Ltac decision procedures.

1.5 Prerequisites

I try to keep the required background knowledge to a minimum in this book. I will assume familiarity with the material from usual discrete math and logic courses taken by all undergraduate computer science majors, and I will assume that readers have significant experience programming in one of the ML dialects, in Haskell, or in some other, closely-related language.

Experience with only dynamically-typed functional languages might lead to befuddlement in some places, but a reader who has come to grok Scheme deeply will probably be fine.

A good portion of the book is about how to formalize programming languages, compilers, and proofs about them. To understand those parts, it will be helpful to have a basic knowledge of formal type systems, operational semantics, and the theorems usually proved about such systems. As a reference on these topics, I recommend *Types and Programming Languages*, by Benjamin C. Pierce.

1.6 Using This Book

This book is generated automatically from Coq source files using the wonderful coqdoc program. The latest PDF version is available at:

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

There is also an online HTML version available, with a hyperlink from each use of an identifier to that identifier's definition:

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

The source code to the book is also freely available at:

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

There, you can find all of the code appearing in this book, with prose interspersed in comments, in exactly the order that you find in this document. You can step through the code interactively with your chosen graphical Coq interface. The code also has special comments indicating which parts of the chapters make suitable starting points for interactive class sessions, where the class works together to construct the programs and proofs. The included Makefile has a target `templates` for building a fresh set of class template files automatically from the book source.

I believe that a good graphical interface to Coq is crucial for using it productively. I use the Proof General[3] mode for Emacs, which supports a number of other proof assistants besides Coq. There is also the standalone CoqIDE program developed by the Coq team. I like being able to combine certified programming and proving with other kinds of work inside the same full-featured editor, and CoqIDE has had a good number of crashes and other annoying bugs in recent history, though I hear that it is improving. In the initial part of this book, I will reference Proof General procedures explicitly, in introducing how to use Coq, but most of the book will be interface-agnostic, so feel free to use CoqIDE if you prefer it.

[3] http://proofgeneral.inf.ed.ac.uk/

1.7 Chapter Source Files

Chapter	Source
Some Quick Examples	`StackMachine.v`
Introducing Inductive Types	`InductiveTypes.v`
Inductive Predicates	`Predicates.v`
Infinite Data and Proofs	`Coinductive.v`
Subset Types and Variations	`Subset.v`
More Dependent Types	`MoreDep.v`
Dependent Data Structures	`DataStruct.v`
Reasoning About Equality Proofs	`Equality.v`
Generic Programming	`Generic.v`
Universes and Axioms	`Universes.v`
Proof Search in Ltac	`Match.v`
Proof by Reflection	`Reflection.v`
Proving in the Large	`Large.v`
First-Order Abstract Syntax	`Firstorder.v`
Dependent De Bruijn Indices	`DeBruijn.v`
Higher-Order Abstract Syntax	`Hoas.v`
Type-Theoretic Interpreters	`Interps.v`
Extensional Transformations	`Extensional.v`
Intensional Transformations	`Intensional.v`
Higher-Order Operational Semantics	`OpSem.v`

Chapter 2

Some Quick Examples

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. I assume that you have installed Coq and Proof General. The code in this book is tested with Coq version 8.2pl1, though parts may work with other versions.

As always, you can step through the source file `StackMachine.v` for this chapter 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 chapter in a new `.v` file in an Emacs buffer. If you do the latter, include a line `Require Import` *List* `Tactics.` at the start of the file, to match some code hidden in this rendering of the chapter 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 source for this book. In either case, you will need to run `make` in the root directory of the source distribution for the book 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" "/path/to/cpdt/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 chapter. 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 supports pretty-printing of tokens in LaTeX or HTML. Where you see a right arrow character, the source contains the ASCII text `->`. Other examples of this substitution appearing in this chapter are a double right arrow for `=>` and the inverted 'A' symbol for `forall`. When in doubt about the ASCII version of a symbol, you can consult the chapter 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. (If you

are not familiar with these semantic techniques, no need to worry; we will stick to "common sense" constructions.)

```
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 chapter, and much of this book is devoted to more involved Ltac examples.

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 book and is not part of the Coq standard library. The book's library contains a number of other tactics that are especially helpful in highly-automated proofs.

The 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.

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.
```

Part I

Basic Programming and Proving

Chapter 3

Introducing Inductive Types

In a sense, CIC is built from just two relatively straightforward features: function types and inductive types. From this modest foundation, we can prove effectively all of the theorems of math and carry out effectively all program verifications, with enough effort expended. This chapter introduces induction and recursion for functional programming in Coq.

3.1 Enumerations

Coq inductive types generalize the algebraic datatypes found in Haskell and ML. Confusingly enough, inductive types also generalize generalized algebraic datatypes (GADTs), by adding the possibility for type dependency. Even so, it is worth backing up from the examples of the last chapter and going over basic, algebraic datatype uses of inductive datatypes, because the chance to prove things about the values of these types adds new wrinkles beyond usual practice in Haskell and ML.

The singleton type **unit** is an inductive type:

```
Inductive unit : Set :=
  | tt.
```

This vernacular command defines a new inductive type **unit** whose only value is tt, as we can see by checking the types of the two identifiers:

```
Check unit.
  unit : Set

Check tt.
  tt : unit
```

We can prove that **unit** is a genuine singleton type.

```
Theorem unit_singleton : ∀ x : unit, x = tt.
```

The important thing about an inductive type is, unsurprisingly, that you can do induction over its values, and induction is the key to proving this theorem. We ask to proceed by induction on the variable x.

```
  induction x.
```

The goal changes to:

```
tt = tt
```

...which we can discharge trivially.

```
  reflexivity.
Qed.
```

It seems kind of odd to write a proof by induction with no inductive hypotheses. We could have arrived at the same result by beginning the proof with:

```
  destruct x.
```

...which corresponds to "proof by case analysis" in classical math. For non-recursive inductive types, the two tactics will always have identical behavior. Often case analysis is sufficient, even in proofs about recursive types, and it is nice to avoid introducing unneeded induction hypotheses.

What exactly *is* the induction principle for **unit**? We can ask Coq:

```
Check unit_ind.
```

unit_ind : $\forall$ P : **unit** $\rightarrow$ Prop, P tt $\rightarrow$ $\forall$ u : **unit**, P u

Every `Inductive` command defining a type T also defines an induction principle named *T_ind*. Coq follows the Curry-Howard correspondence and includes the ingredients of programming and proving in the same single syntactic class. Thus, our type, operations over it, and principles for reasoning about it all live in the same language and are described by the same type system. The key to telling what is a program and what is a proof lies in the distinction between the type `Prop`, which appears in our induction principle; and the type `Set`, which we have seen a few times already.

The convention goes like this: `Set` is the type of normal types, and the values of such types are programs. `Prop` is the type of logical propositions, and the values of such types are proofs. Thus, an induction principle has a type that shows us that it is a function for building proofs.

Specifically, unit_ind quantifies over a predicate P over **unit** values. If we can present a proof that P holds of tt, then we are rewarded with a proof that P holds for any value u of type **unit**. In our last proof, the predicate was (`fun` u : **unit** $\Rightarrow$ u = tt).

We can define an inductive type even simpler than **unit**:

```
Inductive Empty_set : Set := .
```

Empty_set has no elements. We can prove fun theorems about it:

```
Theorem the_sky_is_falling : ∀ x : Empty_set, 2 + 2 = 5.
  destruct 1.
Qed.
```

Because **Empty_set** has no elements, the fact of having an element of this type implies anything. We use `destruct` 1 instead of `destruct` x in the proof because unused quantified

variables are relegated to being referred to by number. (There is a good reason for this, related to the unity of quantifiers and implication. An implication is just a quantification over a proof, where the quantified variable is never used. It generally makes more sense to refer to implication hypotheses by number than by name, and Coq treats our quantifier over an unused variable as an implication in determining the proper behavior.)

We can see the induction principle that made this proof so easy:

```
Check Empty_set_ind.
```

```
Empty_set_ind : ∀ (P : Empty_set → Prop) (e : Empty_set), P e
```

In other words, any predicate over values from the empty set holds vacuously of every such element. In the last proof, we chose the predicate (`fun _ : Empty_set ⇒ 2 + 2 = 5`).

We can also apply this get-out-of-jail-free card programmatically. Here is a lazy way of converting values of **Empty_set** to values of **unit**:

```
Definition e2u (e : Empty_set) : unit := match e with end.
```

We employ `match` pattern matching as in the last chapter. Since we match on a value whose type has no constructors, there is no need to provide any branches.

Moving up the ladder of complexity, we can define the booleans:

```
Inductive bool : Set :=
| true
| false.
```

We can use less vacuous pattern matching to define boolean negation.

```
Definition not (b : bool) : bool :=
  match b with
    | true ⇒ false
    | false ⇒ true
  end.
```

An alternative definition desugars to the above:

```
Definition not' (b : bool) : bool :=
  if b then false else true.
```

We might want to prove that not is its own inverse operation.

```
Theorem not_inverse : ∀ b : bool, not (not b) = b.
  destruct b.
```

After we case-analyze on b, we are left with one subgoal for each constructor of **bool**.

```
2 subgoals

  ============================
   not (not true) = true

subgoal 2 is:
```

```
  not (not false) = false
```

The first subgoal follows by Coq's rules of computation, so we can dispatch it easily:

```
  reflexivity.
```

Likewise for the second subgoal, so we can restart the proof and give a very compact justification.

```
Restart.
  destruct b; reflexivity.
Qed.
```

Another theorem about booleans illustrates another useful tactic.

```
Theorem not_ineq : ∀ b : bool, not b ≠ b.
  destruct b; discriminate.
Qed.
```

`discriminate` is used to prove that two values of an inductive type are not equal, whenever the values are formed with different constructors. In this case, the different constructors are true and false.

At this point, it is probably not hard to guess what the underlying induction principle for **bool** is.

```
Check bool_ind.
  bool_ind : ∀ P : bool → Prop, P true → P false → ∀ b : bool, P b
```

3.2 Simple Recursive Types

The natural numbers are the simplest common example of an inductive type that actually deserves the name.

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

O is zero, and S is the successor function, so that 0 is syntactic sugar for O, 1 for S O, 2 for S (S O), and so on.

Pattern matching works as we demonstrated in the last chapter:

```
Definition isZero (n : nat) : bool :=
  match n with
    | O ⇒ true
    | S _ ⇒ false
  end.

Definition pred (n : nat) : nat :=
  match n with
```

```
    | O ⇒ O
    | S n' ⇒ n'
  end.
```

We can prove theorems by case analysis:

```
Theorem S_isZero : ∀ n : nat, isZero (pred (S (S n))) = false.
  destruct n; reflexivity.
Qed.
```

We can also now get into genuine inductive theorems. First, we will need a recursive function, to make things interesting.

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

Recall that `Fixpoint` is Coq's mechanism for recursive function definitions. Some theorems about plus can be proved without induction.

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.
```

Coq's computation rules automatically simplify the application of plus, because unfolding the definition of plus gives us a `match` expression where the branch to be taken is obvious from syntax alone. If we just reverse the order of the arguments, though, this no longer works, and we need induction.

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

Our first subgoal is plus O O = O, which *is* trivial by computation.

```
  reflexivity.
```

Our second subgoal is more work and also demonstrates our first inductive hypothesis.

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

We can start out by using computation to simplify the goal as far as we can.

```
  simpl.
```

Now the conclusion is S (plus n O) = S n. Using our inductive hypothesis:

```
  rewrite IHn.
```

...we get a trivial conclusion $S\ n = S\ n$.

```
  reflexivity.
```

Not much really went on in this proof, so the *crush* tactic from the Tactics module can prove this theorem automatically.

```
Restart.
  induction n; crush.
Qed.
```

We can check out the induction principle at work here:

```
Check nat_ind.

  nat_ind : ∀ P : nat → Prop,
              P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Each of the two cases of our last proof came from the type of one of the arguments to nat_ind. We chose P to be (fun n : nat ⇒ plus n O = n). The first proof case corresponded to P O and the second case to ($\forall\ n$: nat, $P\ n \rightarrow P\ (S\ n)$). The free variable n and inductive hypothesis *IHn* came from the argument types given here.

Since nat has a constructor that takes an argument, we may sometimes need to know that that constructor is injective.

```
Theorem S_inj : ∀ n m : nat, S n = S m → n = m.
  injection 1; trivial.
Qed.
```

injection refers to a premise by number, adding new equalities between the corresponding arguments of equated terms that are formed with the same constructor. We end up needing to prove $n = m \rightarrow n = m$, so it is unsurprising that a tactic named trivial is able to finish the proof.

There is also a very useful tactic called congruence that can prove this theorem immediately. congruence generalizes discriminate and injection, and it also adds reasoning about the general properties of equality, such as that a function returns equal results on equal arguments. That is, congruence is a *complete decision procedure for the theory of equality and uninterpreted functions*, plus some smarts about inductive types.

We can define a type of lists of natural numbers.

```
Inductive nat_list : Set :=
| NNil : nat_list
| NCons : nat → nat_list → nat_list.
```

Recursive definitions are straightforward extensions of what we have seen before.

```
Fixpoint nlength (ls : nat_list) : nat :=
  match ls with
    | NNil ⇒ O
    | NCons _ ls' ⇒ S (nlength ls')
```

```
  end.

Fixpoint napp (ls1 ls2 : nat_list) : nat_list :=
  match ls1 with
    | NNil ⇒ ls2
    | NCons n ls1' ⇒ NCons n (napp ls1' ls2)
  end.
```

Inductive theorem proving can again be automated quite effectively.

```
Theorem nlength_napp : ∀ ls1 ls2 : nat_list, nlength (napp ls1 ls2)
  = plus (nlength ls1) (nlength ls2).
  induction ls1; crush.
Qed.

Check nat_list_ind.

  nat_list_ind
     : ∀ P : nat_list → Prop,
       P NNil →
       (∀ (n : nat) (n0 : nat_list), P n0 → P (NCons n n0)) →
       ∀ n : nat_list, P n
```

In general, we can implement any "tree" types as inductive types. For example, here are binary trees of naturals.

```
Inductive nat_btree : Set :=
| NLeaf : nat_btree
| NNode : nat_btree → nat → nat_btree → nat_btree.

Fixpoint nsize (tr : nat_btree) : nat :=
  match tr with
    | NLeaf ⇒ S O
    | NNode tr1 _ tr2 ⇒ plus (nsize tr1) (nsize tr2)
  end.

Fixpoint nsplice (tr1 tr2 : nat_btree) : nat_btree :=
  match tr1 with
    | NLeaf ⇒ NNode tr2 O NLeaf
    | NNode tr1' n tr2' ⇒ NNode (nsplice tr1' tr2) n tr2'
  end.

Theorem plus_assoc : ∀ n1 n2 n3 : nat, plus (plus n1 n2) n3 = plus n1 (plus n2 n3).
  induction n1; crush.
Qed.

Theorem nsize_nsplice : ∀ tr1 tr2 : nat_btree, nsize (nsplice tr1 tr2)
  = plus (nsize tr2) (nsize tr1).
  Hint Rewrite n_plus_O plus_assoc : cpdt.

  induction tr1; crush.
```

```
Qed.

Check nat_btree_ind.

  nat_btree_ind
      : ∀ P : nat_btree → Prop,
        P NLeaf →
        (∀ n : nat_btree,
         P n → ∀ (n0 : nat) (n1 : nat_btree), P n1 → P (NNode n n0 n1)) →
        ∀ n : nat_btree, P n
```

3.3 Parameterized Types

We can also define polymorphic inductive types, as with algebraic datatypes in Haskell and ML.

```
Inductive list (T : Set) : Set :=
| Nil : list T
| Cons : T → list T → list T.

Fixpoint length T (ls : list T) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

Fixpoint app T (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons x ls1' ⇒ Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1; crush.
Qed.
```

There is a useful shorthand for writing many definitions that share the same parameter, based on Coq's *section* mechanism. The following block of code is equivalent to the above:

```
Section list.
  Variable T : Set.

  Inductive list : Set :=
  | Nil : list
  | Cons : T → list → list.

  Fixpoint length (ls : list) : nat :=
```

```
    match ls with
      | Nil ⇒ O
      | Cons _ ls' ⇒ S (length ls')
    end.

  Fixpoint app (ls1 ls2 : list) : list :=
    match ls1 with
      | Nil ⇒ ls2
      | Cons x ls1' ⇒ Cons x (app ls1' ls2)
    end.

  Theorem length_app : ∀ ls1 ls2 : list, length (app ls1 ls2)
    = plus (length ls1) (length ls2).
    induction ls1; crush.
  Qed.
End list.
```

After we end the section, the `Variable`s we used are added as extra function parameters for each defined identifier, as needed. We verify that this has happened using the `Print` command, a cousin of `Check` which shows the definition of a symbol, rather than just its type.

```
Print list.
  Inductive list (T : Set) : Set :=
    Nil : list T | Cons : T → list T → list Tlist
```

The final definition is the same as what we wrote manually before. The other elements of the section are altered similarly, turning out exactly as they were before, though we managed to write their definitions more succinctly.

```
Check length.
  length
     : ∀ T : Set, list T → nat
```

The parameter T is treated as a new argument to the induction principle, too.

```
Check list_ind.
  list_ind
     : ∀ (T : Set) (P : list T → Prop),
       P (Nil T) →
       (∀ (t : T) (l : list T), P l → P (Cons t l)) →
       ∀ l : list T, P l
```

Thus, even though we just saw that T is added as an extra argument to the constructor Cons, there is no quantifier for T in the type of the inductive case like there is for each of the other arguments.

3.4 Mutually Inductive Types

We can define inductive types that refer to each other:

```
Inductive even_list : Set :=
| ENil : even_list
| ECons : nat → odd_list → even_list

with odd_list : Set :=
| OCons : nat → even_list → odd_list.
Fixpoint elength (el : even_list) : nat :=
  match el with
    | ENil ⇒ O
    | ECons _ ol ⇒ S (olength ol)
  end

with olength (ol : odd_list) : nat :=
  match ol with
    | OCons _ el ⇒ S (elength el)
  end.
Fixpoint eapp (el1 el2 : even_list) : even_list :=
  match el1 with
    | ENil ⇒ el2
    | ECons n ol ⇒ ECons n (oapp ol el2)
  end

with oapp (ol : odd_list) (el : even_list) : odd_list :=
  match ol with
    | OCons n el' ⇒ OCons n (eapp el' el)
  end.
```

Everything is going roughly the same as in past examples, until we try to prove a theorem similar to those that came before.

```
Theorem elength_eapp : ∀ el1 el2 : even_list,
  elength (eapp el1 el2) = plus (elength el1) (elength el2).
  induction el1; crush.
```

One goal remains:

```
  n : nat
  o : odd_list
  el2 : even_list
  ============================
   S (olength (oapp o el2)) = S (plus (olength o) (elength el2))
```

We have no induction hypothesis, so we cannot prove this goal without starting another induction, which would reach a similar point, sending us into a futile infinite chain of inductions. The problem is that Coq's generation of *T_ind* principles is incomplete. We only get non-mutual induction principles generated by default.

```
Abort.
Check even_list_ind.

  even_list_ind
      : ∀ P : even_list → Prop,
        P ENil →
        (∀ (n : nat) (o : odd_list), P (ECons n o)) →
        ∀ e : even_list, P e
```

We see that no inductive hypotheses are included anywhere in the type. To get them, we must ask for mutual principles as we need them, using the `Scheme` command.

```
Scheme even_list_mut := Induction for even_list Sort Prop
with odd_list_mut := Induction for odd_list Sort Prop.

Check even_list_mut.

  even_list_mut
      : ∀ (P : even_list → Prop) (P0 : odd_list → Prop),
        P ENil →
        (∀ (n : nat) (o : odd_list), P0 o → P (ECons n o)) →
        (∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) →
        ∀ e : even_list, P e
```

This is the principle we wanted in the first place. There is one more wrinkle left in using it: the `induction` tactic will not apply it for us automatically. It will be helpful to look at how to prove one of our past examples without using `induction`, so that we can then generalize the technique to mutual inductive types.

```
Theorem n_plus_O' : ∀ n : nat, plus n O = n.
  apply (nat_ind (fun n ⇒ plus n O = n)); crush.
Qed.
```

From this example, we can see that `induction` is not magic. It only does some bookkeeping for us to make it easy to apply a theorem, which we can do directly with the `apply` tactic. We apply not just an identifier but a partial application of it, specifying the predicate we mean to prove holds for all naturals.

This technique generalizes to our mutual example:

```
Theorem elength_eapp : ∀ el1 el2 : even_list,
  elength (eapp el1 el2) = plus (elength el1) (elength el2).

  apply (even_list_mut
    (fun el1 : even_list ⇒ ∀ el2 : even_list,
```

```
    elength (eapp el1 el2) = plus (elength el1) (elength el2))
  (fun ol : odd_list ⇒ ∀ el : even_list,
    olength (oapp ol el) = plus (olength ol) (elength el))); crush.
Qed.
```

We simply need to specify two predicates, one for each of the mutually inductive types. In general, it would not be a good idea to assume that a proof assistant could infer extra predicates, so this way of applying mutual induction is about as straightforward as we could hope for.

3.5 Reflexive Types

A kind of inductive type called a *reflexive type* is defined in terms of functions that have the type being defined as their range. One very useful class of examples is in modeling variable binders. For instance, here is a type for encoding the syntax of a subset of first-order logic:

```
Inductive formula : Set :=
| Eq : nat → nat → formula
| And : formula → formula → formula
| Forall : (nat → formula) → formula.
```

Our kinds of formulas are equalities between naturals, conjunction, and universal quantification over natural numbers. We avoid needing to include a notion of "variables" in our type, by using Coq functions to encode quantification. For instance, here is the encoding of $\forall\ x$: **nat**, $x = x$:

```
Example forall_refl : formula := Forall (fun x ⇒ Eq x x).
```

We can write recursive functions over reflexive types quite naturally. Here is one translating our formulas into native Coq propositions.

```
Fixpoint formulaDenote (f : formula) : Prop :=
  match f with
    | Eq n1 n2 ⇒ n1 = n2
    | And f1 f2 ⇒ formulaDenote f1 ∧ formulaDenote f2
    | Forall f' ⇒ ∀ n : nat, formulaDenote (f' n)
  end.
```

We can also encode a trivial formula transformation that swaps the order of equality and conjunction operands.

```
Fixpoint swapper (f : formula) : formula :=
  match f with
    | Eq n1 n2 ⇒ Eq n2 n1
    | And f1 f2 ⇒ And (swapper f2) (swapper f1)
    | Forall f' ⇒ Forall (fun n ⇒ swapper (f' n))
  end.
```

It is helpful to prove that this transformation does not make true formulas false.

```
Theorem swapper_preserves_truth : ∀ f, formulaDenote f → formulaDenote (swapper f).
  induction f; crush.
Qed.
```

We can take a look at the induction principle behind this proof.

```
Check formula_ind.

formula_ind
     : ∀ P : formula → Prop,
       (∀ n n0 : nat, P (Eq n n0)) →
       (∀ f0 : formula,
        P f0 → ∀ f1 : formula, P f1 → P (And f0 f1)) →
       (∀ f1 : nat → formula,
        (∀ n : nat, P (f1 n)) → P (Forall f1)) →
       ∀ f2 : formula, P f2
```

Focusing on the Forall case, which comes third, we see that we are allowed to assume that the theorem holds *for any application of the argument function f1*. That is, Coq induction principles do not follow a simple rule that the textual representations of induction variables must get shorter in appeals to induction hypotheses. Luckily for us, the people behind the metatheory of Coq have verified that this flexibility does not introduce unsoundness.

Up to this point, we have seen how to encode in Coq more and more of what is possible with algebraic datatypes in Haskell and ML. This may have given the inaccurate impression that inductive types are a strict extension of algebraic datatypes. In fact, Coq must rule out some types allowed by Haskell and ML, for reasons of soundness. Reflexive types provide our first good example of such a case.

Given our last example of an inductive type, many readers are probably eager to try encoding the syntax of lambda calculus. Indeed, the function-based representation technique that we just used, called *higher-order abstract syntax (HOAS)*, is the representation of choice for lambda calculi in Twelf and in many applications implemented in Haskell and ML. Let us try to import that choice to Coq:

```
Inductive term : Set :=
| App : term → term → term
| Abs : (term → term) → term.

Error: Non strictly positive occurrence of "term" in "(term → term) → term"
```

We have run afoul of the *strict positivity requirement* for inductive definitions, which says that the type being defined may not occur to the left of an arrow in the type of a constructor argument. It is important that the type of a constructor is viewed in terms of a series of arguments and a result, since obviously we need recursive occurrences to the lefts of the

outermost arrows if we are to have recursive occurrences at all.

Why must Coq enforce this restriction? Imagine that our last definition had been accepted, allowing us to write this function:

```
Definition uhoh (t : term) : term :=
  match t with
    | Abs f ⇒ f t
    | _ ⇒ t
  end.
```

Using an informal idea of Coq's semantics, it is easy to verify that the application uhoh (Abs uhoh) will run forever. This would be a mere curiosity in OCaml and Haskell, where non-termination is commonplace, though the fact that we have a non-terminating program without explicit recursive function definitions is unusual.

For Coq, however, this would be a disaster. The possibility of writing such a function would destroy all our confidence that proving a theorem means anything. Since Coq combines programs and proofs in one language, we would be able to prove every theorem with an infinite loop.

Nonetheless, the basic insight of HOAS is a very useful one, and there are ways to realize most benefits of HOAS in Coq. We will study a particular technique of this kind in the later chapters on programming language syntax and semantics.

3.6 An Interlude on Proof Terms

As we have emphasized a few times already, Coq proofs are actually programs, written in the same language we have been using in our examples all along. We can get a first sense of what this means by taking a look at the definitions of some of the induction principles we have used.

```
Print unit_ind.

  unit_ind =
  fun P : unit → Prop ⇒ unit_rect P
      : ∀ P : unit → Prop, P tt → ∀ u : unit, P u
```

We see that this induction principle is defined in terms of a more general principle, unit_rect.

```
Check unit_rect.

  unit_rect
      : ∀ P : unit → Type, P tt → ∀ u : unit, P u
```

unit_rect gives P type unit → Type instead of unit → Prop. Type is another universe, like Set and Prop. In fact, it is a common supertype of both. Later on, we will discuss

exactly what the significances of the different universes are. For now, it is just important that we can use `Type` as a sort of meta-universe that may turn out to be either `Set` or `Prop`. We can see the symmetry inherent in the subtyping relationship by printing the definition of another principle that was generated for **unit** automatically:

```
Print unit_rec.

  unit_rec =
  fun P : unit → Set ⇒ unit_rect P
      : ∀ P : unit → Set, P tt → ∀ u : unit, P u
```

This is identical to the definition for unit_ind, except that we have substituted `Set` for `Prop`. For most inductive types T, then, we get not just induction principles *T_ind*, but also recursion principles *T_rec*. We can use *T_rec* to write recursive definitions without explicit `Fixpoint` recursion. For instance, the following two definitions are equivalent:

```
Definition always_O (u : unit) : nat :=
  match u with
    | tt ⇒ O
  end.

Definition always_O' (u : unit) : nat :=
  unit_rec (fun _ : unit ⇒ nat) O u.
```

Going even further down the rabbit hole, unit_rect itself is not even a primitive. It is a functional program that we can write manually.

```
Print unit_rect.

  unit_rect =
  fun (P : unit → Type) (f : P tt) (u : unit) ⇒
  match u as u0 return (P u0) with
  | tt ⇒ f
  end
      : ∀ P : unit → Type, P tt → ∀ u : unit, P u
```

The only new feature we see is an `as` clause for a `match`, which is used in concert with the `return` clause that we saw in the introduction. Since the type of the `match` is dependent on the value of the object being analyzed, we must give that object a name so that we can refer to it in the `return` clause.

To prove that unit_rect is nothing special, we can reimplement it manually.

```
Definition unit_rect' (P : unit → Type) (f : P tt) (u : unit) :=
  match u with
    | tt ⇒ f
  end.
```

We rely on Coq's heuristics for inferring `match` annotations.

We can check the implementation of *nat_rect* as well:

```
Print nat_rect.
```

```
nat_rect =
fun (P : nat → Type) (f : P O) (f0 : ∀ n : nat, P n → P (S n)) ⇒
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | O ⇒ f
  | S n0 ⇒ f0 n0 (F n0)
  end
   : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Now we have an actual recursive definition. *fix* expressions are an anonymous form of `Fixpoint`, just as `fun` expressions stand for anonymous non-recursive functions. Beyond that, the syntax of *fix* mirrors that of `Fixpoint`. We can understand the definition of *nat_rect* better by reimplementing nat_ind using sections.

```
Section nat_ind'.
```

First, we have the property of natural numbers that we aim to prove.

```
Variable P : nat → Prop.
```

Then we require a proof of the O case.

```
Hypothesis O_case : P O.
```

Next is a proof of the S case, which may assume an inductive hypothesis.

```
Hypothesis S_case : ∀ n : nat, P n → P (S n).
```

Finally, we define a recursive function to tie the pieces together.

```
Fixpoint nat_ind' (n : nat) : P n :=
  match n with
    | O ⇒ O_case
    | S n' ⇒ S_case (nat_ind' n')
  end.
End nat_ind'.
```

Closing the section adds the `Variable`s and `Hypothesis`es as new `fun`-bound arguments to nat_ind', and, modulo the use of `Prop` instead of `Type`, we end up with the exact same definition that was generated automatically for *nat_rect*.

We can also examine the definition of even_list_mut, which we generated with `Scheme` for a mutually-recursive type.

```
Print even_list_mut.
```

```
even_list_mut =
fun (P : even_list → Prop) (P0 : odd_list → Prop)
  (f : P ENil) (f0 : ∀ (n : nat) (o : odd_list), P0 o → P (ECons n o))
  (f1 : ∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) ⇒
fix F (e : even_list) : P e :=
```

```
    match e as e0 return (P e0) with
    | ENil ⇒ f
    | ECons n o ⇒ f0 n o (F0 o)
    end
  with F0 (o : odd_list) : P0 o :=
    match o as o0 return (P0 o0) with
    | OCons n e ⇒ f1 n e (F e)
    end
  for F
    : ∀ (P : even_list → Prop) (P0 : odd_list → Prop),
      P ENil →
      (∀ (n : nat) (o : odd_list), P0 o → P (ECons n o)) →
      (∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) →
      ∀ e : even_list, P e
```

We see a mutually-recursive *fix*, with the different functions separated by `with` in the same way that they would be separated by *and* in ML. A final `for` clause identifies which of the mutually-recursive functions should be the final value of the *fix* expression. Using this definition as a template, we can reimplement even_list_mut directly.

```
Section even_list_mut'.
```

First, we need the properties that we are proving.

```
Variable Peven : even_list → Prop.
Variable Podd : odd_list → Prop.
```

Next, we need proofs of the three cases.

```
Hypothesis ENil_case : Peven ENil.
Hypothesis ECons_case : ∀ (n : nat) (o : odd_list), Podd o → Peven (ECons n o).
Hypothesis OCons_case : ∀ (n : nat) (e : even_list), Peven e → Podd (OCons n e).
```

Finally, we define the recursive functions.

```
Fixpoint even_list_mut' (e : even_list) : Peven e :=
  match e with
    | ENil ⇒ ENil_case
    | ECons n o ⇒ ECons_case n (odd_list_mut' o)
  end
with odd_list_mut' (o : odd_list) : Podd o :=
  match o with
    | OCons n e ⇒ OCons_case n (even_list_mut' e)
  end.
End even_list_mut'.
```

Even induction principles for reflexive types are easy to implement directly. For our **formula** type, we can use a recursive definition much like those we wrote above.

```
Section formula_ind'.
```

```
  Variable P : formula → Prop.
  Hypothesis Eq_case : ∀ n1 n2 : nat, P (Eq n1 n2).
  Hypothesis And_case : ∀ f1 f2 : formula,
    P f1 → P f2 → P (And f1 f2).
  Hypothesis Forall_case : ∀ f : nat → formula,
    (∀ n : nat, P (f n)) → P (Forall f).

  Fixpoint formula_ind' (f : formula) : P f :=
    match f with
      | Eq n1 n2 ⇒ Eq_case n1 n2
      | And f1 f2 ⇒ And_case (formula_ind' f1) (formula_ind' f2)
      | Forall f' ⇒ Forall_case f' (fun n ⇒ formula_ind' (f' n))
    end.
End formula_ind'.
```

3.7 Nested Inductive Types

Suppose we want to extend our earlier type of binary trees to trees with arbitrary finite branching. We can use lists to give a simple definition.

```
Inductive nat_tree : Set :=
| NLeaf' : nat_tree
| NNode' : nat → list nat_tree → nat_tree.
```

This is an example of a *nested* inductive type definition, because we use the type we are defining as an argument to a parametrized type family. Coq will not allow all such definitions; it effectively pretends that we are defining **nat_tree** mutually with a version of **list** specialized to **nat_tree**, checking that the resulting expanded definition satisfies the usual rules. For instance, if we replaced **list** with a type family that used its parameter as a function argument, then the definition would be rejected as violating the positivity restriction.

Like we encountered for mutual inductive types, we find that the automatically-generated induction principle for **nat_tree** is too weak.

```
Check nat_tree_ind.

  nat_tree_ind
     : ∀ P : nat_tree → Prop,
       P NLeaf' →
       (∀ (n : nat) (l : list nat_tree), P (NNode' n l)) →
       ∀ n : nat_tree, P n
```

There is no command like `Scheme` that will implement an improved principle for us. In general, it takes creativity to figure out how to incorporate nested uses to different type families. Now that we know how to implement induction principles manually, we are in a

position to apply just such creativity to this problem.

First, we will need an auxiliary definition, characterizing what it means for a property to hold of every element of a list.

```
Section All.
  Variable T : Set.
  Variable P : T → Prop.

  Fixpoint All (ls : list T) : Prop :=
    match ls with
      | Nil ⇒ True
      | Cons h t ⇒ P h ∧ All t
    end.
End All.
```

It will be useful to look at the definitions of **True** and $\wedge$, since we will want to write manual proofs of them below.

```
Print True.
  Inductive True : Prop := I : True
```

That is, **True** is a proposition with exactly one proof, I, which we may always supply trivially.

Finding the definition of $\wedge$ takes a little more work. Coq supports user registration of arbitrary parsing rules, and it is such a rule that is letting us write $\wedge$ instead of an application of some inductive type family. We can find the underlying inductive type with the *Locate* command.

```
Locate "∧".
  Notation Scope
  "A ∧ B" := and A B : type_scope
                       (default interpretation)

Print and.
  Inductive and (A : Prop) (B : Prop) : Prop := conj : A → B → A ∧ B
  For conj: Arguments A, B are implicit
  For and: Argument scopes are [type_scope type_scope]
  For conj: Argument scopes are [type_scope type_scope _ _]
```

In addition to the definition of *and* itself, we get information on implicit arguments and parsing rules for *and* and its constructor conj. We will ignore the parsing information for now. The implicit argument information tells us that we build a proof of a conjunction by calling the constructor conj on proofs of the conjuncts, with no need to include the types of those proofs as explicit arguments.

Now we create a section for our induction principle, following the same basic plan as in the last section of this chapter.

```
Section nat_tree_ind'.
  Variable P : nat_tree → Prop.

  Hypothesis NLeaf'_case : P NLeaf'.
  Hypothesis NNode'_case : ∀ (n : nat) (ls : list nat_tree),
    All P ls → P (NNode' n ls).
```

A first attempt at writing the induction principle itself follows the intuition that nested inductive type definitions are expanded into mutual inductive definitions.

```
  Fixpoint nat_tree_ind' (tr : nat_tree) : P tr :=
    match tr with
      | NLeaf' ⇒ NLeaf'_case
      | NNode' n ls ⇒ NNode'_case n ls (list_nat_tree_ind ls)
    end

  with list_nat_tree_ind (ls : list nat_tree) : All P ls :=
    match ls with
      | Nil ⇒ I
      | Cons tr rest ⇒ conj (nat_tree_ind' tr) (list_nat_tree_ind rest)
    end.
```

Coq rejects this definition, saying "Recursive call to nat_tree_ind' has principal argument equal to "tr" instead of rest." The term "nested inductive type" hints at the solution to the problem. Just like true mutually-inductive types require mutually-recursive induction principles, nested types require nested recursion.

```
  Fixpoint nat_tree_ind' (tr : nat_tree) : P tr :=
    match tr with
      | NLeaf' ⇒ NLeaf'_case
      | NNode' n ls ⇒ NNode'_case n ls
        ((fix list_nat_tree_ind (ls : list nat_tree) : All P ls :=
          match ls with
            | Nil ⇒ I
            | Cons tr rest ⇒ conj (nat_tree_ind' tr) (list_nat_tree_ind rest)
          end) ls)
    end.
```

We include an anonymous *fix* version of *list_nat_tree_ind* that is literally *nested* inside the definition of the recursive function corresponding to the inductive definition that had the nested use of **list**.

```
End nat_tree_ind'.
```

We can try our induction principle out by defining some recursive functions on **nat_tree**s and proving a theorem about them. First, we define some helper functions that operate on lists.

```
Section map.
  Variables T T' : Set.
  Variable f : T → T'.

  Fixpoint map (ls : list T) : list T' :=
    match ls with
      | Nil ⇒ Nil
      | Cons h t ⇒ Cons (f h) (map t)
    end.
End map.

Fixpoint sum (ls : list nat) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons h t ⇒ plus h (sum t)
  end.
```

Now we can define a size function over our trees.

```
Fixpoint ntsize (tr : nat_tree) : nat :=
  match tr with
    | NLeaf' ⇒ S O
    | NNode' _ trs ⇒ S (sum (map ntsize trs))
  end.
```

Notice that Coq was smart enough to expand the definition of map to verify that we are using proper nested recursion, even through a use of a higher-order function.

```
Fixpoint ntsplice (tr1 tr2 : nat_tree) : nat_tree :=
  match tr1 with
    | NLeaf' ⇒ NNode' O (Cons tr2 Nil)
    | NNode' n Nil ⇒ NNode' n (Cons tr2 Nil)
    | NNode' n (Cons tr trs) ⇒ NNode' n (Cons (ntsplice tr tr2) trs)
  end.
```

We have defined another arbitrary notion of tree splicing, similar to before, and we can prove an analogous theorem about its relationship with tree size. We start with a useful lemma about addition.

```
Lemma plus_S : ∀ n1 n2 : nat,
  plus n1 (S n2) = S (plus n1 n2).
  induction n1; crush.
Qed.
```

Now we begin the proof of the theorem, adding the lemma plus_S as a hint.

```
Theorem ntsize_ntsplice : ∀ tr1 tr2 : nat_tree, ntsize (ntsplice tr1 tr2)
  = plus (ntsize tr2) (ntsize tr1).
  Hint Rewrite plus_S : cpdt.
```

We know that the standard induction principle is insufficient for the task, so we need to provide a `using` clause for the `induction` tactic to specify our alternate principle.

```
induction tr1 using nat_tree_ind'; crush.
```

One subgoal remains:

```
n : nat
ls : list nat_tree
H : All
      (fun tr1 : nat_tree =>
       ∀ tr2 : nat_tree,
       ntsize (ntsplice tr1 tr2) = plus (ntsize tr2) (ntsize tr1)) ls
tr2 : nat_tree
============================
 ntsize
   match ls with
   | Nil ⇒ NNode' n (Cons tr2 Nil)
   | Cons tr trs ⇒ NNode' n (Cons (ntsplice tr tr2) trs)
   end = S (plus (ntsize tr2) (sum (map ntsize ls)))
```

After a few moments of squinting at this goal, it becomes apparent that we need to do a case analysis on the structure of *ls*. The rest is routine.

```
destruct ls; crush.
```

We can go further in automating the proof by exploiting the hint mechanism.

```
Restart.
Hint Extern 1 (ntsize (match ?LS with Nil ⇒ _ | Cons _ _ ⇒ _ end) = _) ⇒
  destruct LS; crush.
induction tr1 using nat_tree_ind'; crush.
Qed.
```

We will go into great detail on hints in a later chapter, but the only important thing to note here is that we register a pattern that describes a conclusion we expect to encounter during the proof. The pattern may contain unification variables, whose names are prefixed with question marks, and we may refer to those bound variables in a tactic that we ask to have run whenever the pattern matches.

The advantage of using the hint is not very clear here, because the original proof was so short. However, the hint has fundamentally improved the readability of our proof. Before, the proof referred to the local variable *ls*, which has an automatically-generated name. To a human reading the proof script without stepping through it interactively, it was not clear where *ls* came from. The hint explains to the reader the process for choosing which variables to case analyze on, and the hint can continue working even if the rest of the proof structure changes significantly.

3.8 Manual Proofs About Constructors

It can be useful to understand how tactics like `discriminate` and `injection` work, so it is worth stepping through a manual proof of each kind. We will start with a proof fit for `discriminate`.

```
Theorem true_neq_false : true ≠ false.
```

We begin with the tactic `red`, which is short for "one step of reduction," to unfold the definition of logical negation.

```
red.
```

```
============================
 true = false → False
```

The negation is replaced with an implication of falsehood. We use the tactic `intro` H to change the assumption of the implication into a hypothesis named H.

```
intro H.
```

```
H : true = false
============================
 False
```

This is the point in the proof where we apply some creativity. We define a function whose utility will become clear soon.

```
Definition f (b : bool) := if b then True else False.
```

It is worth recalling the difference between the lowercase and uppercase versions of truth and falsehood: **True** and **False** are logical propositions, while true and false are boolean values that we can case-analyze. We have defined f such that our conclusion of **False** is computationally equivalent to f false. Thus, the *change* tactic will let us change the conclusion to f false.

```
change (f false).
```

```
H : true = false
============================
 f false
```

Now the righthand side of H's equality appears in the conclusion, so we can rewrite, using the notation ← to request to replace the righthand side the equality with the lefthand side.

```
rewrite ← H.
```

```
  H : true = false
  ============================
   f true
```

We are almost done. Just how close we are to done is revealed by computational simplification.

```
  simpl.
```

```
  H : true = false
  ============================
   True
```

```
  trivial.
Qed.
```

I have no trivial automated version of this proof to suggest, beyond using `discriminate` or `congruence` in the first place.

We can perform a similar manual proof of injectivity of the constructor S. I leave a walkthrough of the details to curious readers who want to run the proof script interactively.

```
Theorem S_inj' : ∀ n m : nat, S n = S m → n = m.
  intros n m H.
  change (pred (S n) = pred (S m)).
  rewrite H.
  reflexivity.
Qed.
```

3.9 Exercises

1. Define an inductive type *truth* with three constructors, *Yes*, *No*, and *Maybe*. *Yes* stands for certain truth, *No* for certain falsehood, and *Maybe* for an unknown situation. Define "not," "and," and "or" for this replacement boolean algebra. Prove that your implementation of "and" is commutative and distributes over your implementation of "or."

2. Modify the first example language of Chapter 2 to include variables, where variables are represented with **nat**. Extend the syntax and semantics of expressions to accommodate the change. Your new expDenote function should take as a new extra first argument a value of type $var \to$ **nat**, where *var* is a synonym for naturals-as-variables, and the function assigns a value to each variable. Define a constant folding function which does a bottom-up pass over an expression, at each stage replacing every binary operation on constants with an equivalent constant. Prove that constant folding preserves the meanings of expressions.

3. Reimplement the second example language of Chapter 2 to use mutually-inductive types instead of dependent types. That is, define two separate (non-dependent) inductive types *nat_exp* and *bool_exp* for expressions of the two different types, rather than a single indexed type. To keep things simple, you may consider only the binary operators that take naturals as operands. Add natural number variables to the language, as in the last exercise, and add an "if" expression form taking as arguments one boolean expression and two natural number expressions. Define semantics and constant-folding functions for this new language. Your constant folding should simplify not just binary operations (returning naturals or booleans) with known arguments, but also "if" expressions with known values for their test expressions but possibly undetermined "then" and "else" cases. Prove that constant-folding a natural number expression preserves its meaning.

4. Using a reflexive inductive definition, define a type **nat_tree** of infinitary trees, with natural numbers at their leaves and a countable infinity of new trees branching out of each internal node. Define a function *increment* that increments the number in every leaf of a **nat_tree**. Define a function *leapfrog* over a natural i and a tree *nt*. *leapfrog* should recurse into the ith child of *nt*, the i+1st child of that node, the i+2nd child of the next node, and so on, until reaching a leaf, in which case *leapfrog* should return the number at that leaf. Prove that the result of any call to *leapfrog* is incremented by one by calling *increment* on the tree.

5. Define a type of trees of trees of trees of (repeat to infinity). That is, define an inductive type *trexp*, whose members are either base cases containing natural numbers or binary trees of *trexp*s. Base your definition on a parameterized binary tree type *btree* that you will also define, so that *trexp* is defined as a nested inductive type. Define a function *total* that sums all of the naturals at the leaves of a *trexp*. Define a function *increment* that increments every leaf of a *trexp* by one. Prove that, for all *tr*, *total* (*increment tr*) $\geq$ *total tr*. On the way to finishing this proof, you will probably want to prove a lemma and add it as a hint using the syntax `Hint Resolve` *name_of_lemma.*.

6. Prove discrimination and injectivity theorems for the **nat_btree** type defined earlier in this chapter. In particular, without using the tactics `discriminate`, `injection`, or `congruence`, prove that no leaf equals any node, and prove that two equal nodes carry the same natural number.

Chapter 4

Inductive Predicates

The so-called "Curry-Howard Correspondence" states a formal connection between functional programs and mathematical proofs. In the last chapter, we snuck in a first introduction to this subject in Coq. Witness the close similarity between the types **unit** and **True** from the standard library:

```
Print unit.

  Inductive unit : Set := tt : unit

Print True.

  Inductive True : Prop := I : True
```

Recall that **unit** is the type with only one value, and **True** is the proposition that always holds. Despite this superficial difference between the two concepts, in both cases we can use the same inductive definition mechanism. The connection goes further than this. We see that we arrive at the definition of **True** by replacing **unit** by **True**, `tt` by `I`, and `Set` by `Prop`. The first two of these differences are superficial changes of names, while the third difference is the crucial one for separating programs from proofs. A term T of type `Set` is a type of programs, and a term of type T is a program. A term T of type `Prop` is a logical proposition, and its proofs are of type T.

unit has one value, `tt`. **True** has one proof, `I`. Why distinguish between these two types? Many people who have read about Curry-Howard in an abstract context and not put it to use in proof engineering answer that the two types in fact *should not* be distinguished. There is a certain aesthetic appeal to this point of view, but I want to argue that it is best to treat Curry-Howard very loosely in practical proving. There are Coq-specific reasons for preferring the distinction, involving efficient compilation and avoidance of paradoxes in the presence of classical math, but I will argue that there is a more general principle that should lead us to avoid conflating programming and proving.

The essence of the argument is roughly this: to an engineer, not all functions of type $A \to B$ are created equal, but all proofs of a proposition $P \to Q$ are. This idea is known as *proof irrelevance*, and its formalizations in logics prevent us from distinguishing between alternate proofs of the same proposition. Proof irrelevance is compatible with, but not derivable in,

Gallina. Apart from this theoretical concern, I will argue that it is most effective to do engineering with Coq by employing different techniques for programs versus proofs. Most of this book is organized around that distinction, describing how to program, by applying standard functional programming techniques in the presence of dependent types; and how to prove, by writing custom Ltac decision procedures.

With that perspective in mind, this chapter is sort of a mirror image of the last chapter, introducing how to define predicates with inductive definitions. We will point out similarities in places, but much of the effective Coq user's bag of tricks is disjoint for predicates versus "datatypes." This chapter is also a covert introduction to dependent types, which are the foundation on which interesting inductive predicates are built, though we will rely on tactics to build dependently-typed proof terms for us for now. A future chapter introduces more manual application of dependent types.

4.1 Propositional Logic

Let us begin with a brief tour through the definitions of the connectives for propositional logic. We will work within a Coq section that provides us with a set of propositional variables. In Coq parlance, these are just terms of type `Prop`.

```
Section Propositional.
  Variables P Q R : Prop.
```

In Coq, the most basic propositional connective is implication, written $\rightarrow$, which we have already used in almost every proof. Rather than being defined inductively, implication is built into Coq as the function type constructor.

We have also already seen the definition of **True**. For a demonstration of a lower-level way of establishing proofs of inductive predicates, we turn to this trivial theorem.

```
  Theorem obvious : True.
    apply I.
  Qed.
```

We may always use the `apply` tactic to take a proof step based on applying a particular constructor of the inductive predicate that we are trying to establish. Sometimes there is only one constructor that could possibly apply, in which case a shortcut is available:

```
  Theorem obvious' : True.
    constructor.
  Qed.
```

There is also a predicate **False**, which is the Curry-Howard mirror image of **Empty_set** from the last chapter.

```
  Print False.
  Inductive False : Prop :=
```

We can conclude anything from **False**, doing case analysis on a proof of **False** in the same way we might do case analysis on, say, a natural number. Since there are no cases to consider, any such case analysis succeeds immediately in proving the goal.

```
Theorem False_imp : False → 2 + 2 = 5.
  destruct 1.
Qed.
```

In a consistent context, we can never build a proof of **False**. In inconsistent contexts that appear in the courses of proofs, it is usually easiest to proceed by demonstrating that inconsistency with an explicit proof of **False**.

```
Theorem arith_neq : 2 + 2 = 5 → 9 + 9 = 835.
  intro.
```

At this point, we have an inconsistent hypothesis $2 + 2 = 5$, so the specific conclusion is not important. We use the `elimtype` tactic to state a proposition, telling Coq that we wish to construct a proof of the new proposition and then prove the original goal by case analysis on the structure of the new auxiliary proof. Since **False** has no constructors, `elimtype` **False** simply leaves us with the obligation to prove **False**.

```
  elimtype False.

H : 2 + 2 = 5
============================
 False
```

For now, we will leave the details of this proof about arithmetic to *crush*.

```
  crush.
Qed.
```

A related notion to **False** is logical negation.

```
Print not.
not = fun A : Prop ⇒ A → False
  : Prop → Prop
```

We see that not is just shorthand for implication of **False**. We can use that fact explicitly in proofs. The syntax ~P expands to not P.

```
Theorem arith_neq' : ¬ (2 + 2 = 5).
  unfold not.

============================
 2 + 2 = 5 → False
```

```
  crush.
Qed.
```

We also have conjunction, which we introduced in the last chapter.

```
Print and.
Inductive and (A : Prop) (B : Prop) : Prop := conj : A → B → A ∧ B
```

The interested reader can check that *and* has a Curry-Howard doppelganger called *prod*, the type of pairs. However, it is generally most convenient to reason about conjunction using tactics. An explicit proof of commutativity of *and* illustrates the usual suspects for such tasks. ∧ is an infix shorthand for *and*.

```
Theorem and_comm : P ∧ Q → Q ∧ P.
```

We start by case analysis on the proof of $P \wedge Q$.

```
 destruct 1.
```

```
H : P
H0 : Q
============================
 Q ∧ P
```

Every proof of a conjunction provides proofs for both conjuncts, so we get a single subgoal reflecting that. We can proceed by splitting this subgoal into a case for each conjunct of $Q \wedge P$.

```
 split.
```

```
2 subgoals
```

```
H : P
H0 : Q
============================
 Q
```

```
subgoal 2 is:
 P
```

In each case, the conclusion is among our hypotheses, so the `assumption` tactic finishes the process.

```
 assumption.
 assumption.
Qed.
```

Coq disjunction is called *or* and abbreviated with the infix operator ∨.

```
Print or.
Inductive or (A : Prop) (B : Prop) : Prop :=
```

or_introl : $A \to A \vee B$ | *or_intror* : $B \to A \vee B$

We see that there are two ways to prove a disjunction: prove the first disjunct or prove the second. The Curry-Howard analogue of this is the Coq sum type. We can demonstrate the main tactics here with another proof of commutativity.

```
Theorem or_comm : P ∨ Q → Q ∨ P.
```

As in the proof for *and*, we begin with case analysis, though this time we are met by two cases instead of one.

```
  destruct 1.
```

2 *subgoals*

```
  H : P
  ============================
   Q ∨ P

subgoal 2 is:
 Q ∨ P
```

We can see that, in the first subgoal, we want to prove the disjunction by proving its second disjunct. The right tactic telegraphs this intent.

```
  right; assumption.
```

The second subgoal has a symmetric proof.

```
1 subgoal

  H : Q
  ============================
   Q ∨ P

  left; assumption.
Qed.
```

It would be a shame to have to plod manually through all proofs about propositional logic. Luckily, there is no need. One of the most basic Coq automation tactics is `tauto`, which is a complete decision procedure for constructive propositional logic. (More on what "constructive" means in the next section.) We can use `tauto` to dispatch all of the purely propositional theorems we have proved so far.

```
Theorem or_comm' : P ∨ Q → Q ∨ P.
  tauto.
Qed.
```

Sometimes propositional reasoning forms important plumbing for the proof of a theorem,

but we still need to apply some other smarts about, say, arithmetic. `intuition` is a generalization of `tauto` that proves everything it can using propositional reasoning. When some goals remain, it uses propositional laws to simplify them as far as possible. Consider this example, which uses the list concatenation operator ++ from the standard library.

```
Theorem arith_comm : ∀ ls1 ls2 : list nat,
  length ls1 = length ls2 ∨ length ls1 + length ls2 = 6
  → length (ls1 ++ ls2) = 6 ∨ length ls1 = length ls2.
  intuition.
```

A lot of the proof structure has been generated for us by `intuition`, but the final proof depends on a fact about lists. The remaining subgoal hints at what cleverness we need to inject.

```
ls1 : list nat
ls2 : list nat
H0 : length ls1 + length ls2 = 6
============================
 length (ls1 ++ ls2) = 6 ∨ length ls1 = length ls2
```

We can see that we need a theorem about lengths of concatenated lists, which we proved last chapter and is also in the standard library.

```
  rewrite app_length.
```

```
ls1 : list nat
ls2 : list nat
H0 : length ls1 + length ls2 = 6
============================
 length ls1 + length ls2 = 6 ∨ length ls1 = length ls2
```

Now the subgoal follows by purely propositional reasoning. That is, we could replace length *ls1* + length *ls2* = 6 with P and length *ls1* = length *ls2* with Q and arrive at a tautology of propositional logic.

```
  tauto.
Qed.
```

`intuition` is one of the main bits of glue in the implementation of *crush*, so, with a little help, we can get a short automated proof of the theorem.

```
Theorem arith_comm' : ∀ ls1 ls2 : list nat,
  length ls1 = length ls2 ∨ length ls1 + length ls2 = 6
  → length (ls1 ++ ls2) = 6 ∨ length ls1 = length ls2.
  Hint Rewrite app_length : cpdt.

  crush.
```

```
  Qed.
End Propositional.
```

4.2 What Does It Mean to Be Constructive?

One potential point of confusion in the presentation so far is the distinction between **bool** and `Prop`. **bool** is a datatype whose two values are `true` and `false`, while `Prop` is a more primitive type that includes among its members **True** and **False**. Why not collapse these two concepts into one, and why must there be more than two states of mathematical truth?

The answer comes from the fact that Coq implements *constructive* or *intuitionistic* logic, in contrast to the *classical* logic that you may be more familiar with. In constructive logic, classical tautologies like $\neg \neg P \rightarrow P$ and $P \vee \neg P$ do not always hold. In general, we can only prove these tautologies when P is *decidable*, in the sense of computability theory. The Curry-Howard encoding that Coq uses for *or* allows us to extract either a proof of P or a proof of $\neg P$ from any proof of $P \vee \neg P$. Since our proofs are just functional programs which we can run, this would give us a decision procedure for the halting problem, where the instantiations of P would be formulas like "this particular Turing machine halts."

Hence the distinction between **bool** and `Prop`. Programs of type **bool** are computational by construction; we can always run them to determine their results. Many `Prop`s are undecidable, and so we can write more expressive formulas with `Prop`s than with **bool**s, but the inevitable consequence is that we cannot simply "run a `Prop` to determine its truth."

Constructive logic lets us define all of the logical connectives in an aesthetically-appealing way, with orthogonal inductive definitions. That is, each connective is defined independently using a simple, shared mechanism. Constructivity also enables a trick called *program extraction*, where we write programs by phrasing them as theorems to be proved. Since our proofs are just functional programs, we can extract executable programs from our final proofs, which we could not do as naturally with classical proofs.

We will see more about Coq's program extraction facility in a later chapter. However, I think it is worth interjecting another warning at this point, following up on the prior warning about taking the Curry-Howard correspondence too literally. It is possible to write programs by theorem-proving methods in Coq, but hardly anyone does it. It is almost always most useful to maintain the distinction between programs and proofs. If you write a program by proving a theorem, you are likely to run into algorithmic inefficiencies that you introduced in your proof to make it easier to prove. It is a shame to have to worry about such situations while proving tricky theorems, and it is a happy state of affairs that you almost certainly will not need to, with the ideal of extracting programs from proofs being confined mostly to theoretical studies.

4.3 First-Order Logic

The $\forall$ connective of first-order logic, which we have seen in many examples so far, is built into Coq. Getting ahead of ourselves a bit, we can see it as the dependent function type constructor. In fact, implication and universal quantification are just different syntactic shorthands for the same Coq mechanism. A formula $P \to Q$ is equivalent to $\forall\ x : P,\ Q$, where x does not appear in Q. That is, the "real" type of the implication says "for every proof of P, there exists a proof of Q."

Existential quantification is defined in the standard library.

```
Print ex.
```

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
  ex_intro : ∀ x : A, P x → ex P
```

ex is parameterized by the type A that we quantify over, and by a predicate P over As. We prove an existential by exhibiting some x of type A, along with a proof of $P\ x$. As usual, there are tactics that save us from worrying about the low-level details most of the time. We use the equality operator $=$, which, depending on the settings in which they learned logic, different people will say either is or is not part of first-order logic. For our purposes, it is.

```
Theorem exist1 : ∃ x : nat, x + 1 = 2.
```

We can start this proof with a tactic *exists*, which should not be confused with the formula constructor shorthand of the same name. (In the PDF version of this document, the reverse 'E' appears instead of the text "exists" in formulas.)

```
  exists 1.
```

The conclusion is replaced with a version using the existential witness that we announced.

```
  ============================
   1 + 1 = 2

  reflexivity.
Qed.
```

We can also use tactics to reason about existential hypotheses.

```
Theorem exist2 : ∀ n m : nat, (∃ x : nat, n + x = m) → n ≤ m.
```

We start by case analysis on the proof of the existential fact.

```
  destruct 1.

  n : nat
  m : nat
  x : nat
  H : n + x = m
  ============================
```

$n \leq m$

The goal has been replaced by a form where there is a new free variable x, and where we have a new hypothesis that the body of the existential holds with x substituted for the old bound variable. From here, the proof is just about arithmetic and is easy to automate.

```
  crush.
Qed.
```

The tactic `intuition` has a first-order cousin called `firstorder`. `firstorder` proves many formulas when only first-order reasoning is needed, and it tries to perform first-order simplifications in any case. First-order reasoning is much harder than propositional reasoning, so `firstorder` is much more likely than `intuition` to get stuck in a way that makes it run for long enough to be useless.

4.4 Predicates with Implicit Equality

We start our exploration of a more complicated class of predicates with a simple example: an alternative way of characterizing when a natural number is zero.

```
Inductive isZero : nat → Prop :=
| IsZero : isZero 0.

Theorem isZero_zero : isZero 0.
  constructor.
Qed.
```

We can call **isZero** a *judgment*, in the sense often used in the semantics of programming languages. Judgments are typically defined in the style of *natural deduction*, where we write a number of *inference rules* with premises appearing above a solid line and a conclusion appearing below the line. In this example, the sole constructor IsZero of **isZero** can be thought of as the single inference rule for deducing **isZero**, with nothing above the line and **isZero** 0 below it. The proof of isZero_zero demonstrates how we can apply an inference rule.

The definition of **isZero** differs in an important way from all of the other inductive definitions that we have seen in this and the previous chapter. Instead of writing just `Set` or `Prop` after the colon, here we write **nat** → `Prop`. We saw examples of parameterized types like **list**, but there the parameters appeared with names *before* the colon. Every constructor of a parameterized inductive type must have a range type that uses the same parameter, whereas the form we use here enables us to use different arguments to the type for different constructors.

For instance, **isZero** forces its argument to be 0. We can see that the concept of equality is somehow implicit in the inductive definition mechanism. The way this is accomplished is similar to the way that logic variables are used in Prolog, and it is a very powerful mechanism that forms a foundation for formalizing all of mathematics. In fact, though it is natural to

think of inductive types as folding in the functionality of equality, in Coq, the true situation is reversed, with equality defined as just another inductive type!

```
Print eq.
```

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

eq is the type we get behind the scenes when uses of infix = are expanded. We see that **eq** has both a parameter x that is fixed and an extra unnamed argument of the same type. The type of **eq** allows us to state any equalities, even those that are provably false. However, examining the type of equality's sole constructor refl_equal, we see that we can only *prove* equality when its two arguments are syntactically equal. This definition turns out to capture all of the basic properties of equality, and the equality-manipulating tactics that we have seen so far, like `reflexivity` and `rewrite`, are implemented treating **eq** as just another inductive type with a well-chosen definition.

Returning to the example of **isZero**, we can see how to make use of hypotheses that use this predicate.

```
Theorem isZero_plus : ∀ n m : nat, isZero m → n + m = n.
```

We want to proceed by cases on the proof of the assumption about **isZero**.

```
  destruct 1.
```

```
n : nat
============================
 n + 0 = n
```

Since **isZero** has only one constructor, we are presented with only one subgoal. The argument m to **isZero** is replaced with that type's argument from the single constructor IsZero. From this point, the proof is trivial.

```
  crush.
Qed.
```

Another example seems at first like it should admit an analogous proof, but in fact provides a demonstration of one of the most basic gotchas of Coq proving.

```
Theorem isZero_contra : isZero 1 → False.
```

Let us try a proof by cases on the assumption, as in the last proof.

```
  destruct 1.
```

```
============================
 False
```

It seems that case analysis has not helped us much at all! Our sole hypothesis disappears, leaving us, if anything, worse off than we were before. What went wrong? We have met an important restriction in tactics like `destruct` and `induction` when applied to types with

arguments. If the arguments are not already free variables, they will be replaced by new free variables internally before doing the case analysis or induction. Since the argument 1 to **isZero** is replaced by a fresh variable, we lose the crucial fact that it is not equal to 0.

Why does Coq use this restriction? We will discuss the issue in detail in a future chapter, when we see the dependently-typed programming techniques that would allow us to write this proof term manually. For now, we just say that the algorithmic problem of "logically complete case analysis" is undecidable when phrased in Coq's logic. A few tactics and design patterns that we will present in this chapter suffice in almost all cases. For the current example, what we want is a tactic called `inversion`, which corresponds to the concept of inversion that is frequently used with natural deduction proof systems.

```
  Undo.
  inversion 1.
Qed.
```

What does `inversion` do? Think of it as a version of `destruct` that does its best to take advantage of the structure of arguments to inductive types. In this case, `inversion` completed the proof immediately, because it was able to detect that we were using **isZero** with an impossible argument.

Sometimes using `destruct` when you should have used `inversion` can lead to confusing results. To illustrate, consider an alternate proof attempt for the last theorem.

```
Theorem isZero_contra' : isZero 1 → 2 + 2 = 5.
  destruct 1.

  ============================
   1 + 1 = 4
```

What on earth happened here? Internally, `destruct` replaced 1 with a fresh variable, and, trying to be helpful, it also replaced the occurrence of 1 within the unary representation of each number in the goal. This has the net effect of decrementing each of these numbers. If you are doing a proof and encounter a strange transmutation like this, there is a good chance that you should go back and replace a use of `destruct` with `inversion`.

```
Abort.
```

4.5 Recursive Predicates

We have already seen all of the ingredients we need to build interesting recursive predicates, like this predicate capturing even-ness.

```
Inductive even : nat → Prop :=
| EvenO : even O
| EvenSS : ∀ n, even n → even (S (S n)).
```

Think of **even** as another judgment defined by natural deduction rules. EvenO is a rule with nothing above the line and **even** O below the line, and EvenSS is a rule with **even** *n* above the line and **even** $(S\ (S\ n))$ below.

The proof techniques of the last section are easily adapted.

```
Theorem even_0 : even 0.
  constructor.
Qed.

Theorem even_4 : even 4.
  constructor; constructor; constructor.
Qed.
```

It is not hard to see that sequences of constructor applications like the above can get tedious. We can avoid them using Coq's hint facility.

```
Hint Constructors even.

Theorem even_4' : even 4.
  auto.
Qed.

Theorem even_1_contra : even 1 → False.
  inversion 1.
Qed.

Theorem even_3_contra : even 3 → False.
  inversion 1.

  H : even 3
  n : nat
  H1 : even 1
  H0 : n = 1
  ============================
  False
```

`inversion` can be a little overzealous at times, as we can see here with the introduction of the unused variable *n* and an equality hypothesis about it. For more complicated predicates, though, adding such assumptions is critical to dealing with the undecidability of general inversion.

```
  inversion H1.
Qed.
```

We can also do inductive proofs about **even**.

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

It seems a reasonable first choice to proceed by induction on *n*.

```
  induction n; crush.
```

```
n : nat
IHn : ∀ m : nat, even n → even m → even (n + m)
m : nat
H : even (S n)
H0 : even m
============================
 even (S (n + m))
```

We will need to use the hypotheses *H* and *H0* somehow. The most natural choice is to invert *H*.

```
inversion H.
```

```
n : nat
IHn : ∀ m : nat, even n → even m → even (n + m)
m : nat
H : even (S n)
H0 : even m
n0 : nat
H2 : even n0
H1 : S n0 = n
============================
 even (S (S n0 + m))
```

Simplifying the conclusion brings us to a point where we can apply a constructor.

```
simpl.
```

```
============================
 even (S (S (n0 + m)))
constructor.
```

```
============================
 even (n0 + m)
```

At this point, we would like to apply the inductive hypothesis, which is:

```
IHn : ∀ m : nat, even n → even m → even (n + m)
```

Unfortunately, the goal mentions *n0* where it would need to mention *n* to match *IHn*. We could keep looking for a way to finish this proof from here, but it turns out that we can make our lives much easier by changing our basic strategy. Instead of inducting on the

structure of *n*, we should induct *on the structure of one of the* ***even*** *proofs*. This technique is commonly called *rule induction* in programming language semantics. In the setting of Coq, we have already seen how predicates are defined using the same inductive type mechanism as datatypes, so the fundamental unity of rule induction with "normal" induction is apparent.

Restart.

```
induction 1.
```

```
m : nat
============================
 even m → even (0 + m)

subgoal 2 is:
 even m → even (S (S n) + m)
```

The first case is easily discharged by *crush*, based on the hint we added earlier to try the constructors of **even**.

```
crush.
```

Now we focus on the second case:

```
intro.
```

```
m : nat
n : nat
H : even n
IHeven : even m → even (n + m)
H0 : even m
============================
 even (S (S n) + m)
```

We simplify and apply a constructor, as in our last proof attempt.

```
simpl; constructor.
```

```
============================
 even (n + m)
```

Now we have an exact match with our inductive hypothesis, and the remainder of the proof is trivial.

```
apply IHeven; assumption.
```

In fact, *crush* can handle all of the details of the proof once we declare the induction strategy.

Restart.

```
  induction 1; crush.
Qed.
```

Induction on recursive predicates has similar pitfalls to those we encountered with inversion in the last section.

```
Theorem even_contra : ∀ n, even (S (n + n)) → False.
  induction 1.

  n : nat
  ============================
   False

subgoal 2 is:
 False
```

We are already sunk trying to prove the first subgoal, since the argument to **even** was replaced by a fresh variable internally. This time, we find it easiest to prove this theorem by way of a lemma. Instead of trusting `induction` to replace expressions with fresh variables, we do it ourselves, explicitly adding the appropriate equalities as new assumptions.

```
Abort.

Lemma even_contra' : ∀ n', even n' → ∀ n, n' = S (n + n) → False.
  induction 1; crush.
```

At this point, it is useful to consider all cases of *n* and *n0* being zero or nonzero. Only one of these cases has any trickiness to it.

```
  destruct n; destruct n0; crush.

  n : nat
  H : even (S n)
  IHeven : ∀ n0 : nat, S n = S (n0 + n0) → False
  n0 : nat
  H0 : S n = n0 + S n0
  ============================
   False
```

At this point it is useful to use a theorem from the standard library, which we also proved with a different name in the last chapter.

```
  Check plus_n_Sm.

  plus_n_Sm
      : ∀ n m : nat, S (n + m) = n + S m

  rewrite ← plus_n_Sm in H0.
```

The induction hypothesis lets us complete the proof.

```
  apply IHeven with n0; assumption.
```

As usual, we can rewrite the proof to avoid referencing any locally-generated names, which makes our proof script more readable and more robust to changes in the theorem statement. We use the notation ← to request a hint that does right-to-left rewriting, just like we can with the `rewrite` tactic.

```
  Restart.
  Hint Rewrite ← plus_n_Sm : cpdt.

  induction 1; crush;
    match goal with
      | [ H : S ?N = ?N0 + ?N0 ⊢ _ ] ⇒ destruct N; destruct N0
    end; crush; eauto.
Qed.
```

We write the proof in a way that avoids the use of local variable or hypothesis names, using the `match` tactic form to do pattern-matching on the goal. We use unification variables prefixed by question marks in the pattern, and we take advantage of the possibility to mention a unification variable twice in one pattern, to enforce equality between occurrences. The hint to rewrite with plus_n_Sm in a particular direction saves us from having to figure out the right place to apply that theorem, and we also take critical advantage of a new tactic, `eauto`.

crush uses the tactic `intuition`, which, when it runs out of tricks to try using only propositional logic, by default tries the tactic `auto`, which we saw in an earlier example. `auto` attempts Prolog-style logic programming, searching through all proof trees up to a certain depth that are built only out of hints that have been registered with `Hint` commands. Compared to Prolog, `auto` places an important restriction: it never introduces new unification variables during search. That is, every time a rule is applied during proof search, all of its arguments must be deducible by studying the form of the goal. `eauto` relaxes this restriction, at the cost of possibly exponentially greater running time. In this particular case, we know that `eauto` has only a small space of proofs to search, so it makes sense to run it. It is common in effectively-automated Coq proofs to see a bag of standard tactics applied to pick off the "easy" subgoals, finishing with `eauto` to handle the tricky parts that can benefit from ad-hoc exhaustive search.

The original theorem now follows trivially from our lemma.

```
Theorem even_contra : ∀ n, even (S (n + n)) → False.
  intros; eapply even_contra'; eauto.
Qed.
```

We use a variant `eapply` of `apply` which has the same relationship to `apply` as `eauto` has to `auto`. `apply` only succeeds if all arguments to the rule being used can be determined from the form of the goal, whereas `eapply` will introduce unification variables for undetermined arguments. `eauto` is able to determine the right values for those unification variables.

By considering an alternate attempt at proving the lemma, we can see another common pitfall of inductive proofs in Coq. Imagine that we had tried to prove even_contra' with all of the ∀ quantifiers moved to the front of the lemma statement.

```
Lemma even_contra'' : ∀ n' n, even n' → n' = S (n + n) → False.
  induction 1; crush;
    match goal with
      | [ H : S ?N = ?N0 + ?N0 ⊢ _ ] ⇒ destruct N; destruct N0
    end; crush; eauto.
```

One subgoal remains:

```
n : nat
H : even (S (n + n))
IHeven : S (n + n) = S (S (S (n + n))) → False
============================
 False
```

We are out of luck here. The inductive hypothesis is trivially true, since its assumption is false. In the version of this proof that succeeded, *IHeven* had an explicit quantification over *n*. This is because the quantification of *n* *appeared after the thing we are inducting on* in the theorem statement. In general, quantified variables and hypotheses that appear before the induction object in the theorem statement stay fixed throughout the inductive proof. Variables and hypotheses that are quantified after the induction object may be varied explicitly in uses of inductive hypotheses.

Why should Coq implement `induction` this way? One answer is that it avoids burdening this basic tactic with additional heuristic smarts, but that is not the whole picture. Imagine that `induction` analyzed dependencies among variables and reordered quantifiers to preserve as much freedom as possible in later uses of inductive hypotheses. This could make the inductive hypotheses more complex, which could in turn cause particular automation machinery to fail when it would have succeeded before. In general, we want to avoid quantifiers in our proofs whenever we can, and that goal is furthered by the refactoring that the `induction` tactic forces us to do.

```
Abort.
```

4.6 Exercises

1. Prove these tautologies of propositional logic, using only the tactics `apply`, `assumption`, **constructor**, `destruct`, `intro`, `intros`, left, right, `split`, and `unfold`.
   (a) (**True** ∨ **False**) ∧ (**False** ∨ **True**)
   (b) $P \to \neg \neg P$
   (c) $P \wedge (Q \vee R) \to (P \wedge Q) \vee (P \wedge R)$
2. Prove the following tautology of first-order logic, using only the tactics `apply`, `assert`, `assumption`, `destruct`, `eapply`, *eassumption*, and *exists*. You will probably find

`assert` useful for stating and proving an intermediate lemma, enabling a kind of "forward reasoning," in contrast to the "backward reasoning" that is the default for Coq tactics. *eassumption* is a version of `assumption` that will do matching of unification variables. Let some variable T of type `Set` be the set of individuals. x is a constant symbol, p is a unary predicate symbol, q is a binary predicate symbol, and f is a unary function symbol.

(a) $p\ x \to (\forall\ x,\ p\ x \to \exists\ y,\ q\ x\ y) \to (\forall\ x\ y,\ q\ x\ y \to q\ y\ (\mathsf{f}\ y)) \to \exists\ z,\ q\ z\ (\mathsf{f}\ z)$

3. Define an inductive predicate capturing when a natural number is an integer multiple of either 6 or 10. Prove that 13 does not satisfy your predicate, and prove that any number satisfying the predicate is not odd. It is probably easiest to prove the second theorem by indicating "odd-ness" as equality to $2 \times n + 1$ for some n.

4. Define a simple programming language, its semantics, and its typing rules, and then prove that well-typed programs cannot go wrong. Specifically:

   (a) Define *var* as a synonym for the natural numbers.

   (b) Define an inductive type exp of expressions, containing natural number constants, natural number addition, pairing of two other expressions, extraction of the first component of a pair, extraction of the second component of a pair, and variables (based on the *var* type you defined).

   (c) Define an inductive type *cmd* of commands, containing expressions and variable assignments. A variable assignment node should contain the variable being assigned, the expression being assigned to it, and the command to run afterward.

   (d) Define an inductive type val of values, containing natural number constants and pairings of values.

   (e) Define a type of variable assignments, which assign a value to each variable.

   (f) Define a big-step evaluation relation eval, capturing what it means for an expression to evaluate to a value under a particular variable assignment. "Big step" means that the evaluation of every expression should be proved with a single instance of the inductive predicate you will define. For instance, "1 + 1 evaluates to 2 under assignment *va*" should be derivable for any assignment *va*.

   (g) Define a big-step evaluation relation *run*, capturing what it means for a command to run to a value under a particular variable assignment. The value of a command is the result of evaluating its final expression.

   (h) Define a type of variable typings, which are like variable assignments, but map variables to types instead of values. You might use polymorphism to share some code with your variable assignments.

   (i) Define typing judgments for expressions, values, and commands. The expression and command cases will be in terms of a typing assignment.

(j) Define a predicate *varsType* to express when a variable assignment and a variable typing agree on the types of variables.

(k) Prove that any expression that has type t under variable typing vt evaluates under variable assignment va to some value that also has type t in vt, as long as va and vt agree.

(l) Prove that any command that has type t under variable typing vt evaluates under variable assignment va to some value that also has type t in vt, as long as va and vt agree.

A few hints that may be helpful:

(a) One easy way of defining variable assignments and typings is to define both as instances of a polymorphic map type. The map type at parameter T can be defined to be the type of arbitrary functions from variables to T. A helpful function for implementing insertion into such a functional map is `eq_nat_dec`, which you can make available with `Require Import` *Arith.*. `eq_nat_dec` has a dependent type that tells you that it makes accurate decisions on whether two natural numbers are equal, but you can use it as if it returned a boolean, e.g., `if eq_nat_dec` *n m* `then` *E1* `else` *E2*.

(b) If you follow the last hint, you may find yourself writing a proof that involves an expression with `eq_nat_dec` that you would like to simplify. Running `destruct` on the particular call to `eq_nat_dec` should do the trick. You can automate this advice with a piece of Ltac:

```
match goal with
  | [ ⊢ context[eq_nat_dec ?X ?Y] ] ⇒ destruct (eq_nat_dec X Y)
end
```

(c) You probably do not want to use an inductive definition for compatibility of variable assignments and typings.

(d) The Tactics module from this book contains a variant *crush'* of *crush*. *crush'* takes two arguments. The first argument is a list of lemmas and other functions to be tried automatically in "forward reasoning" style, where we add new facts without being sure yet that they link into a proof of the conclusion. The second argument is a list of predicates on which inverison should be attempted automatically. For instance, running *crush'* (lemma1, lemma2) pred will search for chances to apply lemma1 and lemma2 to hypotheses that are already available, adding the new concluded fact if suitable hypotheses can be found. Inversion will be attempted on any hypothesis using pred, but only those inversions that narrow the field of possibilities to one possible rule will be kept. The format of the list arguments to *crush'* is that you can pass an empty list as tt, a singleton list as the unadorned single element, and a multiple-element list as a tuple of the elements.

(e) If you want *crush'* to apply polymorphic lemmas, you may have to do a little extra work, if the type parameter is not a free variable of your proof context (so that *crush'* does not know to try it). For instance, if you define a polymorphic map insert function *assign* of some type $\forall\ T : \texttt{Set}$, ..., and you want particular applications of *assign* added automatically with type parameter U, you would need to include *assign* in the lemma list as *assign* U (if you have implicit arguments off) or *assign* $(T := U)$ or @*assign* U (if you have implicit arguments on).

Chapter 5

Infinite Data and Proofs

In lazy functional programming languages like Haskell, infinite data structures are everywhere. Infinite lists and more exotic datatypes provide convenient abstractions for communication between parts of a program. Achieving similar convenience without infinite lazy structures would, in many cases, require acrobatic inversions of control flow.

Laziness is easy to implement in Haskell, where all the definitions in a program may be thought of as mutually recursive. In such an unconstrained setting, it is easy to implement an infinite loop when you really meant to build an infinite list, where any finite prefix of the list should be forceable in finite time. Haskell programmers learn how to avoid such slip-ups. In Coq, such a laissez-faire policy is not good enough.

We spent some time in the last chapter discussing the Curry-Howard isomorphism, where proofs are identified with functional programs. In such a setting, infinite loops, intended or otherwise, are disastrous. If Coq allowed the full breadth of definitions that Haskell did, we could code up an infinite loop and use it to prove any proposition vacuously. That is, the addition of general recursion would make CIC *inconsistent*. For an arbitrary proposition P, we could write:

```
Fixpoint bad (u : unit) : P := bad u.
```

This would leave us with *bad* `tt` as a proof of P.

There are also algorithmic considerations that make universal termination very desirable. We have seen how tactics like `reflexivity` compare terms up to equivalence under computational rules. Calls to recursive, pattern-matching functions are simplified automatically, with no need for explicit proof steps. It would be very hard to hold onto that kind of benefit if it became possible to write non-terminating programs; we would be running smack into the halting problem.

One solution is to use types to contain the possibility of non-termination. For instance, we can create a "non-termination monad," inside which we must write all of our general-recursive programs. This is a heavyweight solution, and so we would like to avoid it whenever possible.

Luckily, Coq has special support for a class of lazy data structures that happens to contain most examples found in Haskell. That mechanism, *co-inductive types*, is the subject of this chapter.

5.1 Computing with Infinite Data

Let us begin with the most basic type of infinite data, *streams*, or lazy lists.

```
Section stream.
  Variable A : Set.

  CoInductive stream : Set :=
  | Cons : A → stream → stream.
End stream.
```

The definition is surprisingly simple. Starting from the definition of **list**, we just need to change the keyword `Inductive` to `CoInductive`. We could have left a Nil constructor in our definition, but we will leave it out to force all of our streams to be infinite.

How do we write down a stream constant? Obviously simple application of constructors is not good enough, since we could only denote finite objects that way. Rather, whereas recursive definitions were necessary to *use* values of recursive inductive types effectively, here we find that we need *co-recursive definitions* to *build* values of co-inductive types effectively.

We can define a stream consisting only of zeroes.

```
CoFixpoint zeroes : stream nat := Cons 0 zeroes.
```

We can also define a stream that alternates between true and false.

```
CoFixpoint trues : stream bool := Cons true falses
with falses : stream bool := Cons false trues.
```

Co-inductive values are fair game as arguments to recursive functions, and we can use that fact to write a function to take a finite approximation of a stream.

```
Fixpoint approx A (s : stream A) (n : nat) : list A :=
  match n with
    | O ⇒ nil
    | S n' ⇒
      match s with
        | Cons h t ⇒ h :: approx t n'
      end
  end.

Eval simpl in approx zeroes 10.
     = 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: nil
     : list nat

Eval simpl in approx trues 10.
```

```
= true
  :: false
     :: true
        :: false
           :: true :: false :: true :: false :: true :: false :: nil
: list bool
```

So far, it looks like co-inductive types might be a magic bullet, allowing us to import all of the Haskeller's usual tricks. However, there are important restrictions that are dual to the restrictions on the use of inductive types. Fixpoints *consume* values of inductive types, with restrictions on which *arguments* may be passed in recursive calls. Dually, co-fixpoints *produce* values of co-inductive types, with restrictions on what may be done with the *results* of co-recursive calls.

The restriction for co-inductive types shows up as the *guardedness condition*, and it can be broken into two parts. First, consider this stream definition, which would be legal in Haskell.

```
CoFixpoint looper : stream nat := looper.

Error:
Recursive definition of looper is ill-formed.
In environment
looper : stream nat

unguarded recursive call in "looper"
```

The rule we have run afoul of here is that *every co-recursive call must be guarded by a constructor*; that is, every co-recursive call must be a direct argument to a constructor of the co-inductive type we are generating. It is a good thing that this rule is enforced. If the definition of *looper* were accepted, our approx function would run forever when passed *looper*, and we would have fallen into inconsistency.

The second rule of guardedness is easiest to see by first introducing a more complicated, but legal, co-fixpoint.

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

  CoFixpoint map (s : stream A) : stream B :=
    match s with
      | Cons h t ⇒ Cons (f h) (map t)
    end.
End map.
```

This code is a literal copy of that for the list map function, with the Nil case removed and

`Fixpoint` changed to `CoFixpoint`. Many other standard functions on lazy data structures can be implemented just as easily. Some, like *filter*, cannot be implemented. Since the predicate passed to *filter* may reject every element of the stream, we cannot satisfy even the first guardedness condition.

The second condition is subtler. To illustrate it, we start off with another co-recursive function definition that *is* legal. The function interleave takes two streams and produces a new stream that alternates between their elements.

```
Section interleave.
  Variable A : Set.

  CoFixpoint interleave (s1 s2 : stream A) : stream A :=
    match s1, s2 with
      | Cons h1 t1, Cons h2 t2 ⇒ Cons h1 (Cons h2 (interleave t1 t2))
    end.
End interleave.
```

Now say we want to write a weird stuttering version of map that repeats elements in a particular way, based on interleaving.

```
Section map'.
  Variables A B : Set.
  Variable f : A → B.

  CoFixpoint map' (s : stream A) : stream B :=
    match s with
      | Cons h t ⇒ interleave (Cons (f h) (map' s)) (Cons (f h) (map' s))
    end.
```

We get another error message about an unguarded recursive call. This is because we are violating the second guardedness condition, which says that, not only must co-recursive calls be arguments to constructors, there must also *not be anything but* `match`*es and calls to constructors of the same co-inductive type* wrapped around these immediate uses of co-recursive calls. The actual implemented rule for guardedness is a little more lenient than what we have just stated, but you can count on the illegality of any exception that would enhance the expressive power of co-recursion.

Why enforce a rule like this? Imagine that, instead of interleave, we had called some other, less well-behaved function on streams. Perhaps this other function might be defined mutually with map'. It might deconstruct its first argument, retrieving map' s from within Cons (f h) (map' s). Next it might try a `match` on this retrieved value, which amounts to deconstructing map' s. To figure out how this `match` turns out, we need to know the top-level structure of map' s, but this is exactly what we started out trying to determine! We run into a loop in the evaluation process, and we have reached a witness of inconsistency if we are evaluating approx (map' s) 1 for any s.

```
End map'.
```

5.2 Infinite Proofs

Let us say we want to give two different definitions of a stream of all ones, and then we want to prove that they are equivalent.

```
CoFixpoint ones : stream nat := Cons 1 ones.
Definition ones' := map S zeroes.
```

The obvious statement of the equality is this:

```
Theorem ones_eq : ones = ones'.
```

However, faced with the initial subgoal, it is not at all clear how this theorem can be proved. In fact, it is unprovable. The **eq** predicate that we use is fundamentally limited to equalities that can be demonstrated by finite, syntactic arguments. To prove this equivalence, we will need to introduce a new relation.

```
Abort.
```

Co-inductive datatypes make sense by analogy from Haskell. What we need now is a *co-inductive proposition*. That is, we want to define a proposition whose proofs may be infinite, subject to the guardedness condition. The idea of infinite proofs does not show up in usual mathematics, but it can be very useful (unsurprisingly) for reasoning about infinite data structures. Besides examples from Haskell, infinite data and proofs will also turn out to be useful for modelling inherently infinite mathematical objects, like program executions.

We are ready for our first co-inductive predicate.

```
Section stream_eq.
  Variable A : Set.

  CoInductive stream_eq : stream A → stream A → Prop :=
  | Stream_eq : ∀ h t1 t2,
    stream_eq t1 t2
    → stream_eq (Cons h t1) (Cons h t2).
End stream_eq.
```

We say that two streams are equal if and only if they have the same heads and their tails are equal. We use the normal finite-syntactic equality for the heads, and we refer to our new equality recursively for the tails.

We can try restating the theorem with **stream_eq**.

```
Theorem ones_eq : stream_eq ones ones'.
```

Coq does not support tactical co-inductive proofs as well as it supports tactical inductive proofs. The usual starting point is the *cofix* tactic, which asks to structure this proof as a co-fixpoint.

```
  cofix.

  ones_eq : stream_eq ones ones'
  ============================
```

stream_eq ones ones'

It looks like this proof might be easier than we expected!

```
assumption.
```

```
Proof completed.
```

Unfortunately, we are due for some disappointment in our victory lap.

```
Qed.
```

```
Error:
Recursive definition of ones_eq is ill-formed.

In environment
ones_eq : stream_eq ones ones'

unguarded recursive call in "ones_eq"
```

Via the Curry-Howard correspondence, the same guardedness condition applies to our co-inductive proofs as to our co-inductive data structures. We should be grateful that this proof is rejected, because, if it were not, the same proof structure could be used to prove any co-inductive theorem vacuously, by direct appeal to itself!

Thinking about how Coq would generate a proof term from the proof script above, we see that the problem is that we are violating the first part of the guardedness condition. During our proofs, Coq can help us check whether we have yet gone wrong in this way. We can run the command *Guarded* in any context to see if it is possible to finish the proof in a way that will yield a properly guarded proof term.

```
Guarded.
```

Running *Guarded* here gives us the same error message that we got when we tried to run `Qed`. In larger proofs, *Guarded* can be helpful in detecting problems *before* we think we are ready to run `Qed`.

We need to start the co-induction by applying one of **stream_eq**'s constructors. To do that, we need to know that both arguments to the predicate are Conses. Informally, this is trivial, but `simpl` is not able to help us.

```
Undo.
simpl.

ones_eq : stream_eq ones ones'
============================
```

stream_eq ones ones'

It turns out that we are best served by proving an auxiliary lemma.

```
Abort.
```

First, we need to define a function that seems pointless on first glance.

```
Definition frob A (s : stream A) : stream A :=
  match s with
    | Cons h t ⇒ Cons h t
  end.
```

Next, we need to prove a theorem that seems equally pointless.

```
Theorem frob_eq : ∀ A (s : stream A), s = frob s.
  destruct s; reflexivity.
Qed.
```

But, miraculously, this theorem turns out to be just what we needed.

```
Theorem ones_eq : stream_eq ones ones'.
  cofix.
```

We can use the theorem to rewrite the two streams.

```
  rewrite (frob_eq ones).
  rewrite (frob_eq ones').
```

```
  ones_eq : stream_eq ones ones'
  ============================
   stream_eq (frob ones) (frob ones')
```

Now `simpl` is able to reduce the streams.

```
  simpl.
```

```
  ones_eq : stream_eq ones ones'
  ============================
   stream_eq (Cons 1 ones)
     (Cons 1
        ((cofix map (s : stream nat) : stream nat :=
            match s with
            | Cons h t ⇒ Cons (S h) (map t)
            end) zeroes))
```

Since we have exposed the Cons structure of each stream, we can apply the constructor of **stream_eq**.

```
  constructor.
```

```
  ones_eq : stream_eq ones ones'
  ============================
   stream_eq ones
     ((cofix map (s : stream nat) : stream nat :=
         match s with
         | Cons h t ⇒ Cons (S h) (map t)
         end) zeroes)
```

Now, modulo unfolding of the definition of `map`, we have matched our assumption.

```
  assumption.
Qed.
```

Why did this silly-looking trick help? The answer has to do with the constraints placed on Coq's evaluation rules by the need for termination. The *cofix*-related restriction that foiled our first attempt at using `simpl` is dual to a restriction for *fix*. In particular, an application of an anonymous *fix* only reduces when the top-level structure of the recursive argument is known. Otherwise, we would be unfolding the recursive definition ad infinitum.

Fixpoints only reduce when enough is known about the *definitions* of their arguments. Dually, co-fixpoints only reduce when enough is known about *how their results will be used*. In particular, a *cofix* is only expanded when it is the discriminee of a `match`. Rewriting with our superficially silly lemma wrapped new `match`es around the two *cofix*es, triggering reduction.

If *cofix*es reduced haphazardly, it would be easy to run into infinite loops in evaluation, since we are, after all, building infinite objects.

One common source of difficulty with co-inductive proofs is bad interaction with standard Coq automation machinery. If we try to prove `ones_eq'` with automation, like we have in previous inductive proofs, we get an invalid proof.

```
Theorem ones_eq' : stream_eq ones ones'.
  cofix; crush.

  Guarded.
Abort.
```

The standard `auto` machinery sees that our goal matches an assumption and so applies that assumption, even though this violates guardedness. One usually starts a proof like this by `destruct`ing some parameter and running a custom tactic to figure out the first proof rule to apply for each case. Alternatively, there are tricks that can be played with "hiding" the co-inductive hypothesis.

5.3 Simple Modeling of Non-Terminating Programs

We close the chapter with a quick motivating example for more complex uses of co-inductive types. We will define a co-inductive semantics for a simple assembly language and use that semantics to prove that assembly programs always run forever. This basic technique can be combined with typing judgments for more advanced languages, where some ill-typed programs can go wrong, but no well-typed programs go wrong.

We define suggestive synonyms for **nat**, for cases where we use natural numbers as registers or program labels. That is, we consider our idealized machine to have infinitely many registers and infinitely many code addresses.

```
Definition reg := nat.
Definition label := nat.
```

Our instructions are loading of a constant into a register, copying from one register to another, unconditional jump, and conditional jump based on whether the value in a register is not zero.

```
Inductive instr : Set :=
| Imm : reg → nat → instr
| Copy : reg → reg → instr
| Jmp : label → instr
| Jnz : reg → label → instr.
```

We define a type regs of maps from registers to values. To define a function `set` for setting a register's value in a map, we import the *Arith* module from Coq's standard library, and we use its function eq_nat_dec for comparing natural numbers.

```
Definition regs := reg → nat.
Require Import Arith.
Definition set (rs : regs) (r : reg) (v : nat) : regs :=
  fun r' ⇒ if eq_nat_dec r r' then v else rs r'.
```

An inductive **exec** judgment captures the effect of an instruction on the program counter and register bank.

```
Inductive exec : label → regs → instr → label → regs → Prop :=
| E_Imm : ∀ pc rs r n, exec pc rs (Imm r n) (S pc) (set rs r n)
| E_Copy : ∀ pc rs r1 r2, exec pc rs (Copy r1 r2) (S pc) (set rs r1 (rs r2))
| E_Jmp : ∀ pc rs pc', exec pc rs (Jmp pc') pc' rs
| E_JnzF : ∀ pc rs r pc', rs r = 0 → exec pc rs (Jnz r pc') (S pc) rs
| E_JnzT : ∀ pc rs r pc' n, rs r = S n → exec pc rs (Jnz r pc') pc' rs.
```

We prove that **exec** represents a total function. In our proof script, we use a `match` tactic with a *context* pattern. This particular example finds an occurrence of a pattern Jnz ?r _ anywhere in the current subgoal's conclusion. We use a Coq library tactic *case_eq* to do case analysis on whether the current value *rs r* of the register *r* is zero or not. *case_eq* differs from `destruct` in saving an equality relating the old variable to the new form we deduce for

it.

```
Lemma exec_total : ∀ pc rs i,
  ∃ pc', ∃ rs', exec pc rs i pc' rs'.
  Hint Constructors exec.

  destruct i; crush; eauto;
    match goal with
      | [ ⊢ context[Jnz ?r _] ] ⇒ case_eq (rs r)
    end; eauto.
Qed.
```

We are ready to define a co-inductive judgment capturing the idea that a program runs forever. We define the judgment in terms of a program **prog**, represented as a function mapping each label to the instruction found there.

```
Section safe.
  Variable prog : label → instr.

  CoInductive safe : label → regs → Prop :=
  | Step : ∀ pc r pc' r',
    exec pc r (prog pc) pc' r'
    → safe pc' r'
    → safe pc r.
```

Now we can prove that any starting address and register bank lead to safe infinite execution. Recall that proofs of existentially-quantified formulas are all built with a single constructor of the inductive type **ex**. This means that we can use `destruct` to "open up" such proofs. In the proof below, we want to perform this opening up on an appropriate use of the exec_total lemma. This lemma's conclusion begins with two existential quantifiers, so we want to tell `destruct` that it should not stop at the first quantifier. We accomplish our goal by using an *intro pattern* with `destruct`. Consult the Coq manual for the details of intro patterns; the specific pattern [? [? ?]] that we use here accomplishes our goal of destructing both quantifiers at once.

```
  Theorem always_safe : ∀ pc rs,
    safe pc rs.
    cofix; intros;
      destruct (exec_total pc rs (prog pc)) as [? [? ?]];
        econstructor; eauto.
  Qed.
End safe.
```

If we print the proof term that was generated, we can verify that the proof is structured as a *cofix*, with each co-recursive call properly guarded.

```
Print always_safe.
```

5.4 Exercises

1. (a) Define a co-inductive type of infinite trees carrying data of a fixed parameter type. Each node should contain a data value and two child trees.

   (b) Define a function *everywhere* for building a tree with the same data value at every node.

   (c) Define a function map for building an output tree out of two input trees by traversing them in parallel and applying a two-argument function to their corresponding data values.

   (d) Define a tree falses where every node has the value false.

   (e) Define a tree *true_false* where the root node has value true, its children have value false, all nodes at the next have the value true, and so on, alternating boolean values from level to level.

   (f) Prove that *true_false* is equal to the result of mapping the boolean "or" function *orb* over *true_false* and falses. You can make *orb* available with `Require Import Bool.`. You may find the lemma *orb_false_r* from the same module helpful. Your proof here should not be about the standard equality =, but rather about some new equality relation that you define.

Part II

Programming with Dependent Types

Chapter 6

Subset Types and Variations

So far, we have seen many examples of what we might call "classical program verification." We write programs, write their specifications, and then prove that the programs satisfy their specifications. The programs that we have written in Coq have been normal functional programs that we could just as well have written in Haskell or ML. In this chapter, we start investigating uses of *dependent types* to integrate programming, specification, and proving into a single phase.

6.1 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*.

One aspects in particular of the definition of pred_strong1 that 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 →

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: program components of type `Prop` are erased systematically.

We can reimplement our dependently-typed pred based on *subset types*, defined in the standard library with the type family **sig**.

`Print` *sig.*

`Inductive` **sig** $(A : \texttt{Type})\ (P : A \to \texttt{Prop}) : \texttt{Type} :=$
  exist : $\forall\ x : A,\ P\ x \to$ **sig** P
For **sig**: *Argument A is implicit*
For exist: *Argument A is implicit*

sig is a Curry-Howard twin of **ex**, except that **sig** is in `Type`, while **ex** 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.
```

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.
```

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. 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. 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.

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.
```

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 you 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.

6.2 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.

```
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); congruence.
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. We need to assume the existence of an equality decision procedure for the type of list elements.

```
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')
```

6.3 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.
```

6.4 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.
```

6.5 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, in the style introduced in Chapter 4.

```
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 type-checker 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} + {∀ 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, only eauto will apply this hint.

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

```
  refine (fix F (e : exp) : {t : type | hasType e t} + {∀ 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)}
```

6.6 Exercises

All of the notations defined in this chapter, plus some extras, are available for import from the module MoreSpecif of the book source.

1. Write a function of type $\forall\ n\ m$: **nat**, $\{n \leq m\} + \{n > m\}$. That is, this function decides whether one natural is less than another, and its dependent type guarantees that its results are accurate.

2. (a) Define *var*, a type of propositional variables, as a synonym for **nat**.

   (b) Define an inductive type **prop** of propositional logic formulas, consisting of variables, negation, and binary conjunction and disjunction.

   (c) Define a function propDenote from variable truth assignments and **prop**s to `Prop`, based on the usual meanings of the connectives. Represent truth assignments as functions from *var* to **bool**.

   (d) Define a function *bool_true_dec* that checks whether a boolean is true, with a maximally expressive dependent type. That is, the function should have type $\forall$ b, $\{b = \mathsf{true}\} + \{b = \mathsf{true} \rightarrow \mathbf{False}\}$.

   (e) Define a function *decide* that determines whether a particular **prop** is true under a particular truth assignment. That is, the function should have type $\forall$ (*truth* : *var* $\rightarrow$ **bool**) (p : **prop**), {propDenote *truth* p} + {$\neg$ propDenote *truth* p}. This function is probably easiest to write in the usual tactical style, instead of programming with `refine`. *bool_true_dec* may come in handy as a hint.

   (f) Define a function *negate* that returns a simplified version of the negation of a **prop**. That is, the function should have type $\forall\ p$: **prop**, {*p'* : **prop** | $\forall$ *truth*, propDenote *truth* $p \leftrightarrow \neg$ propDenote *truth p'*}. To simplify a variable, just negate it. Simplify a negation by returning its argument. Simplify conjunctions and disjunctions using De Morgan's laws, negating the arguments recursively and switching the kind of connective. *decide* may be useful in some of the proof obligations, even if you do not use it in the computational part of *negate*'s definition. Lemmas like *decide* allow us to compensate for the lack of a general Law of the Excluded Middle in CIC.

3. Implement the DPLL satisfiability decision procedure for boolean formulas in conjunctive normal form, with a dependent type that guarantees its correctness. An example of a reasonable type for this function would be $\forall$ f : **formula**, {*truth* : *tvals* | *formulaTrue truth* f} + {$\forall$ *truth*, $\neg$ *formulaTrue truth* f}. Implement at least "the basic backtracking algorithm" as defined here:

   http://en.wikipedia.org/wiki/DPLL_algorithm

   It might also be instructive to implement the unit propagation and pure literal elimination optimizations described there or some other optimizations that have been used in modern SAT solvers.

Chapter 7

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, 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 proof assistants that we surveyed in Chapter 1. This chapter 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.

7.1 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 chapter. We need to combine a `return` clause with a new kind of annotation, an `in` clause. 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.
```

Using `return` alone allowed us to express a dependency of the `match` result type on the *value* of the discriminee. What `in` adds to our arsenal is a way of expressing a dependency on the *type* of the discriminee. Specifically, the *n1* in the `in` clause above is a *binding occurrence* whose scope is the `return` clause.

We may use `in` clauses only to bind names for the arguments of an inductive type family. That is, each `in` clause must be an inductive type family name applied to a sequence of underscores and variable names of the proper length. The positions for *parameters* to the type family must all be underscores. Parameters are those arguments declared with section variables or with entries to the left of the first colon in an inductive definition. They cannot

vary depending on which constructor was used to build the discriminee, so Coq prohibits pointless matches on them. It is those arguments defined in the type to the right of the colon that we may name with `in` clauses.

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 ⇒ ???
    | 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 in this and the next few chapters 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.
```

7.2 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 ineffiency 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. The only new thing we see is the %type annotation, which tells Coq to parse the `match` expression using the notations associated with types. Without this annotation, the × would be interpreted as multiplication on naturals, rather than as the product type constructor. **type** is one example of an identifer bound to a *notation scope*. We will deal more explicitly with notations and notation scopes in later chapters.

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 our old favorite, 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 future chapter, 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.

This is the only new trick we need to learn to complete the proof. We can back up and give a short, automated proof. The main inconvenience in the proof is that we cannot write a pattern that matches a `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.
```

7.3 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 chapter. 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 arbitary 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 satisifies 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 ver-

bose. 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.
```

7.4 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.

```
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 book source includes statements, proofs, and hint commands for a handful of such omittted theorems. Since they are orthogonal to our use of dependent types, we hide them in the rendered versions of this book.

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 first 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 $n = \text{length}\ s$.

```
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 book 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. It has a straightforward implementation. We introduce a spurious match on s so that `simpl` will know to reduce calls to dec_star. The heuristic that `simpl` uses is only to unfold identifier definitions when doing so would simplify some `match` expression.

```
Definition dec_star : {star P s} + {¬ star P s}.
  refine (match s return _ with
            | "" ⇒ Reduce (dec_star' (n := length s) 0 _)
            | _ ⇒ Reduce (dec_star' (n := length s) 0 _)
          end); 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.
```

7.5 Exercises

1. Define a kind of dependently-typed lists, where a list's type index gives a lower bound on how many of its elements satisfy a particular predicate. In particular, for an arbitrary set A and a predicate P over it:

   (a) Define a type *plist* : **nat** → Set. Each *plist* *n* should be a list of As, where it is guaranteed that at least *n* distinct elements satisfy P. There is wide latitude in choosing how to encode this. You should try to avoid using subset types or any other mechanism based on annotating non-dependent types with propositions after-the-fact.

   (b) Define a version of list concatenation that works on *plist*s. The type of this new function should express as much information as possible about the output *plist*.

   (c) Define a function *plistOut* for translating *plist*s to normal **list**s.

   (d) Define a function *plistIn* for translating **list**s to *plist*s. The type of *plistIn* should make it clear that the best bound on P-matching elements is chosen. You may assume that you are given a dependently-typed function for deciding instances of P.

   (e) Prove that, for any list *ls*, *plistOut* (*plistIn ls*) = *ls*. This should be the only part of the exercise where you use tactic-based proving.

   (f) Define a function *grab* : ∀ *n* (*ls* : *plist* (*S n*)), **sig** P. That is, when given a *plist* guaranteed to contain at least one element satisfying P, *grab* produces such

an element. **sig** is the type family of sigma types, and **sig** P is extensionally equivalent to $\{x : A \mid P\ x\}$, though the latter form uses an eta-expansion of P instead of P itself as the predicate.

Chapter 8

Dependent Data Structures

Our red-black tree example from the last chapter 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 chapter exploring it.

8.1 More Length-Indexed Lists

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

```
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 $\{m : \mathbf{nat} \mid m < n\}$. The type family names 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 ⇒ ?
    | 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 chapter. 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
```

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.
```

8.2 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).
```

8.2.1 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, which we will discuss in more detail in later chapters. 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 Sets.

```
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. In a later chapter, we will meet other, more common approaches to language formalization. Such approaches 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.

8.3 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 $n = S\ n'$, 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].
```

8.4 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. This is the same problem that we met in Chapter 3 with other nested inductive types.

```
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
```

The automatically-generated induction principle is too weak. For the Node case, it gives us no inductive hypothesis. We could write our own induction principle, as we did in Chapter 3, 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"
```

The special-case rule for nested datatypes 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 yet another of the inductive definition techniques introduced in Chapter 3, reflexive types. 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.

8.4.1 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 instantation 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 : _ → _ ⊢ _ ] ⇒
        generalize (IHn (fun idx ⇒ tests (Some idx)) (fun idx ⇒ bodies (Some idx)));
          clear IHn; intro IHn
    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. We treat this issue with our discussion of axioms in a later chapter. For now, 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.
```

8.5 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.

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 with the techniques we studied in Chapter 3. 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.

8.6 Exercises

Some of the type family definitions and associated functions from this chapter are duplicated in the DepList module of the book source. Some of their names have been changed to be more sensible in a general context.

1. Define a tree analogue of **hlist**. That is, define a parameterized type of binary trees with data at their leaves, and define a type family *htree* indexed by trees. The structure of an *htree* mirrors its index tree, with the type of each data element (which only occur at leaves) determined by applying a type function to the corresponding element of the index tree. Define a type standing for all possible paths from the root of a tree to leaves and use it to implement a function *tget* for extracting an element of an *htree* by path. Define a function *htmap2* for "mapping over two trees in parallel." That is,

htmap2 takes in two *htree*s with the same index tree, and it forms a new *htree* with the same index by applying a binary function pointwise.

Repeat this process so that you implement each definition for each of the three definition styles covered in this chapter: inductive, recursive, and index function.

2. Write a dependently-typed interpreter for a simple programming language with ML-style pattern-matching, using one of the encodings of heterogeneous lists to represent the different branches of a `case` expression. (There are other ways to represent the same thing, but the point of this exercise is to practice using those heterogeneous list types.) The object language is defined informally by this grammar:

   $t ::= \mathsf{bool} \mid t + t$
   $p ::= x \mid b \mid \mathsf{inl}\ p \mid \mathsf{inr}\ p$
   $\mathsf{e} ::= x \mid b \mid \mathsf{inl}\ \mathsf{e} \mid \mathsf{inr}\ \mathsf{e} \mid \mathtt{case}\ \mathsf{e}\ \mathit{of}\ [p \Rightarrow \mathsf{e}]^* \mid \_ \Rightarrow \mathsf{e}$

   x stands for a variable, and b stands for a boolean constant. The production for `case` expressions means that a pattern-match includes zero or more pairs of patterns and expressions, along with a default case.
   Your interpreter should be implemented in the style demonstrated in this chapter. That is, your definition of expressions should use dependent types and de Bruijn indices to combine syntax and typing rules, such that the type of an expression tells the types of variables that are in scope. You should implement a simple recursive function translating types t to `Set`, and your interpreter should produce values in the image of this translation.

Chapter 9

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 chapter, we will focus on design patterns for circumventing these tricky issues, and we will introduce the different notions of equality as they are germane.

9.1 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 subsituted.

```
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 chapter, we will see examples where alternate definitions may simplify proofs. Before that point, we will introduce effective proof methods for goals that use proofs of the standard propositional equality "as data."

9.2 Heterogeneous Lists Revisited

One of our example dependent data structures from the last chapter 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 chapter.

```
  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 `match`es 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 chapter, 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
     : forall (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 : forall (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 : forall (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 $x = x'$, refering 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.
```

9.3 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. 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 $(ls1 \mathbin{+\!\!+} ls2) \mathbin{+\!\!+} ls3$, for some values of the list variables. If we rewrite the type of the proof used to type-cast this value to something like $ls1 \mathbin{+\!\!+} ls2 \mathbin{+\!\!+} ls3 = ls1 \mathbin{+\!\!+} ls2 \mathbin{+\!\!+} ls3$, 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].
```

9.4 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 of 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.

```
  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'.
```

9.5 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 section, we demonstrate this by showing how each the previous two sections' 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 section. 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 intercovert 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 chapter, 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.

9.6 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.

```
  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.
```

9.7 Exercises

1. Implement and prove correct a substitution function for simply-typed lambda calculus. In particular:

   (a) Define a datatype **type** of lambda types, including just booleans and function types.

   (b) Define a type family exp : **list type** → **type** → Type of lambda expressions, including boolean constants, variables, and function application and abstraction.

   (c) Implement a definitional interpreter for exps, by way of a recursive function over expressions and substitutions for free variables, like in the related example from the last chapter.

   (d) Implement a function subst : ∀ *t' ts t*, exp (*t'* :: *ts*) *t* → exp *ts t'* → exp *ts t*. The type of the first expression indicates that its most recently bound free variable has type *t'*. The second expression also has type *t'*, and the job of subst is to substitute the second expression for every occurrence of the "first" variable of the first expression.

   (e) Prove that subst preserves program meanings. That is, prove
   ∀ *t' ts t* (e : exp (*t'* :: *ts*) *t*) (*e'* : exp *ts t'*) (*s* : **hlist** typeDenote *ts*),
   expDenote (subst e *e'*) *s* = expDenote e (expDenote *e' s* ::: *s*)

   where ::: is an infix operator for heterogeneous "cons" that is defined in the book's DepList module.

The material presented up to this point should be sufficient to enable a good solution of this exercise, with enough ingenuity. If you get stuck, it may be helpful to use the following structure. None of these elements need to appear in your solution, but we can at least guarantee that there is a reasonable solution based on them.

(a) The DepList module will be useful. You can get the standard dependent list definitions there, instead of copying-and-pasting from the last chapter. It is worth reading the source for that module over, since it defines some new helpful functions and notations that we did not use last chapter.

(b) Define a recursive function liftVar : $\forall$ *ts1 ts2 t t'*, **member** *t* (*ts1* ++ *ts2*) $\rightarrow$ **member** *t* (*ts1* ++ *t'* :: *ts2*). This function should "lift" a de Bruijn variable so that its type refers to a new variable inserted somewhere in the index list.

(c) Define a recursive function *lift'* : $\forall$ *ts t* (e : exp *ts t*) *ts1 ts2 t'*, *ts* = *ts1* ++ *ts2* $\rightarrow$ exp (*ts1* ++ *t'* :: *ts2*) *t* which performs a similar lifting on an exp. The convoluted type is to get around restrictions on `match` annotations. We delay "realizing" that the first index of e is built with list concatenation until after a dependent `match`, and the new explicit proof argument must be used to cast some terms that come up in the `match` body.

(d) Define a function lift : $\forall$ *ts t t'*, exp *ts t* $\rightarrow$ exp (*t'* :: *ts*) *t*, which handles simpler top-level lifts. This should be an easy one-liner based on *lift'*.

(e) Define a recursive function substVar : $\forall$ *ts1 ts2 t t'*, **member** *t* (*ts1* ++ *t'* :: *ts2*) $\rightarrow$ (*t'* = *t*) + **member** *t* (*ts1* ++ *ts2*). This function is the workhorse behind substitution applied to a variable. It returns inl to indicate that the variable we pass to it is the variable that we are substituting for, and it returns inr to indicate that the variable we are examining is *not* the one we are substituting for. In the first case, we get a proof that the necessary typing relationship holds, and, in the second case, we get the original variable modified to reflect the removal of the substitutee from the typing context.

(f) Define a recursive function *subst'* : $\forall$ *ts t* (e : exp *ts t*) *ts1 t' ts2*, *ts* = *ts1* ++ *t'* :: *ts2* $\rightarrow$ exp (*ts1* ++ *ts2*) *t'* $\rightarrow$ exp (*ts1* ++ *ts2*) *t*. This is the workhorse of substitution in expressions, employing the same proof-passing trick as for *lift'*. You will probably want to use lift somewhere in the definition of *subst'*.

(g) Now `subst` should be a one-liner, defined in terms of *subst'*.

(h) Prove a correctness theorem for each auxiliary function, leading up to the proof of `subst` correctness.

(i) All of the reasoning about equality proofs in these theorems follows a regular pattern. If you have an equality proof that you want to replace with refl_equal somehow, run `generalize` on that proof variable. Your goal is to get to the point where you can `rewrite` with the original proof to change the type of the generalized version. To avoid type errors (the infamous "second-order unification"

failure messages), it will be helpful to run `generalize` on other pieces of the proof context that mention the equality's lefthand side. You might also want to use `generalize dependent`, which generalizes not just one variable but also all variables whose types depend on it. `generalize dependent` has the sometimes-helpful property of removing from the context all variables that it generalizes. Once you do manage the mind-bending trick of using the equality proof to rewrite its own type, you will be able to rewrite with UIP_refl.

(j) A variant of the *ext_eq* axiom from the end of this chapter is available in the book module Axioms, and you will probably want to use it in the *lift'* and *subst'* correctness proofs.

(k) The *change* tactic should come in handy in the proofs about lift and `subst`, where you want to introduce "extraneous" list concatenations with nil to match the forms of earlier theorems.

(l) Be careful about `destruct`ing a term "too early." You can use `generalize` on proof terms to bring into the proof context any important propositions about the term. Then, when you `destruct` the term, it is updated in the extra propositions, too. The *case_eq* tactic is another alternative to this approach, based on saving an equality between the original term and its new form.

Chapter 10

Generic Programming

Generic programming makes it possible to write functions that operate over different types of data. Parametric polymorphism in ML and Haskell is one of the simplest examples. ML-style module systems and Haskell type classes are more flexible cases. These language features are often not as powerful so we would like. For instance, while Haskell includes a type class classifying those types whose values can be pretty-printed, per-type pretty-printing is usually either implemented manually or implemented via a *deriving* clause, which triggers ad-hoc code generation. Some clever encoding tricks have been used to achieve better within Haskell and other languages, but we can do datatype-generic programming much more cleanly with dependent types. Thanks to the expressive power of CIC, we need no special language support.

Generic programming can often be very useful in Coq developments, so we devote this chapter to studying it. In a proof assistant, there is the new possibility of generic proofs about generic programs, which we also devote some space to.

10.1 Reflecting Datatype Definitions

The key to generic programming with dependent types is *universe types*. This concept should not be confused with the idea of *universes* from the metatheory of CIC and related languages. Rather, the idea of universe types is to define inductive types that provide *syntactic representations* of Coq types. We cannot directly write CIC programs that do case analysis on types, but we *can* case analyze on reflected syntactic versions of those types.

Thus, to begin, we must define a syntactic representation of some class of datatypes. In this chapter, our running example will have to do with basic algebraic datatypes, of the kind found in ML and Haskell, but without additional bells and whistles like type parameters and mutually-recursive definitions.

The first step is to define a representation for constructors of our datatypes.

```
Record constructor : Type := Con {
  nonrecursive : Type;
```

```
  recursive : nat
}.
```

The idea is that a constructor represented as Con T *n* has *n* arguments of the type that we are defining. Additionally, all of the other, non-recursive arguments can be encoded in the type T. When there are no non-recursive arguments, T can be **unit**. When there are two non-recursive arguments, of types A and B, T can be $A * B$. We can generalizer to any number of arguments via tupling.

With this definition, it as easy to define a datatype representation in terms of lists of constructors.

```
Definition datatype := list constructor.
```

Here are a few example encodings for some common types from the Coq standard library. While our syntax type does not support type parameters directly, we can implement them at the meta level, via functions from types to datatypes.

```
Definition Empty_set_dt : datatype := nil.
Definition unit_dt : datatype := Con unit 0 :: nil.
Definition bool_dt : datatype := Con unit 0 :: Con unit 0 :: nil.
Definition nat_dt : datatype := Con unit 0 :: Con unit 1 :: nil.
Definition list_dt (A : Type) : datatype := Con unit 0 :: Con A 1 :: nil.
```

Empty_set has no constructors, so its representation is the empty list. **unit** has one constructor with no arguments, so its one reflected constructor indicates no non-recursive data and 0 recursive arguments. The representation for **bool** just duplicates this single argumentless constructor. We get from **bool** to **nat** by changing one of the constructors to indicate 1 recursive argument. We get from **nat** to **list** by adding a non-recursive argument of a parameter type A.

As a further example, we can do the same encoding for a generic binary tree type.

```
Section tree.
  Variable A : Type.

  Inductive tree : Type :=
  | Leaf : A → tree
  | Node : tree → tree → tree.
End tree.

Definition tree_dt (A : Type) : datatype := Con A 0 :: Con unit 2 :: nil.
```

Each datatype representation stands for a family of inductive types. For a specific real datatype and a reputed representation for it, it is useful to define a type of *evidence* that the datatype is compatible with the encoding.

```
Section denote.
  Variable T : Type.
```

This variable stands for the concrete datatype that we are interested in.

```
  Definition constructorDenote (c : constructor) :=
```

nonrecursive $c \to$ **ilist** T (recursive c) $\to T$.

We write that a constructor is represented as a function returning a T. Such a function takes two arguments, which pack together the non-recursive and recursive arguments of the constructor. We represent a tuple of all recursive arguments using the length-indexed list type **ilist** that we met in Chapter 7.

```
Definition datatypeDenote := hlist constructorDenote.
```

Finally, the evidence for type T is a hetergeneous list, including a constructor denotation for every constructor encoding in a datatype encoding. Recall that, since we are inside a section binding T as a variable, constructorDenote is automatically parameterized by T.

```
End denote.
```

Some example pieces of evidence should help clarify the convention. First, we define some helpful notations, providing different ways of writing constructor denotations. There is really just one notation, but we need several versions of it to cover different choices of which variables will be used in the body of a definition. The ASCII ~> from the notation will be rendered later as $\rightsquigarrow$.

```
Notation "[ ! , ! ~> x ]" := ((fun _ _ ⇒ x) : constructorDenote _ (Con _ _)).
Notation "[ v , ! ~> x ]" := ((fun v _ ⇒ x) : constructorDenote _ (Con _ _)).
Notation "[ ! , r ~> x ]" := ((fun _ r ⇒ x) : constructorDenote _ (Con _ _)).
Notation "[ v , r ~> x ]" := ((fun v r ⇒ x) : constructorDenote _ (Con _ _)).

Definition Empty_set_den : datatypeDenote Empty_set Empty_set_dt :=
  HNil.
Definition unit_den : datatypeDenote unit unit_dt :=
  [!, ! ⇝ tt] ::: HNil.
Definition bool_den : datatypeDenote bool bool_dt :=
  [!, ! ⇝ true] ::: [!, ! ⇝ false] ::: HNil.
Definition nat_den : datatypeDenote nat nat_dt :=
  [!, ! ⇝ O] ::: [!, r ⇝ S (hd r)] ::: HNil.
Definition list_den (A : Type) : datatypeDenote (list A) (list_dt A) :=
  [!, ! ⇝ nil] ::: [x, r ⇝ x :: hd r] ::: HNil.
Definition tree_den (A : Type) : datatypeDenote (tree A) (tree_dt A) :=
  [v, ! ⇝ Leaf v] ::: [!, r ⇝ Node (hd r) (hd (tl r))] ::: HNil.
```

10.2 Recursive Definitions

We built these encodings of datatypes to help us write datatype-generic recursive functions. To do so, we will want a reflected representation of a *recursion scheme* for each type, similar to the *T_rect* principle generated automatically for an inductive definition of T. A clever reuse of datatypeDenote yields a short definition.

```
Definition fixDenote (T : Type) (dt : datatype) :=
```

$\forall$ (R : Type), datatypeDenote R dt $\rightarrow$ (T $\rightarrow$ R).

The idea of a recursion scheme is parameterized by a type and a reputed encoding of it. The principle itself is polymorphic in a type R, which is the return type of the recursive function that we mean to write. The next argument is a hetergeneous list of one case of the recursive function definition for each datatype constructor. The datatypeDenote function turns out to have just the right definition to express the type we need; a set of function cases is just like an alternate set of constructors where we replace the original type T with the function result type R. Given such a reflected definition, a fixDenote invocation returns a function from T to R, which is just what we wanted.

We are ready to write some example functions now. It will be useful to use one new function from the DepList library included in the book source.

```
Check hmake.
```

```
hmake
     : ∀ (A : Type) (B : A → Type),
       (∀ x : A, B x) → ∀ ls : list A, hlist B l
```

hmake is a kind of map alternative that goes from a regular **list** to an **hlist**. We can use it to define a generic size function which counts the number of constructors used to build a value in a datatype.

```
Definition size T dt (fx : fixDenote T dt) : T → nat :=
  fx nat (hmake (B := constructorDenote nat) (fun _ _ r ⇒ foldr plus 1 r) dt).
```

Our definition is parameterized over a recursion scheme fx. We instantiate fx by passing it the function result type and a set of function cases, where we build the latter with hmake. The function argument to hmake takes three arguments: the representation of a constructor, its non-recursive arguments, and the results of recursive calls on all of its recursive arguments. We only need the recursive call results here, so we call them r and bind the other two inputs with wildcards. The actual case body is simple: we add together the recursive call results and increment the result by one (to account for the current constructor). This foldr function is an **hlist**-specific version defined in the DepList module.

It is instructive to build fixDenote values for our example types and see what specialized size functions result from them.

```
Definition Empty_set_fix : fixDenote Empty_set Empty_set_dt :=
  fun R _ emp ⇒ match emp with end.
Eval compute in size Empty_set_fix.
```

```
     = fun emp : Empty_set ⇒ match emp return nat with
                             end
     : Empty_set → nat
```

Despite all the fanciness of the generic size function, CIC's standard computation rules suffice to normalize the generic function specialization to exactly what we would have written manually.

```
Definition unit_fix : fixDenote unit unit_dt :=
  fun R cases _ ⇒ (hhd cases) tt INil.
Eval compute in size unit_fix.
```

```
     = fun _ : unit ⇒ 1
     : unit → nat
```

Again normalization gives us the natural function definition. We see this pattern repeated for our other example types.

```
Definition bool_fix : fixDenote bool bool_dt :=
  fun R cases b ⇒ if b
    then (hhd cases) tt INil
    else (hhd (htl cases)) tt INil.
Eval compute in size bool_fix.
```

```
     = fun b : bool ⇒ if b then 1 else 1
     : bool → nat
```

```
Definition nat_fix : fixDenote nat nat_dt :=
  fun R cases ⇒ fix F (n : nat) : R :=
    match n with
      | O ⇒ (hhd cases) tt INil
      | S n' ⇒ (hhd (htl cases)) tt (ICons (F n') INil)
    end.
```

To peek at the size function for **nat**, it is useful to avoid full computation, so that the recursive definition of addition is not expanded inline. We can accomplish this with proper flags for the *cbv* reduction strategy.

```
Eval cbv beta iota delta -[plus] in size nat_fix.
```

```
     = fix F (n : nat) : nat := match n with
                                | 0 ⇒ 1
                                | S n' ⇒ F n' + 1
                                end
     : nat → nat
```

```
Definition list_fix (A : Type) : fixDenote (list A) (list_dt A) :=
  fun R cases ⇒ fix F (ls : list A) : R :=
    match ls with
      | nil ⇒ (hhd cases) tt INil
      | x :: ls' ⇒ (hhd (htl cases)) x (ICons (F ls') INil)
    end.
Eval cbv beta iota delta -[plus] in fun A ⇒ size (@list_fix A).
```

```
     = fun A : Type ⇒
       fix F (ls : list A) : nat :=
         match ls with
```

```
      | nil ⇒ 1
      | _ :: ls' ⇒ F ls' + 1
      end
  : ∀ A : Type, list A → nat

Definition tree_fix (A : Type) : fixDenote (tree A) (tree_dt A) :=
  fun R cases ⇒ fix F (t : tree A) : R :=
    match t with
      | Leaf x ⇒ (hhd cases) x INil
      | Node t1 t2 ⇒ (hhd (htl cases)) tt (ICons (F t1) (ICons (F t2) INil))
    end.
Eval cbv beta iota delta -[plus] in fun A ⇒ size (@tree_fix A).

  = fun A : Type ⇒
    fix F (t : tree A) : nat :=
      match t with
      | Leaf _ ⇒ 1
      | Node t1 t2 ⇒ F t1 + (F t2 + 1)
      end
  : ∀ A : Type, tree A → n
```

10.2.1 Pretty-Printing

It is also useful to do generic pretty-printing of datatype values, rendering them as human-readable strings. To do so, we will need a bit of metadata for each constructor. Specifically, we need the name to print for the constructor and the function to use to render its non-recursive arguments. Everything else can be done generically.

```
Record print_constructor (c : constructor) : Type := PI {
  printName : string;
  printNonrec : nonrecursive c → string
}.
```

It is useful to define a shorthand for applying the constructor PI. By applying it explicitly to an unknown application of the constructor Con, we help type inference work.

```
Notation "^" := (PI (Con _ _)).
```

As in earlier examples, we define the type of metadata for a datatype to be a heterogeneous list type collecting metadata for each constructor.

```
Definition print_datatype := hlist print_constructor.
```

We will be doing some string manipulation here, so we import the notations associated with strings.

```
Local Open Scope string_scope.
```

Now it is easy to implement our generic printer, using another function from DepList.

```
Check hmap.
```

```
hmap
     : ∀ (A : Type) (B1 B2 : A → Type),
       (∀ x : A, B1 x → B2 x) →
       ∀ ls : list A, hlist B1 ls → hlist B2 ls
```

```
Definition print T dt (pr : print_datatype dt) (fx : fixDenote T dt) : T → string :=
  fx string (hmap (B1 := print_constructor) (B2 := constructorDenote string)
    (fun _ pc x r ⇒ printName pc ++ "(" ++ printNonrec pc x
      ++ foldr (fun s acc ⇒ ", " ++ s ++ acc) ")" r) pr).
```

Some simple tests establish that print gets the job done.

```
Eval compute in print HNil Empty_set_fix.
```

```
     = fun emp : Empty_set ⇒ match emp return string with
                             end
     : Empty_set → string
```

```
Eval compute in print (^ "tt" (fun _ ⇒ "") ::: HNil) unit_fix.
```

```
     = fun _ : unit ⇒ "tt()"
     : unit → string
```

```
Eval compute in print (^ "true" (fun _ ⇒ "")
  ::: ^ "false" (fun _ ⇒ "")
  ::: HNil) bool_fix.
```

```
   = fun b : bool ⇒ if b then "true()" else "false()"
   : bool → s
```

```
Definition print_nat := print (^ "O" (fun _ ⇒ "")
  ::: ^ "S" (fun _ ⇒ "")
  ::: HNil) nat_fix.
Eval cbv beta iota delta -[append] in print_nat.
```

```
     = fix F (n : nat) : string :=
         match n with
         | 0%nat ⇒ "O" ++ "(" ++ "" ++ ")"
         | S n' ⇒ "S" ++ "(" ++ "" ++ ", " ++ F n' ++ ")"
         end
     : nat → string
```

```
Eval simpl in print_nat 0.
```

```
     = "O()"
     : string
```

```
Eval simpl in print_nat 1.
```

```
     = "S(, O())"
     : string
```

```
Eval simpl in print_nat 2.
     = "S(, S(, O()))"
     : string

Eval cbv beta iota delta -[append] in fun A (pr : A → string) ⇒
  print (^ "nil" (fun _ ⇒ ""))
  ::: ^ "cons" pr
  ::: HNil) (@list_fix A).
     = fun (A : Type) (pr : A → string) ⇒
       fix F (ls : list A) : string :=
         match ls with
         | nil ⇒ "nil" ++ "(" ++ "" ++ ")"
         | x :: ls' ⇒ "cons" ++ "(" ++ pr x ++ ", " ++ F ls' ++ ")"
         end
     : ∀ A : Type, (A → string) → list A → string

Eval cbv beta iota delta -[append] in fun A (pr : A → string) ⇒
  print (^ "Leaf" pr
  ::: ^ "Node" (fun _ ⇒ ""))
  ::: HNil) (@tree_fix A).
     = fun (A : Type) (pr : A → string) ⇒
       fix F (t : tree A) : string :=
         match t with
         | Leaf x ⇒ "Leaf" ++ "(" ++ pr x ++ ")"
         | Node t1 t2 ⇒
             "Node" ++ "(" ++ "" ++ ", " ++ F t1 ++ ", " ++ F t2 ++ ")"
         end
     : ∀ A : Type, (A → string) → tree A → string
```

10.2.2 Mapping

By this point, we have developed enough machinery that it is old hat to define a generic function similar to the list map function.

```
Definition map T dt (dd : datatypeDenote T dt) (fx : fixDenote T dt) (f : T → T)
  : T → T :=
  fx T (hmap (B1 := constructorDenote T) (B2 := constructorDenote T)
    (fun _ c x r ⇒ f (c x r)) dd).

Eval compute in map Empty_set_den Empty_set_fix.
     = fun (_ : Empty_set → Empty_set) (emp : Empty_set) ⇒
       match emp return Empty_set with
       end
     : (Empty_set → Empty_set) → Empty_set → Empty_set
```

```
Eval compute in map unit_den unit_fix.
     = fun (f : unit → unit) (_ : unit) ⇒ f tt
     : (unit → unit) → unit → unit
Eval compute in map bool_den bool_fix.
     = fun (f : bool → bool) (b : bool) ⇒ if b then f true else f false
     : (bool → bool) → bool → bool
Eval compute in map nat_den nat_fix.
     = fun f : nat → nat ⇒
       fix F (n : nat) : nat :=
         match n with
         | 0%nat ⇒ f 0%nat
         | S n' ⇒ f (S (F n'))
         end
     : (nat → nat) → nat → nat
Eval compute in fun A ⇒ map (list_den A) (@list_fix A).
     = fun (A : Type) (f : list A → list A) ⇒
       fix F (ls : list A) : list A :=
         match ls with
         | nil ⇒ f nil
         | x :: ls' ⇒ f (x :: F ls')
         end
     : ∀ A : Type, (list A → list A) → list A → list A
Eval compute in fun A ⇒ map (tree_den A) (@tree_fix A).
     = fun (A : Type) (f : tree A → tree A) ⇒
       fix F (t : tree A) : tree A :=
         match t with
         | Leaf x ⇒ f (Leaf x)
         | Node t1 t2 ⇒ f (Node (F t1) (F t2))
         end
     : ∀ A : Type, (tree A → tree A) → tree A → tree A
Definition map_nat := map nat_den nat_fix.
Eval simpl in map_nat S 0.
     = 1%nat
     : nat
Eval simpl in map_nat S 1.
     = 3%nat
     : nat
Eval simpl in map_nat S 2.
     = 5%nat
```

```
    : nat
```

10.3 Proving Theorems about Recursive Definitions

We would like to be able to prove theorems about our generic functions. To do so, we need to establish additional well-formedness properties that must hold of pieces of evidence.

```
Section ok.
  Variable T : Type.
  Variable dt : datatype.

  Variable dd : datatypeDenote T dt.
  Variable fx : fixDenote T dt.
```

First, we characterize when a piece of evidence about a datatype is acceptable. The basic idea is that the type T should really be an inductive type with the definition given by dd. Semantically, inductive types are characterized by the ability to do induction on them. Therefore, we require that the usual induction principle is true, with respect to the constructors given in the encoding dd.

```
  Definition datatypeDenoteOk :=
    ∀ P : T → Prop,
      (∀ c (m : member c dt) (x : nonrecursive c) (r : ilist T (recursive c)),
        (∀ i : fin (recursive c), P (get r i))
        → P ((hget dd m) x r))
      → ∀ v, P v.
```

This definition can take a while to digest. The quantifier over m : **member** c dt is considering each constructor in turn; like in normal induction principles, each constructor has an associated proof case. The expression hget dd m then names the constructor we have selected. After binding m, we quantify over all possible arguments (encoded with x and r) to the constructor that m selects. Within each specific case, we quantify further over i : **fin** (recursive c) to consider all of our induction hypotheses, one for each recursive argument of the current constructor.

We have completed half the burden of defining side conditions. The other half comes in characterizing when a recursion scheme fx is valid. The natural condition is that fx behaves appropriately when applied to any constructor application.

```
  Definition fixDenoteOk :=
    ∀ (R : Type) (cases : datatypeDenote R dt)
      c (m : member c dt)
      (x : nonrecursive c) (r : ilist T (recursive c)),
      fx cases ((hget dd m) x r)
      = (hget cases m) x (imap (fx cases) r).
```

As for datatypeDenoteOk, we consider all constructors and all possible arguments to them by quantifying over m, x, and r. The lefthand side of the equality that follows shows a call to

the recursive function on the specific constructor application that we selected. The righthand side shows an application of the function case associated with constructor m, applied to the non-recursive arguments and to appropriate recursive calls on the recursive arguments.

```
End ok.
```

We are now ready to prove that the size function we defined earlier always returns positive results. First, we establish a simple lemma.

```
Lemma foldr_plus : ∀ n (ils : ilist nat n),
  foldr plus 1 ils > 0.
  induction ils; crush.
Qed.

Theorem size_positive : ∀ T dt
  (dd : datatypeDenote T dt) (fx : fixDenote T dt)
  (dok : datatypeDenoteOk dd) (fok : fixDenoteOk dd fx)
  (v : T),
  size fx v > 0.
  unfold size; intros.

  ============================
   fx nat
     (hmake
        (fun (x : constructor) (_ : nonrecursive x)
           (r : ilist nat (recursive x)) ⇒ foldr plus 1%nat r) dt) v > 0
```

Our goal is an inequality over a particular call to size, with its definition expanded. How can we proceed here? We cannot use `induction` directly, because there is no way for Coq to know that T is an inductive type. Instead, we need to use the induction principle encoded in our hypothesis *dok* of type datatypeDenoteOk *dd*. Let us try applying it directly.

```
  apply dok.

Error: Impossible to unify "datatypeDenoteOk dd" with
 "fx nat
    (hmake
       (fun (x : constructor) (_: nonrecursive x)
          (r : ilist nat (recursive x)) ⇒ foldr plus 1%nat r) dt) v > 0".
```

Matching the type of *dok* with the type of our conclusion requires more than simple first-order unification, so `apply` is not up to the challenge. We can use the `pattern` tactic to get our goal into a form that makes it apparent exactly what the induction hypothesis is.

```
  pattern v.
```

```
============================
 (fun t : T ⇒
  fx nat
    (hmake
       (fun (x : constructor) (_ : nonrecursive x)
          (r : ilist nat (recursive x)) ⇒ foldr plus 1%nat r) dt) t > 0) v

apply dok; crush.

H : ∀ i : fin (recursive c),
    fx nat
      (hmake
         (fun (x : constructor) (_ : nonrecursive x)
            (r : ilist nat (recursive x)) ⇒ foldr plus 1%nat r) dt)
      (get r i) > 0
============================
 hget
   (hmake
      (fun (x0 : constructor) (_ : nonrecursive x0)
         (r0 : ilist nat (recursive x0)) ⇒ foldr plus 1%nat r0) dt) m x
   (imap
      (fx nat
         (hmake
            (fun (x0 : constructor) (_ : nonrecursive x0)
               (r0 : ilist nat (recursive x0)) ⇒
             foldr plus 1%nat r0) dt)) r) > 0
```

An induction hypothesis H is generated, but we turn out not to need it for this example. We can simplify the goal using a library theorem about the composition of hget and hmake.

```
rewrite hget_hmake.

============================
 foldr plus 1%nat
   (imap
      (fx nat
         (hmake
            (fun (x0 : constructor) (_ : nonrecursive x0)
               (r0 : ilist nat (recursive x0)) ⇒
             foldr plus 1%nat r0) dt)) r) > 0
```

The lemma we proved earlier finishes the proof.

```
apply foldr_plus.
```

Using hints, we can redo this proof in a nice automated form.

```
  Restart.

  Hint Rewrite hget_hmake : cpdt.
  Hint Resolve foldr_plus.

  unfold size; intros; pattern v; apply dok; crush.
Qed.
```

It turned out that, in this example, we only needed to use induction degenerately as case analysis. A more involved theorem may only be proved using induction hypotheses. We will give its proof only in unautomated form and leave effective automation as an exercise for the motivated reader.

In particular, it ought to be the case that generic `map` applied to an identity function is itself an identity function.

```
Theorem map_id : ∀ T dt
  (dd : datatypeDenote T dt) (fx : fixDenote T dt)
  (dok : datatypeDenoteOk dd) (fok : fixDenoteOk dd fx)
  (v : T),
  map dd fx (fun x ⇒ x) v = v.
```

Let us begin as we did in the last theorem, after adding another useful library equality as a hint.

```
  Hint Rewrite hget_hmap : cpdt.

  unfold map; intros; pattern v; apply dok; crush.

  H : ∀ i : fin (recursive c),
      fx T
        (hmap
           (fun (x : constructor) (c : constructorDenote T x)
              (x0 : nonrecursive x) (r : ilist T (recursive x)) ⇒
            c x0 r) dd) (get r i) = get r i
  ============================
   hget dd m x
     (imap
        (fx T
           (hmap
              (fun (x0 : constructor) (c0 : constructorDenote T x0)
                 (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
               c0 x1 r0) dd)) r) = hget dd m x r
```

Our goal is an equality whose two sides begin with the same function call and initial arguments. We believe that the remaining arguments are in fact equal as well, and the `f_equal` tactic applies this reasoning step for us formally.

```
  f_equal.
```

```
============================
 imap
   (fx T
      (hmap
         (fun (x0 : constructor) (c0 : constructorDenote T x0)
            (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) =>
          c0 x1 r0) dd)) r = r
```

At this point, it is helpful to proceed by an inner induction on the heterogeneous list r of recursive call results. We could arrive at a cleaner proof by breaking this step out into an explicit lemma, but here we will do the induction inline to save space.

`induction` r; *crush*.

The base case is discharged automatically, and the inductive case looks like this, where H is the outer IH (for induction over T values) and *IHn* is the inner IH (for induction over the recursive arguments).

```
H : ∀ i : fin (S n),
    fx T
      (hmap
         (fun (x : constructor) (c : constructorDenote T x)
            (x0 : nonrecursive x) (r : ilist T (recursive x)) =>
          c x0 r) dd)
      (match i in (fin n') return ((fin (pred n') → T) → T) with
       | First n ⇒ fun _ : fin n → T ⇒ a
       | Next n idx' ⇒ fun get_ls' : fin n → T ⇒ get_ls' idx'
       end (get r)) =
    match i in (fin n') return ((fin (pred n') → T) → T) with
    | First n ⇒ fun _ : fin n → T ⇒ a
    | Next n idx' ⇒ fun get_ls' : fin n → T ⇒ get_ls' idx'
    end (get r)
IHr : (∀ i : fin n,
       fx T
         (hmap
            (fun (x : constructor) (c : constructorDenote T x)
               (x0 : nonrecursive x) (r : ilist T (recursive x)) =>
             c x0 r) dd) (get r i) = get r i) →
      imap
        (fx T
           (hmap
              (fun (x : constructor) (c : constructorDenote T x)
                 (x0 : nonrecursive x) (r : ilist T (recursive x)) =>
```

```
                    c x0 r) dd)) r = r
============================
 ICons
   (fx T
      (hmap
         (fun (x0 : constructor) (c0 : constructorDenote T x0)
            (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
          c0 x1 r0) dd) a)
   (imap
      (fx T
         (hmap
            (fun (x0 : constructor) (c0 : constructorDenote T x0)
               (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
             c0 x1 r0) dd)) r) = ICons a r
```

We see another opportunity to apply `f_equal`, this time to split our goal into two different equalities over corresponding arguments. After that, the form of the first goal matches our outer induction hypothesis *H*, when we give type inference some help by specifying the right quantifier instantiation.

```
f_equal.
apply (H First).

============================
 imap
   (fx T
      (hmap
         (fun (x0 : constructor) (c0 : constructorDenote T x0)
            (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
          c0 x1 r0) dd)) r = r
```

Now the goal matches the inner IH *IHr*.

```
apply IHr; crush.

i : fin n
============================
 fx T
   (hmap
      (fun (x0 : constructor) (c0 : constructorDenote T x0)
         (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
       c0 x1 r0) dd) (get r i) = get r i
```

We can finish the proof by applying the outer IH again, specialized to a different **fin**

value.

```
  apply (H (Next i)).
Qed.
```

Chapter 11

Universes and Axioms

Many traditional theorems can be proved in Coq without special knowledge of CIC, the logic behind the prover. A development just seems to be using a particular ASCII notation for standard formulas based on set theory. Nonetheless, as we saw in Chapter 4, CIC differs from set theory in starting from fewer orthogonal primitives. It is possible to define the usual logical connectives as derived notions. The foundation of it all is a dependently-typed functional programming language, based on dependent function types and inductive type families. By using the facilities of this language directly, we can accomplish some things much more easily than in mainstream math.

Gallina, which adds features to the more theoretical CIC, is the logic implemented in Coq. It has a relatively simple foundation that can be defined rigorously in a page or two of formal proof rules. Still, there are some important subtleties that have practical ramifications. This chapter focuses on those subtleties, avoiding formal metatheory in favor of example code.

11.1 The `Type` Hierarchy

Every object in Gallina has a type.

```
Check 0.
```

```
  0
     : nat
```

It is natural enough that zero be considered as a natural number.

```
Check nat.
```

```
  nat
     : Set
```

From a set theory perspective, it is unsurprising to consider the natural numbers as a "set."

```
Check Set.
```

```
Set
   : Type
```

The type `Set` may be considered as the set of all sets, a concept that set theory handles in terms of *classes*. In Coq, this more general notion is `Type`.

```
Check Type.
```

```
Type
   : Type
```

Strangely enough, `Type` appears to be its own type. It is known that polymorphic languages with this property are inconsistent. That is, using such a language to encode proofs is unwise, because it is possible to "prove" any proposition. What is really going on here?

Let us repeat some of our queries after toggling a flag related to Coq's printing behavior.

```
Set Printing Universes.
Check nat.
```

```
nat
   : Set
```

```
Check Set.
```

```
Set
   : Type (* (0)+1 *)
```

```
Check Type.
```

```
Type (* Top.3 *)
   : Type (* (Top.3)+1 *)
```

Occurrences of `Type` are annotated with some additional information, inside comments. These annotations have to do with the secret behind `Type`: it really stands for an infinite hierarchy of types. The type of `Set` is `Type`(0), the type of `Type`(0) is `Type`(1), the type of `Type`(1) is `Type`(2), and so on. This is how we avoid the "`Type : Type`" paradox. As a convenience, the universe hierarchy drives Coq's one variety of subtyping. Any term whose type is `Type` at level i is automatically also described by `Type` at level j when $j > \mathsf{i}$.

In the outputs of our first `Check` query, we see that the type level of `Set`'s type is (0)+1. Here 0 stands for the level of `Set`, and we increment it to arrive at the level that *classifies* `Set`.

In the second query's output, we see that the occurrence of `Type` that we check is assigned a fresh *universe variable* *Top*.3. The output type increments *Top*.3 to move up a level in the universe hierarchy. As we write code that uses definitions whose types mention universe variables, unification may refine the values of those variables. Luckily, the user rarely has to worry about the details.

Another crucial concept in CIC is *predicativity*. Consider these queries.

```
Check ∀ T : nat, fin T.
  ∀ T : nat, fin T
     : Set
Check ∀ T : Set, T.
  ∀ T : Set, T
     : Type (* max(0, (0)+1) *)
Check ∀ T : Type, T.
  ∀ T : Type (* Top.9 *) , T
     : Type (* max(Top.9, (Top.9)+1) *)
```

These outputs demonstrate the rule for determining which universe a ∀ type lives in. In particular, for a type ∀ x : *T1*, *T2*, we take the maximum of the universes of *T1* and *T2*. In the first example query, both *T1* (**nat**) and *T2* (**fin** T) are in `Set`, so the ∀ type is in `Set`, too. In the second query, *T1* is `Set`, which is at level (0)+1; and *T2* is T, which is at level 0. Thus, the ∀ exists at the maximum of these two levels. The third example illustrates the same outcome, where we replace `Set` with an occurrence of `Type` that is assigned universe variable *Top*.9. This universe variable appears in the places where 0 appeared in the previous query.

The behind-the-scenes manipulation of universe variables gives us predicativity. Consider this simple definition of a polymorphic identity function.

```
Definition id (T : Set) (x : T) : T := x.
Check id 0.
  id 0
     : nat

Check id Set.

Error: Illegal application (Type Error):
...
The 1st term has type "Type (* (Top.15)+1 *)" which should be coercible to "Set".
```

The parameter T of id must be instantiated with a `Set`. **nat** is a `Set`, but `Set` is not. We can try fixing the problem by generalizing our definition of id.

```
Reset id.
Definition id (T : Type) (x : T) : T := x.
Check id 0.
  id 0
     : nat
Check id Set.
```

```
  id Set
    : Type (* Top.17 *)
Check id Type.
  id Type (* Top.18 *)
    : Type (* Top.19 *)
```

So far so good. As we apply id to different T values, the inferred index for T's Type occurrence automatically moves higher up the type hierarchy.

```
Check id id.
```

Error: *Universe inconsistency* (*cannot enforce* *Top*.16 < *Top*.16).

This error message reminds us that the universe variable for T still exists, even though it is usually hidden. To apply id to itself, that variable would need to be less than itself in the type hierarchy. Universe inconsistency error messages announce cases like this one where a term could only type-check by violating an implied constraint over universe variables. Such errors demonstrate that Type is *predicative*, where this word has a CIC meaning closely related to its usual mathematical meaning. A predicative system enforces the constraint that, for any object of quantified type, none of those quantifiers may ever be instantiated with the object itself. Impredicativity is associated with popular paradoxes in set theory, involving inconsistent constructions like "the set of all sets that do not contain themselves." Similar paradoxes result from uncontrolled impredicativity in Coq.

11.1.1 Inductive Definitions

Predicativity restrictions also apply to inductive definitions. As an example, let us consider a type of expression trees that allows injection of any native Coq value. The idea is that an exp T stands for a reflected expression of type T.

```
Inductive exp : Set → Set :=
| Const : ∀ T : Set, T → exp T
| Pair : ∀ T1 T2, exp T1 → exp T2 → exp (T1 * T2)
| Eq : ∀ T, exp T → exp T → exp bool.
```

Error: Large *non*-propositional *inductive types must be* in Type.

This definition is *large* in the sense that at least one of its constructors takes an argument whose type has type Type. Coq would be inconsistent if we allowed definitions like this one in their full generality. Instead, we must change exp to live in Type. We will go even further and move exp's index to Type as well.

```
Inductive exp : Type → Type :=
```

```
| Const : ∀ T, T → exp T
| Pair : ∀ T1 T2, exp T1 → exp T2 → exp (T1 * T2)
| Eq : ∀ T, exp T → exp T → exp bool.
```

Note that before we had to include an annotation : `Set` for the variable T in Const's type, but we need no annotation now. When the type of a variable is not known, and when that variable is used in a context where only types are allowed, Coq infers that the variable is of type `Type`. That is the right behavior here, but it was wrong for the `Set` version of exp.

Our new definition is accepted. We can build some sample expressions.

```
Check Const 0.
```

```
  Const 0
     : exp nat
```

```
Check Pair (Const 0) (Const tt).
```

```
  Pair (Const 0) (Const tt)
     : exp (nat * unit)
```

```
Check Eq (Const Set) (Const Type).
```

```
  Eq (Const Set) (Const Type (* Top.59 *) )
     : exp bool
```

We can check many expressions, including fancy expressions that include types. However, it is not hard to hit a type-checking wall.

```
Check Const (Const O).
```

Error: *Universe inconsistency* (*cannot enforce* $Top.42 < Top.42$).

We are unable to instantiate the parameter T of Const with an exp type. To see why, it is helpful to print the annotated version of exp's inductive definition.

```
Print exp.
```

```
Inductive exp
             : Type (* Top.8 *) →
               Type
               (* max(0, (Top.11)+1, (Top.14)+1, (Top.15)+1, (Top.19)+1) *) :=
  Const : ∀ T : Type (* Top.11 *) , T → exp T
| Pair : ∀ (T1 : Type (* Top.14 *) ) (T2 : Type (* Top.15 *) ),
         exp T1 → exp T2 → exp (T1 * T2)
| Eq : ∀ T : Type (* Top.19 *) , exp T → exp T → exp bool
```

We see that the index type of exp has been assigned to universe level $Top.8$. In addition, each of the four occurrences of `Type` in the types of the constructors gets its own universe variable. Each of these variables appears explicitly in the type of exp. In particular, any

type exp T lives at a universe level found by incrementing by one the maximum of the four argument variables. A consequence of this is that exp *must* live at a higher universe level than any type which may be passed to one of its constructors. This consequence led to the universe inconsistency.

Strangely, the universe variable *Top*.8 only appears in one place. Is there no restriction imposed on which types are valid arguments to exp? In fact, there is a restriction, but it only appears in a global set of universe constraints that are maintained "off to the side," not appearing explicitly in types. We can print the current database.

```
Print Universes.
```

$Top.19 < Top.9 \leq Top.8$
$Top.15 < Top.9 \leq Top.8 \leq Coq.Init.Datatypes.38$
$Top.14 < Top.9 \leq Top.8 \leq Coq.Init.Datatypes.37$
$Top.11 < Top.9 \leq Top.8$

`Print` *Universes* outputs many more constraints, but we have collected only those that mention *Top* variables. We see one constraint for each universe variable associated with a constructor argument from exp's definition. *Top*.19 is the type argument to Eq. The constraint for *Top*.19 effectively says that *Top*.19 must be less than *Top*.8, the universe of exp's indices; an intermediate variable *Top*.9 appears as an artifact of the way the constraint was generated.

The next constraint, for *Top*.15, is more complicated. This is the universe of the second argument to the Pair constructor. Not only must *Top*.15 be less than *Top*.8, but it also comes out that *Top*.8 must be less than *Coq.Init.Datatypes*.38. What is this new universe variable? It is from the definition of the *prod* inductive family, to which types of the form $A * B$ are desugared.

```
Print prod.
```

```
Inductive prod (A : Type (* Coq.Init.Datatypes.37 *) )
          (B : Type (* Coq.Init.Datatypes.38 *) )
            : Type (* max(Coq.Init.Datatypes.37, Coq.Init.Datatypes.38) *) :=
    pair : A → B → A * B
```

We see that the constraint is enforcing that indices to exp must not live in a higher universe level than B-indices to *prod*. The next constraint above establishes a symmetric condition for A.

Thus it is apparent that Coq maintains a tortuous set of universe variable inequalities behind the scenes. It may look like some functions are polymorphic in the universe levels of their arguments, but what is really happening is imperative updating of a system of constraints, such that all uses of a function are consistent with a global set of universe levels. When the constraint system may not be evolved soundly, we get a universe inconsistency error.

Something interesting is revealed in the annotated definition of *prod*. A type *prod A*

B lives at a universe that is the maximum of the universes of A and B. From our earlier experiments, we might expect that *prod*'s universe would in fact need to be *one higher* than the maximum. The critical difference is that, in the definition of *prod*, A and B are defined as *parameters*; that is, they appear named to the left of the main colon, rather than appearing (possibly unnamed) to the right.

Parameters are not as flexible as normal inductive type arguments. The range types of all of the constructors of a parameterized type must share the same parameters. Nonetheless, when it is possible to define a polymorphic type in this way, we gain the ability to use the new type family in more ways, without triggering universe inconsistencies. For instance, nested pairs of types are perfectly legal.

```
Check (nat, (Type, Set)).
```

```
  (nat, (Type (* Top.44 *) , Set))
     : Set * (Type (* Top.45 *) * Type (* Top.46 *) )
```

The same cannot be done with a counterpart to *prod* that does not use parameters.

```
Inductive prod' : Type → Type → Type :=
| pair' : ∀ A B : Type, A → B → prod' A B.

Check (pair' nat (pair' Type Set)).
```

Error: *Universe inconsistency* (*cannot enforce* *Top*.51 < *Top*.51).

The key benefit parameters bring us is the ability to avoid quantifying over types in the types of constructors. Such quantification induces less-than constraints, while parameters only introduce less-than-or-equal-to constraints.

Coq includes one more (potentially confusing) feature related to parameters. While Gallina does not support real universe polymorphism, there is a convenience facility that mimics universe polymorphism in some cases. We can illustrate what this means with a simple example.

```
Inductive foo (A : Type) : Type :=
| Foo : A → foo A.

Check foo nat.
```

```
  foo nat
     : Set
```

```
Check foo Set.
```

```
  foo Set
     : Type
```

```
Check foo True.
```

```
  foo True
```

```
    : Prop
```

The basic pattern here is that Coq is willing to automatically build a "copied-and-pasted" version of an inductive definition, where some occurrences of `Type` have been replaced by `Set` or `Prop`. In each context, the type-checker tries to find the valid replacements that are lowest in the type hierarchy. Automatic cloning of definitions can be much more convenient than manual cloning. We have already taken advantage of the fact that we may re-use the same families of tuple and list types to form values in `Set` and `Type`.

Imitation polymorphism can be confusing in some contexts. For instance, it is what is responsible for this weird behavior.

```
Inductive bar : Type := Bar : bar.

Check bar.
  bar
    : Prop
```

The type that Coq comes up with may be used in strictly more contexts than the type one might have expected.

11.2 The `Prop` Universe

In Chapter 4, we saw parallel versions of useful datatypes for "programs" and "proofs." The convention was that programs live in `Set`, and proofs live in `Prop`. We gave little explanation for why it is useful to maintain this distinction. There is certainly documentation value from separating programs from proofs; in practice, different concerns apply to building the two types of objects. It turns out, however, that these concerns motivate formal differences between the two universes in Coq.

Recall the types **sig** and **ex**, which are the program and proof versions of existential quantification. Their definitions differ only in one place, where **sig** uses `Type` and **ex** uses `Prop`.

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

Print ex.
  Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P
```

It is natural to want a function to extract the first components of data structures like these. Doing so is easy enough for **sig**.

```
Definition projS A (P : A → Prop) (x : sig P) : A :=
  match x with
```

```
    | exist v _ ⇒ v
  end.
```

We run into trouble with a version that has been changed to work with **ex**.

```
Definition projE A (P : A → Prop) (x : ex P) : A :=
  match x with
    | ex_intro v _ ⇒ v
  end.
```

```
Error:
Incorrect elimination of "x" in the inductive type "ex":
the return type has sort "Type" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Type
because proofs can be eliminated only to build proofs.
```

In formal Coq parlance, "elimination" means "pattern-matching." The typing rules of Gallina forbid us from pattern-matching on a discriminee whose type belongs to `Prop`, whenever the result type of the `match` has a type besides `Prop`. This is a sort of "information flow" policy, where the type system ensures that the details of proofs can never have any effect on parts of a development that are not also marked as proofs.

This restriction matches informal practice. We think of programs and proofs as clearly separated, and, outside of constructive logic, the idea of computing with proofs is ill-formed. The distinction also has practical importance in Coq, where it affects the behavior of extraction.

Recall that extraction is Coq's facility for translating Coq developments into programs in general-purpose programming languages like OCaml. Extraction *erases* proofs and leaves programs intact. A simple example with **sig** and **ex** demonstrates the distinction.

```
Definition sym_sig (x : sig (fun n ⇒ n = 0)) : sig (fun n ⇒ 0 = n) :=
  match x with
    | exist n pf ⇒ exist _ n (sym_eq pf)
  end.
```

Extraction sym_sig.

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

let sym_sig x = x
```

Since extraction erases proofs, the second components of **sig** values are elided, making **sig** a simple identity type family. The sym_sig operation is thus an identity function.

```
Definition sym_ex (x : ex (fun n ⇒ n = 0)) : ex (fun n ⇒ 0 = n) :=
  match x with
```

```
    | ex_intro n pf ⇒ ex_intro _ n (sym_eq pf)
  end.
```

Extraction sym_ex.

```
(** val sym_ex : __ **)

let sym_ex = __
```

In this example, the **ex** type itself is in `Prop`, so whole **ex** packages are erased. Coq extracts every proposition as the type `__`, whose single constructor is `__`. Not only are proofs replaced by `__`, but proof arguments to functions are also removed completely, as we see here.

Extraction is very helpful as an optimization over programs that contain proofs. In languages like Haskell, advanced features make it possible to program with proofs, as a way of convincing the type checker to accept particular definitions. Unfortunately, when proofs are encoded as values in GADTs, these proofs exist at runtime and consume resources. In contrast, with Coq, as long as you keep all of your proofs within `Prop`, extraction is guaranteed to erase them.

Many fans of the Curry-Howard correspondence support the idea of *extracting programs from proofs*. In reality, few users of Coq and related tools do any such thing. Instead, extraction is better thought of as an optimization that reduces the runtime costs of expressive typing.

We have seen two of the differences between proofs and programs: proofs are subject to an elimination restriction and are elided by extraction. The remaining difference is that `Prop` is *impredicative*, as this example shows.

```
Check ∀ P Q : Prop, P ∨ Q → Q ∨ P.
  ∀ P Q : Prop, P ∨ Q → Q ∨ P
    : Prop
```

We see that it is possible to define a `Prop` that quantifies over other `Prop`s. This is fortunate, as we start wanting that ability even for such basic purposes as stating propositional tautologies. In the next section of this chapter, we will see some reasons why unrestricted impredicativity is undesirable. The impredicativity of `Prop` interacts crucially with the elimination restriction to avoid those pitfalls.

Impredicativity also allows us to implement a version of our earlier `exp` type that does not suffer from the weakness that we found.

```
Inductive expP : Type → Prop :=
| ConstP : ∀ T, T → expP T
| PairP : ∀ T1 T2, expP T1 → expP T2 → expP (T1 * T2)
| EqP : ∀ T, expP T → expP T → expP bool.

Check ConstP 0.
```

```
  ConstP 0
     : expP nat
Check PairP (ConstP 0) (ConstP tt).
  PairP (ConstP 0) (ConstP tt)
     : expP (nat * unit)
Check EqP (ConstP Set) (ConstP Type).
  EqP (ConstP Set) (ConstP Type)
     : expP bool
Check ConstP (ConstP O).
  ConstP (ConstP 0)
     : expP (expP nat)
```

In this case, our victory is really a shallow one. As we have marked **expP** as a family of proofs, we cannot deconstruct our expressions in the usual programmatic ways, which makes them almost useless for the usual purposes. Impredicative quantification is much more useful in defining inductive families that we really think of as judgments. For instance, this code defines a notion of equality that is strictly stronger than the base equality $=$.

```
Inductive eqPlus : ∀ T, T → T → Prop :=
| Base : ∀ T (x : T), eqPlus x x
| Func : ∀ dom ran (f1 f2 : dom → ran),
  (∀ x : dom, eqPlus (f1 x) (f2 x))
  → eqPlus f1 f2.
Check (Base 0).
  Base 0
     : eqPlus 0 0
Check (Func (fun n ⇒ n) (fun n ⇒ 0 + n) (fun n ⇒ Base n)).
  Func (fun n : nat ⇒ n) (fun n : nat ⇒ 0 + n) (fun n : nat ⇒ Base n)
     : eqPlus (fun n : nat ⇒ n) (fun n : nat ⇒ 0 + n)
Check (Base (Base 1)).
  Base (Base 1)
     : eqPlus (Base 1) (Base 1)
```

11.3 Axioms

While the specific logic Gallina is hardcoded into Coq's implementation, it is possible to add certain logical rules in a controlled way. In other words, Coq may be used to reason about many different refinements of Gallina where strictly more theorems are provable. We achieve this by asserting *axioms* without proof.

We will motivate the idea by touring through some standard axioms, as enumerated in Coq's online FAQ. I will add additional commentary as appropriate.

11.3.1 The Basics

One simple example of a useful axiom is the law of the excluded middle.

```
Require Import Classical_Prop.
Print classic.
```

```
*** [ classic : ∀ P : Prop, P ∨ ¬ P ]
```

In the implementation of module *Classical_Prop*, this axiom was defined with the command

```
Axiom classic : ∀ P : Prop, P ∨ ¬ P.
```

An `Axiom` may be declared with any type, in any of the universes. There is a synonym `Parameter` for `Axiom`, and that synonym is often clearer for assertions not of type `Prop`. For instance, we can assert the existence of objects with certain properties.

```
Parameter n : nat.
Axiom positive : n > 0.
Reset n.
```

This kind of "axiomatic presentation" of a theory is very common outside of higher-order logic. However, in Coq, it is almost always preferable to stick to defining your objects, functions, and predicates via inductive definitions and functional programming.

In general, there is a significant burden associated with any use of axioms. It is easy to assert a set of axioms that together is *inconsistent*. That is, a set of axioms may imply **False**, which allows any theorem to proved, which defeats the purpose of a proof assistant. For example, we could assert the following axiom, which is consistent by itself but inconsistent when combined with *classic*.

```
Axiom not_classic : ∃ P : Prop, ¬ (P ∨ ¬ P).

Theorem uhoh : False.
  generalize classic not_classic; firstorder.
Qed.

Theorem uhoh_again : 1 + 1 = 3.
  destruct uhoh.
Qed.

Reset not_classic.
```

On the subject of the law of the excluded middle itself, this axiom is usually quite harmless, and many practical Coq developments assume it. It has been proved metatheoretically to be consistent with CIC. Here, "proved metatheoretically" means that someone proved on paper that excluded middle holds in a *model* of CIC in set theory. All of the other axioms that we will survey in this section hold in the same model, so they are all consistent together.

Recall that Coq implements *constructive* logic by default, where excluded middle is not provable. Proofs in constructive logic can be thought of as programs. A $\forall$ quantifier denotes a dependent function type, and a disjunction denotes a variant type. In such a setting, excluded middle could be interpreted as a decision procedure for arbitrary propositions, which computability theory tells us cannot exist. Thus, constructive logic with excluded middle can no longer be associated with our usual notion of programming.

Given all this, why is it all right to assert excluded middle as an axiom? The intuitive justification is that the elimination restriction for `Prop` prevents us from treating proofs as programs. An excluded middle axiom that quantified over `Set` instead of `Prop` *would* be problematic. If a development used that axiom, we would not be able to extract the code to OCaml (soundly) without implementing a genuine universal decision procedure. In contrast, values whose types belong to `Prop` are always erased by extraction, so we sidestep the axiom's algorithmic consequences.

Because the proper use of axioms is so precarious, there are helpful commands for determining which axioms a theorem relies on.

```
Theorem t1 : ∀ P : Prop, P → ¬ ¬ P.
  tauto.
Qed.

Print Assumptions t1.
```

```
Closed under the global context
```

```
Theorem t2 : ∀ P : Prop, ¬ ¬ P → P.

  tauto.

Error: tauto failed.

  intro P; destruct (classic P); tauto.
Qed.

Print Assumptions t2.
```

```
Axioms:
classic : ∀ P : Prop, P ∨ ¬ P
```

It is possible to avoid this dependence in some specific cases, where excluded middle *is* provable, for decidable families of propositions.

```
Theorem classic_nat_eq : ∀ n m : nat, n = m ∨ n ≠ m.
  induction n; destruct m; intuition; generalize (IHn m); intuition.
Qed.

Theorem t2' : ∀ n m : nat, ¬ ¬ (n = m) → n = m.
  intros n m; destruct (classic_nat_eq n m); tauto.
Qed.
```

```
Print Assumptions t2'.
```
Closed *under the global context*

Mainstream mathematical practice assumes excluded middle, so it can be useful to have it available in Coq developments, though it is also nice to know that a theorem is proved in a simpler formal system than classical logic. There is a similar story for *proof irrelevance*, which simplifies proof issues that would not even arise in mainstream math.

```
Require Import ProofIrrelevance.
Print proof_irrelevance.
```
*** [*proof_irrelevance* : $\forall$ (P : Prop) ($p1$ $p2$: P), $p1 = p2$]

This axiom asserts that any two proofs of the same proposition are equal. If we replaced $p1 = p2$ by $p1 \leftrightarrow p2$, then the statement would be provable. However, equality is a stronger notion than logical equivalence. Recall this example function from Chapter 6.

```
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 might want to prove that different proofs of $n > 0$ do not lead to different results from our richly-typed predecessor function.

```
Theorem pred_strong1_irrel : ∀ n (pf1 pf2 : n > 0), pred_strong1 pf1 = pred_strong1 pf2.
  destruct n; crush.
Qed.
```

The proof script is simple, but it involved peeking into the definition of pred_strong1. For more complicated function definitions, it can be considerably more work to prove that they do not discriminate on details of proof arguments. This can seem like a shame, since the Prop elimination restriction makes it impossible to write any function that does otherwise. Unfortunately, this fact is only true metatheoretically, unless we assert an axiom like *proof_irrelevance*. With that axiom, we can prove our theorem without consulting the definition of pred_strong1.

```
Theorem pred_strong1_irrel' : ∀ n (pf1 pf2 : n > 0), pred_strong1 pf1 = pred_strong1 pf2.
  intros; f_equal; apply proof_irrelevance.
Qed.
```

In the chapter on equality, we already discussed some axioms that are related to proof irrelevance. In particular, Coq's standard library includes this axiom:

```
Require Import Eqdep.
Import Eq_rect_eq.
Print eq_rect_eq.
```

```
*** [ eq_rect_eq :
∀ (U : Type) (p : U) (Q : U → Type) (x : Q p) (h : p = p),
x = eq_rect p Q x p h ]
```

This axiom says that it is permissible to simplify pattern matches over proofs of equalities like `e = e`. The axiom is logically equivalent to some simpler corollaries.

```
Corollary UIP_refl : ∀ A (x : A) (pf : x = x), pf = refl_equal x.
  intros; replace pf with (eq_rect x (eq x) (refl_equal x) x pf); [
    symmetry; apply eq_rect_eq
    | exact (match pf as pf' return match pf' in _ = y return x = y with
                                      | refl_equal ⇒ refl_equal x
                                    end = pf' with
               | refl_equal ⇒ refl_equal _
             end) ].
Qed.

Corollary UIP : ∀ A (x y : A) (pf1 pf2 : x = y), pf1 = pf2.
  intros; generalize pf1 pf2; subst; intros;
    match goal with
      | [ ⊢ ?pf1 = ?pf2 ] ⇒ rewrite (UIP_refl pf1); rewrite (UIP_refl pf2); reflexivity
    end.
Qed.
```

These corollaries are special cases of proof irrelevance. In developments that only need proof irrelevance for equality, there is no need to assert full irrelevance.

Another facet of proof irrelevance is that, like excluded middle, it is often provable for specific propositions. For instance, UIP is provable whenever the type A has a decidable equality operation. The module *Eqdep_dec* of the standard library contains a proof. A similar phenomenon applies to other notable cases, including less-than proofs. Thus, it is often possible to use proof irrelevance without asserting axioms.

There are two more basic axioms that are often assumed, to avoid complications that do not arise in set theory.

```
Require Import FunctionalExtensionality.
Print functional_extensionality_dep.
*** [ functional_extensionality_dep :
∀ (A : Type) (B : A → Type) (f g : ∀ x : A, B x),
(∀ x : A, f x = g x) → f = g ]
```

This axiom says that two functions are equal if they map equal inputs to equal outputs. Such facts are not provable in general in CIC, but it is consistent to assume that they are.

A simple corollary shows that the same property applies to predicates. In some cases, one might prefer to assert this corollary as the axiom, to restrict the consequences to proofs

and not programs.

```
Corollary predicate_extensionality : ∀ (A : Type) (B : A → Prop) (f g : ∀ x : A, B x),
  (∀ x : A, f x = g x) → f = g.
  intros; apply functional_extensionality_dep; assumption.
Qed.
```

11.3.2 Axioms of Choice

Some Coq axioms are also points of contention in mainstream math. The most prominent example is the axiom of choice. In fact, there are multiple versions that we might consider, and, considered in isolation, none of these versions means quite what it means in classical set theory.

First, it is possible to implement a choice operator *without* axioms in some potentially surprising cases.

```
Require Import ConstructiveEpsilon.
Check constructive_definite_description.
```

```
constructive_definite_description
     : ∀ (A : Set) (f : A → nat) (g : nat → A),
       (∀ x : A, g (f x) = x) →
       ∀ P : A → Prop,
       (∀ x : A, {P x} + {¬ P x}) →
       (∃! x : A, P x) → {x : A | P x}
```

```
Print Assumptions constructive_definite_description.
```

Closed *under the global context*

This function transforms a decidable predicate P into a function that produces an element satisfying P from a proof that such an element exists. The functions f and g, in conjunction with an associated injectivity property, are used to express the idea that the set A is countable. Under these conditions, a simple brute force algorithm gets the job done: we just enumerate all elements of A, stopping when we find one satisfying P. The existence proof, specified in terms of *unique* existence $\exists!$, guarantees termination. The definition of this operator in Coq uses some interesting techniques, as seen in the implementation of the *ConstructiveEpsilon* module.

Countable choice is provable in set theory without appealing to the general axiom of choice. To support the more general principle in Coq, we must also add an axiom. Here is a functional version of the axiom of unique choice.

```
Require Import ClassicalUniqueChoice.
Check dependent_unique_choice.
```

```
dependent_unique_choice
     : ∀ (A : Type) (B : A → Type) (R : ∀ x : A, B x → Prop),
       (∀ x : A, ∃! y : B x, R x y) →
```

$$\exists\ \mathsf{f} : \forall\ x\ :\ A,\ B\ x, \forall\ x\ :\ A,\ R\ x\ (\mathsf{f}\ x)$$

This axiom lets us convert a relational specification R into a function implementing that specification. We need only prove that R is truly a function. An alternate, stronger formulation applies to cases where R maps each input to one or more outputs. We also simplify the statement of the theorem by considering only non-dependent function types.

```
Require Import ClassicalChoice.
Check choice.
```

```
choice
     : ∀ (A B : Type) (R : A → B → Prop),
       (∀ x : A, ∃ y : B, R x y) →
       ∃ f : A → B, ∀ x : A, R x (f x)
```

This principle is proved as a theorem, based on the unique choice axiom and an additional axiom of relational choice from the *RelationalChoice* module.

In set theory, the axiom of choice is a fundamental philosophical commitment one makes about the universe of sets. In Coq, the choice axioms say something weaker. For instance, consider the simple restatement of the choice axiom where we replace existential quantification by its Curry-Howard analogue, subset types.

```
Definition choice_Set (A B : Type) (R : A → B → Prop) (H : ∀ x : A, {y : B | R x y})
  : {f : A → B | ∀ x : A, R x (f x)} :=
  exist (fun f ⇒ ∀ x : A, R x (f x))
  (fun x ⇒ proj1_sig (H x)) (fun x ⇒ proj2_sig (H x)).
```

Via the Curry-Howard correspondence, this "axiom" can be taken to have the same meaning as the original. It is implemented trivially as a transformation not much deeper than uncurrying. Thus, we see that the utility of the axioms that we mentioned earlier comes in their usage to build programs from proofs. Normal set theory has no explicit proofs, so the meaning of the usual axiom of choice is subtlely different. In Gallina, the axioms implement a controlled relaxation of the restrictions on information flow from proofs to programs.

However, when we combine an axiom of choice with the law of the excluded middle, the idea of "choice" becomes more interesting. Excluded middle gives us a highly non-computational way of constructing proofs, but it does not change the computational nature of programs. Thus, the axiom of choice is still giving us a way of translating between two different sorts of "programs," but the input programs (which are proofs) may be written in a rich language that goes beyond normal computability. This truly is more than repackaging a function with a different type.

The Coq tools support a command-line flag `-impredicative-set`, which modifies Gallina in a more fundamental way by making `Set` impredicative. A term like $\forall\ T$: `Set`, T has type `Set`, and inductive definitions in `Set` may have constructors that quantify over arguments of any types. To maintain consistency, an elimination restriction must be imposed, similarly to the restriction for `Prop`. The restriction only applies to large inductive types, where some

constructor quantifies over a type of type `Type`. In such cases, a value in this inductive type may only be pattern-matched over to yield a result type whose type is `Set` or `Prop`. This contrasts with `Prop`, where the restriction applies even to non-large inductive types, and where the result type may only have type `Prop`.

In old versions of Coq, `Set` was impredicative by default. Later versions make `Set` predicative to avoid inconsistency with some classical axioms. In particular, one should watch out when using impredicative `Set` with axioms of choice. In combination with excluded middle or predicate extensionality, this can lead to inconsistency. Impredicative `Set` can be useful for modeling inherently impredicative mathematical concepts, but almost all Coq developments get by fine without it.

11.3.3 Axioms and Computation

One additional axiom-related wrinkle arises from an aspect of Gallina that is very different from set theory: a notion of *computational equivalence* is central to the definition of the formal system. Axioms tend not to play well with computation. Consider this example. We start by implementing a function that uses a type equality proof to perform a safe type-cast.

```
Definition cast (x y : Set) (pf : x = y) (v : x) : y :=
  match pf with
    | refl_equal ⇒ v
  end.
```

Computation over programs that use cast can proceed smoothly.

```
Eval compute in (cast (refl_equal (nat → nat)) (fun n ⇒ S n)) 12.
```

```
    = 13
    : nat
```

Things do not go as smoothly when we use cast with proofs that rely on axioms.

```
Theorem t3 : (∀ n : nat, fin (S n)) = (∀ n : nat, fin (n + 1)).
  change ((∀ n : nat, (fun n ⇒ fin (S n)) n) = (∀ n : nat, (fun n ⇒ fin (n + 1)) n));
    rewrite (functional_extensionality (fun n ⇒ fin (n + 1)) (fun n ⇒ fin (S n))); crush.
Qed.

Eval compute in (cast t3 (fun _ ⇒ First)) 12.
```

```
    = match t3 in (_ = P) return P with
      | refl_equal ⇒ fun n : nat ⇒ First
      end 12
    : fin (12 + 1)
```

Computation gets stuck in a pattern-match on the proof t3. The structure of t3 is not known, so the match cannot proceed. It turns out a more basic problem leads to this

particular situation. We ended the proof of t3 with Qed, so the definition of t3 is not available to computation. That is easily fixed.

Reset t3.

```
Theorem t3 : (∀ n : nat, fin (S n)) = (∀ n : nat, fin (n + 1)).
  change ((∀ n : nat, (fun n ⇒ fin (S n)) n) = (∀ n : nat, (fun n ⇒ fin (n + 1)) n));
    rewrite (functional_extensionality (fun n ⇒ fin (n + 1)) (fun n ⇒ fin (S n))); crush.
Defined.

Eval compute in (cast t3 (fun _ ⇒ First)) 12.

    = match
        match
          match
            functional_extensionality
    ....
```

We elide most of the details. A very unwieldy tree of nested matches on equality proofs appears. This time evaluation really *is* stuck on a use of an axiom.

If we are careful in using tactics to prove an equality, we can still compute with casts over the proof.

```
Lemma plus1 : ∀ n, S n = n + 1.
  induction n; simpl; intuition.
Defined.

Theorem t4 : ∀ n, fin (S n) = fin (n + 1).
  intro; f_equal; apply plus1.
Defined.

Eval compute in cast (t4 13) First.
    = First
    : fin (13 + 1)
```

Part III

Proof Engineering

Chapter 12

Proof Search in Ltac

We have seen many examples of proof automation so far. This chapter aims to give a principled presentation of the features of Ltac, focusing in particular on the Ltac `match` construct, which supports a novel approach to backtracking search. First, though, we will run through some useful automation tactics that are built into Coq. They are described in detail in the manual, so we only outline what is possible.

12.1 Some Built-In Automation Tactics

A number of tactics are called repeatedly by *crush*. `intuition` simplifies propositional structure of goals. `congruence` applies the rules of equality and congruence closure, plus properties of constructors of inductive types. The `omega` tactic provides a complete decision procedure for a theory that is called quantifier-free linear arithmetic or Presburger arithmetic, depending on whom you ask. That is, `omega` proves any goal that follows from looking only at parts of that goal that can be interpreted as propositional formulas whose atomic formulas are basic comparison operations on natural numbers or integers.

The `ring` tactic solves goals by appealing to the axioms of rings or semi-rings (as in algebra), depending on the type involved. Coq developments may declare new types to be parts of rings and semi-rings by proving the associated axioms. There is a simlar tactic *field* for simplifying values in fields by conversion to fractions over rings. Both `ring` and *field* can only solve goals that are equalities. The *fourier* tactic uses Fourier's method to prove inequalities over real numbers, which are axiomatized in the Coq standard library.

The *setoid* facility makes it possible to register new equivalence relations to be understood by tactics like `rewrite`. For instance, `Prop` is registered as a setoid with the equivalence relation "if and only if." The ability to register new setoids can be very useful in proofs of a kind common in math, where all reasoning is done after "modding out by a relation."

12.2 Hint Databases

Another class of built-in tactics includes `auto`, `eauto`, and autorewrite. These are based on *hint databases*, which we have seen extended in many examples so far. These tactics are important, because, in Ltac programming, we cannot create "global variables" whose values can be extended seamlessly by different modules in different source files. We have seen the advantages of hints so far, where *crush* can be defined once and for all, while still automatically applying the hints we add throughout developments.

The basic hints for `auto` and `eauto` are `Hint Immediate` *lemma*, asking to try solving a goal immediately by applying a lemma and discharging any hypotheses with a single proof step each; `Resolve` *lemma*, which does the same but may add new premises that are themselves to be subjects of nested proof search; *Constructor* **type**, which acts like `Resolve` applied to every constructor of an inductive type; and `Unfold` ident, which tries unfolding ident when it appears at the head of a proof goal. Each of these `Hint` commands may be used with a suffix, as in `Hint Resolve` *lemma* : *my_db*. This adds the hint only to the specified database, so that it would only be used by, for instance, `auto with` *my_db*. An additional argument to `auto` specifies the maximum depth of proof trees to search in depth-first order, as in `auto` 8 or `auto` 8 `with` *my_db*. The default depth is 5.

All of these `Hint` commands can be issued alternatively with a more primitive hint kind, `Extern`. A few examples should do best to explain how `Hint Extern` works.

```
Theorem bool_neq : true ≠ false.
  auto.
```

crush would have discharged this goal, but the default hint database for `auto` contains no hint that applies.

```
Abort.
```

It is hard to come up with a **bool**-specific hint that is not just a restatement of the theorem we mean to prove. Luckily, a simpler form suffices.

```
Hint Extern 1 (_ ≠ _) ⇒ congruence.

Theorem bool_neq : true ≠ false.
  auto.
Qed.
```

Our hint says: "whenever the conclusion matches the pattern _ ≠ _, try applying `congruence`." The 1 is a cost for this rule. During proof search, whenever multiple rules apply, rules are tried in increasing cost order, so it pays to assign high costs to relatively expensive `Extern` hints.

`Extern` hints may be implemented with the full Ltac language. This example shows a case where a hint uses a `match`.

```
Section forall_and.
  Variable A : Set.
  Variables P Q : A → Prop.
```

```
Hypothesis both : ∀ x, P x ∧ Q x.

Theorem forall_and : ∀ z, P z.
  crush.
```

crush makes no progress beyond what `intros` would have accomplished. `auto` will not apply the hypothesis *both* to prove the goal, because the conclusion of *both* does not unify with the conclusion of the goal. However, we can teach `auto` to handle this kind of goal.

```
  Hint Extern 1 (P ?X) ⇒
    match goal with
      | [ H : ∀ x, P x ∧ _ ⊢ _ ] ⇒ apply (proj1 (H X))
    end.

  auto.
Qed.
```

We see that an `Extern` pattern may bind unification variables that we use in the associated tactic. `proj1` is a function from the standard library for extracting a proof of R from a proof of $R \wedge S$.

```
End forall_and.
```

After our success on this example, we might get more ambitious and seek to generalize the hint to all possible predicates P.

```
Hint Extern 1 (?P ?X) ⇒
  match goal with
    | [ H : ∀ x, P x ∧ _ ⊢ _ ] ⇒ apply (proj1 (H X))
  end.
```

User error: *Bound head variable*

Coq's `auto` hint databases work as tables mapping *head symbols* to lists of tactics to try. Because of this, the constant head of an `Extern` pattern must be determinable statically. In our first `Extern` hint, the head symbol was `not`, since $x \neq y$ desugars to `not` (`eq` x y); and, in the second example, the head symbol was P.

This restriction on `Extern` hints is the main limitation of the `auto` mechanism, preventing us from using it for general context simplifications that are not keyed off of the form of the conclusion. This is perhaps just as well, since we can often code more efficient tactics with specialized Ltac programs, and we will see how in later sections of the chapter.

We have used `Hint` *Rewrite* in many examples so far. *crush* uses these hints by calling `autorewrite`. Our rewrite hints have taken the form `Hint` *Rewrite lemma* : *cpdt*, adding them to the *cpdt* rewrite database. This is because, in contrast to `auto`, `autorewrite` has no default database. Thus, we set the convention that *crush* uses the *cpdt* database.

This example shows a direct use of `autorewrite`.

```
Section autorewrite.
```

```
Variable A : Set.
Variable f : A → A.

Hypothesis f_f : ∀ x, f (f x) = f x.

Hint Rewrite f_f : my_db.

Lemma f_f_f : ∀ x, f (f (f x)) = f x.
  intros; autorewrite with my_db; reflexivity.
Qed.
```

There are a few ways in which autorewrite can lead to trouble when insufficient care is taken in choosing hints. First, the set of hints may define a nonterminating rewrite system, in which case invocations to autorewrite may not terminate. Second, we may add hints that "lead autorewrite down the wrong path." For instance:

```
Section garden_path.
  Variable g : A → A.
  Hypothesis f_g : ∀ x, f x = g x.
  Hint Rewrite f_g : my_db.

  Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with my_db.

============================
 g (g (g x)) = g x

    Abort.
```

Our new hint was used to rewrite the goal into a form where the old hint could no longer be applied. This "non-monotonicity" of rewrite hints contrasts with the situation for auto, where new hints may slow down proof search but can never "break" old proofs.

```
Reset garden_path.
```

autorewrite works with quantified equalities that include additional premises, but we must be careful to avoid similar incorrect rewritings.

```
Section garden_path.
  Variable P : A → Prop.
  Variable g : A → A.
  Hypothesis f_g : ∀ x, P x → f x = g x.
  Hint Rewrite f_g : my_db.

  Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with my_db.


============================
 g (g (g x)) = g x
```

```
subgoal 2 is:
 P x
subgoal 3 is:
 P (f x)
subgoal 4 is:
 P (f x)
```

```
Abort.
```

The inappropriate rule fired the same three times as before, even though we know we will not be able to prove the premises.

Reset garden_path.

Our final, successful, attempt uses an extra argument to `Hint` *Rewrite* that specifies a tactic to apply to generated premises.

```
Section garden_path.
  Variable P : A → Prop.
  Variable g : A → A.
  Hypothesis f_g : ∀ x, P x → f x = g x.
  Hint Rewrite f_g using assumption : my_db.

  Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with my_db; reflexivity.
  Qed.
```

autorewrite will still use *f_g* when the generated premise is among our assumptions.

```
  Lemma f_f_f_g : ∀ x, P x → f (f x) = g x.
    intros; autorewrite with my_db; reflexivity.
  Qed.
End garden_path.
```

It can also be useful to use the autorewrite `with` *db* `in` * form, which does rewriting in hypotheses, as well as in the conclusion.

```
Lemma in_star : ∀ x y, f (f (f (f x))) = f (f y)
  → f x = f (f (f y)).
  intros; autorewrite with my_db in *; assumption.
Qed.
```

```
End autorewrite.
```

12.3 Ltac Programming Basics

We have already seen many examples of Ltac programs. In the rest of this chapter, we attempt to give a more principled introduction to the important features and design patterns.

One common use for `match` tactics is identification of subjects for case analysis, as we see in this tactic definition.

```
Ltac find_if :=
  match goal with
    | [ ⊢ if ?X then _ else _ ] ⇒ destruct X
  end.
```

The tactic checks if the conclusion is an `if`, `destruct`ing the test expression if so. Certain classes of theorem are trivial to prove automatically with such a tactic.

```
Theorem hmm : ∀ (a b c : bool),
  if a
    then if b
      then True
      else True
    else if c
      then True
      else True.
  intros; repeat find_if; constructor.
Qed.
```

The `repeat` that we use here is called a *tactical*, or tactic combinator. The behavior of `repeat` t is to loop through running t, running t on all generated subgoals, running t on *their* generated subgoals, and so on. When t fails at any point in this search tree, that particular subgoal is left to be handled by later tactics. Thus, it is important never to use `repeat` with a tactic that always succeeds.

Another very useful Ltac building block is *context patterns*.

```
Ltac find_if_inside :=
  match goal with
    | [ ⊢ context[if ?X then _ else _] ] ⇒ destruct X
  end.
```

The behavior of this tactic is to find any subterm of the conclusion that is an `if` and then `destruct` the test expression. This version subsumes *find_if*.

```
Theorem hmm' : ∀ (a b c : bool),
  if a
    then if b
      then True
      else True
    else if c
      then True
      else True.
  intros; repeat find_if_inside; constructor.
Qed.
```

We can also use *find_if_inside* to prove goals that *find_if* does not simplify sufficiently.

```
Theorem hmm2 : ∀ (a b : bool),
  (if a then 42 else 42) = (if b then 42 else 42).
  intros; repeat find_if_inside; reflexivity.
Qed.
```

Many decision procedures can be coded in Ltac via "`repeat match` loops." For instance, we can implement a subset of the functionality of `tauto`.

```
Ltac my_tauto :=
  repeat match goal with
           | [ H : ?P ⊢ ?P ] ⇒ exact H

           | [ ⊢ True ] ⇒ constructor
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ ⊢ _ → _ ] ⇒ intro

           | [ H : False ⊢ _ ] ⇒ destruct H
           | [ H : _ ∧ _ ⊢ _ ] ⇒ destruct H
           | [ H : _ ∨ _ ⊢ _ ] ⇒ destruct H

           | [ H1 : ?P → ?Q, H2 : ?P ⊢ _ ] ⇒
             let H := fresh "H" in
               generalize (H1 H2); clear H1; intro H
         end.
```

Since `match` patterns can share unification variables between hypothesis and conclusion patterns, it is easy to figure out when the conclusion matches a hypothesis. The `exact` tactic solves a goal completely when given a proof term of the proper type.

It is also trivial to implement the "introduction rules" for a few of the connectives. Implementing elimination rules is only a little more work, since we must give a name for a hypothesis to `destruct`.

The last rule implements modus ponens. The most interesting part is the use of the Ltac-level `let` with a `fresh` expression. `fresh` takes in a name base and returns a fresh hypothesis variable based on that name. We use the new name variable H as the name we assign to the result of modus ponens. The use of `generalize` changes our conclusion to be an implication from Q. We clear the original hypothesis and move Q into the context with name H.

```
Section propositional.
  Variables P Q R : Prop.

  Theorem propositional : (P ∨ Q ∨ False) ∧ (P → Q) → True ∧ Q.
    my_tauto.
  Qed.
End propositional.
```

It was relatively easy to implement modus ponens, because we do not lose information

by clearing every implication that we use. If we want to implement a similarly-complete procedure for quantifier instantiation, we need a way to ensure that a particular proposition is not already included among our hypotheses. To do that effectively, we first need to learn a bit more about the semantics of `match`.

It is tempting to assume that `match` works like it does in ML. In fact, there are a few critical differences in its behavior. One is that we may include arbitrary expressions in patterns, instead of being restricted to variables and constructors. Another is that the same variable may appear multiple times, inducing an implicit equality constraint.

There is a related pair of two other differences that are much more important than the others. `match` has a *backtracking semantics for failure*. In ML, pattern matching works by finding the first pattern to match and then executing its body. If the body raises an exception, then the overall match raises the same exception. In Coq, failures in case bodies instead trigger continued search through the list of cases.

For instance, this (unnecessarily verbose) proof script works:

```
Theorem m1 : True.
  match goal with
    | [ ⊢ _ ] ⇒ intro
    | [ ⊢ True ] ⇒ constructor
  end.
Qed.
```

The first case matches trivially, but its body tactic fails, since the conclusion does not begin with a quantifier or implication. In a similar ML match, that would mean that the whole pattern-match fails. In Coq, we backtrack and try the next pattern, which also matches. Its body tactic succeeds, so the overall tactic succeeds as well.

The example shows how failure can move to a different pattern within a `match`. Failure can also trigger an attempt to find *a different way of matching a single pattern*. Consider another example:

```
Theorem m2 : ∀ P Q R : Prop, P → Q → R → Q.
  intros; match goal with
            | [ H : _ ⊢ _ ] ⇒ idtac H
          end.
```

Coq prints "*H1*". By applying *idtac* with an argument, a convenient debugging tool for "leaking information out of `match`es," we see that this `match` first tries binding H to *H1*, which cannot be used to prove Q. Nonetheless, the following variation on the tactic succeeds at proving the goal:

```
  match goal with
    | [ H : _ ⊢ _ ] ⇒ exact H
  end.
Qed.
```

The tactic first unifies H with *H1*, as before, but `exact` H fails in that case, so the tactic engine searches for more possible values of H. Eventually, it arrives at the correct value, so

that `exact` H and the overall tactic succeed.

Now we are equipped to implement a tactic for checking that a proposition is not among our hypotheses:

```
Ltac notHyp P :=
  match goal with
    | [ _ : P ⊢ _ ] ⇒ fail 1
    | _ ⇒
      match P with
        | ?P1 ∧ ?P2 ⇒ first [ notHyp P1 | notHyp P2 | fail 2 ]
        | _ ⇒ idtac
      end
  end.
```

We use the equality checking that is built into pattern-matching to see if there is a hypothesis that matches the proposition exactly. If so, we use the `fail` tactic. Without arguments, `fail` signals normal tactic failure, as you might expect. When `fail` is passed an argument n, n is used to count outwards through the enclosing cases of backtracking search. In this case, `fail` 1 says "fail not just in this pattern-matching branch, but for the whole `match`." The second case will never be tried when the `fail` 1 is reached.

This second case, used when P matches no hypothesis, checks if P is a conjunction. Other simplifications may have split conjunctions into their component formulas, so we need to check that at least one of those components is also not represented. To achieve this, we apply the *first* tactical, which takes a list of tactics and continues down the list until one of them does not fail. The `fail` 2 at the end says to `fail` both the *first* and the `match` wrapped around it.

The body of the ?P1 ∧ ?P2 case guarantees that, if it is reached, we either succeed completely or fail completely. Thus, if we reach the wildcard case, P is not a conjunction. We use *idtac*, a tactic that would be silly to apply on its own, since its effect is to succeed at doing nothing. Nonetheless, *idtac* is a useful placeholder for cases like what we see here.

With the non-presence check implemented, it is easy to build a tactic that takes as input a proof term and adds its conclusion as a new hypothesis, only if that conclusion is not already present, failing otherwise.

```
Ltac extend pf :=
  let t := type of pf in
    notHyp t; generalize pf; intro.
```

We see the useful **type** *of* operator of Ltac. This operator could not be implemented in Gallina, but it is easy to support in Ltac. We end up with t bound to the type of *pf*. We check that t is not already present. If so, we use a `generalize`/`intro` combo to add a new hypothesis proved by *pf*.

With these tactics defined, we can write a tactic *completer* for adding to the context all consequences of a set of simple first-order formulas.

```
Ltac completer :=
```

```
  repeat match goal with
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ H : _ ∧ _ ⊢ _ ] ⇒ destruct H
           | [ H : ?P → ?Q, H' : ?P ⊢ _ ] ⇒
             generalize (H H'); clear H; intro H
           | [ ⊢ ∀ x, _ ] ⇒ intro

           | [ H : ∀ x, ?P x → _, H' : ?P ?X ⊢ _ ] ⇒
             extend (H X H')
         end.
```

We use the same kind of conjunction and implication handling as previously. Note that, since $\rightarrow$ is the special non-dependent case of $\forall$, the fourth rule handles `intro` for implications, too.

In the fifth rule, when we find a $\forall$ fact H with a premise matching one of our hypotheses, we add the appropriate instantiation of H's conclusion, if we have not already added it.

We can check that *completer* is working properly:

```
Section firstorder.
  Variable A : Set.
  Variables P Q R S : A → Prop.

  Hypothesis H1 : ∀ x, P x → Q x ∧ R x.
  Hypothesis H2 : ∀ x, R x → S x.

  Theorem fo : ∀ x, P x → S x.
    completer.

  x : A
  H : P x
  H0 : Q x
  H3 : R x
  H4 : S x
  ============================
   S x

    assumption.
  Qed.
End firstorder.
```

We narrowly avoided a subtle pitfall in our definition of *completer*. Let us try another definition that even seems preferable to the original, to the untrained eye.

```
Ltac completer' :=
  repeat match goal with
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ H : _ ∧ _ ⊢ _ ] ⇒ destruct H
           | [ H : ?P → _, H' : ?P ⊢ _ ] ⇒
```

```
      generalize (H H'); clear H; intro H
    | [ ⊢ ∀ x, _ ] ⇒ intro

    | [ H : ∀ x, ?P x → _, H' : ?P ?X ⊢ _ ] ⇒
      extend (H X H')
  end.
```

The only difference is in the modus ponens rule, where we have replaced an unused unification variable ?Q with a wildcard. Let us try our example again with this version:

```
Section firstorder'.
  Variable A : Set.
  Variables P Q R S : A → Prop.

  Hypothesis H1 : ∀ x, P x → Q x ∧ R x.
  Hypothesis H2 : ∀ x, R x → S x.

  Theorem fo' : ∀ x, P x → S x.

    completer'.
```

Coq loops forever at this point. What went wrong?

```
  Abort.
End firstorder'.
```

A few examples should illustrate the issue. Here we see a `match`-based proof that works fine:

```
Theorem t1 : ∀ x : nat, x = x.
  match goal with
    | [ ⊢ ∀ x, _ ] ⇒ trivial
  end.
Qed.
```

This one fails.

```
Theorem t1' : ∀ x : nat, x = x.

  match goal with
    | [ ⊢ ∀ x, ?P ] ⇒ trivial
  end.
```

User error: *No matching clauses* `for match` *goal*

```
Abort.
```

The problem is that unification variables may not contain locally-bound variables. In this case, ?P would need to be bound to $x = x$, which contains the local quantified variable x. By using a wildcard in the earlier version, we avoided this restriction.

The Coq 8.2 release includes a special pattern form for a unification variable with an explicit set of free variables. That unification variable is then bound to a function from the free variables to the "real" value. In Coq 8.1 and earlier, there is no such workaround.

No matter which version you use, it is important to be aware of this restriction. As we have alluded to, the restriction is the culprit behind the infinite-looping behavior of *completer'*. We unintentionally match quantified facts with the modus ponens rule, circumventing the "already present" check and leading to different behavior.

12.4 Functional Programming in Ltac

Ltac supports quite convenient functional programming, with a Lisp-with-syntax kind of flavor. However, there are a few syntactic conventions involved in getting programs to be accepted. The Ltac syntax is optimized for tactic-writing, so one has to deal with some inconveniences in writing more standard functional programs.

To illustrate, let us try to write a simple list length function. We start out writing it just like in Gallina, simply replacing `Fixpoint` (and its annotations) with `Ltac`.

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ls' ⇒ S (length ls')
  end.
```

Error: *The reference ls' was* not *found* `in` *the current environment*

At this point, we hopefully remember that pattern variable names must be prefixed by question marks in Ltac.

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒ S (length ls')
  end.
```

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

The problem is that Ltac treats the expression S (length *ls'*) as an invocation of a tactic S with argument length *ls'*. We need to use a special annotation to "escape into" the Gallina parsing nonterminal.

```
Ltac length ls :=
  match ls with
```

```
    | nil ⇒ O
    | _ :: ?ls' ⇒ constr:(S (length ls'))
  end.
```

This definition is accepted. It can be a little awkward to test Ltac definitions like this. Here is one method.

```
Goal False.
  let n := length (1 :: 2 :: 3 :: nil) in
    pose n.

  n := S (length (2 :: 3 :: nil)) : nat
  ============================
   False
```

We use the *pose* tactic, which extends the proof context with a new variable that is set equal to particular a term. We could also have used *idtac n* in place of *pose n*, which would have printed the result without changing the context.

n only has the length calculation unrolled one step. What has happened here is that, by escaping into the *constr* nonterminal, we referred to the length function of Gallina, rather than the length Ltac function that we are defining.

```
Abort.
```

Reset length.

The thing to remember is that Gallina terms built by tactics must be bound explicitly via `let` or a similar technique, rather than inserting Ltac calls directly in other Gallina terms.

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒
      let ls'' := length ls' in
        constr:(S ls'')
  end.

Goal False.
  let n := length (1 :: 2 :: 3 :: nil) in
    pose n.

  n := 3 : nat
  ============================
   False

Abort.
```

We can also use anonymous function expressions and local function definitions in Ltac, as this example of a standard list map function shows.

```
Ltac map T f :=
  let rec map' ls :=
    match ls with
      | nil ⇒ constr:(@nil T)
      | ?x :: ?ls' ⇒
        let x' := f x in
          let ls'' := map' ls' in
            constr:(x' :: ls'')
    end in
  map'.
```

Ltac functions can have no implicit arguments. It may seem surprising that we need to pass T, the carried type of the output list, explicitly. We cannot just use **type** *of* f, because f is an Ltac term, not a Gallina term, and Ltac programs are dynamically typed. f could use very syntactic methods to decide to return differently typed terms for different inputs. We also could not replace *constr*:(@nil T) with *constr*:nil, because we have no strongly-typed context to use to infer the parameter to nil. Luckily, we do have sufficient context within *constr*:(x' :: *ls''*).

Sometimes we need to employ the opposite direction of "nonterminal escape," when we want to pass a complicated tactic expression as an argument to another tactic, as we might want to do in invoking map.

```
Goal False.
  let ls := map (nat * nat)%type ltac:(fun x ⇒ constr:(x, x)) (1 :: 2 :: 3 :: nil) in
    pose ls.

  l := (1, 1) :: (2, 2) :: (3, 3) :: nil : list (nat * nat)
  ============================
   False

Abort.
```

12.5 Recursive Proof Search

Deciding how to instantiate quantifiers is one of the hardest parts of automated first-order theorem proving. For a given problem, we can consider all possible bounded-length sequences of quantifier instantiations, applying only propositional reasoning at the end. This is probably a bad idea for almost all goals, but it makes for a nice example of recursive proof search procedures in Ltac.

We can consider the maximum "dependency chain" length for a first-order proof. We define the chain length for a hypothesis to be 0, and the chain length for an instantiation of a quantified fact to be one greater than the length for that fact. The tactic *inster* ***n*** is meant to try all possible proofs with chain length at most ***n***.

```
Ltac inster n :=
```

```
  intuition;
    match n with
      | S ?n' ⇒
        match goal with
          | [ H : ∀ x : ?T, _, x : ?T ⊢ _ ] ⇒ generalize (H x); inster n'
        end
    end.
```

inster begins by applying propositional simplification. Next, it checks if any chain length remains. If so, it tries all possible ways of instantiating quantified hypotheses with properly-typed local variables. It is critical to realize that, if the recursive call *inster n'* fails, then the `match` *goal* just seeks out another way of unifying its pattern against proof state. Thus, this small amount of code provides an elegant demonstration of how backtracking `match` enables exhaustive search.

We can verify the efficacy of *inster* with two short examples. The built-in `firstorder` tactic (with no extra arguments) is able to prove the first but not the second.

```
Section test_inster.
  Variable A : Set.
  Variables P Q : A → Prop.
  Variable f : A → A.
  Variable g : A → A → A.

  Hypothesis H1 : ∀ x y, P (g x y) → Q (f x).

  Theorem test_inster : ∀ x y, P (g x y) → Q (f x).
    inster 2.
  Qed.

  Hypothesis H3 : ∀ u v, P u ∧ P v ∧ u ≠ v → P (g u v).
  Hypothesis H4 : ∀ u, Q (f u) → P u ∧ P (f u).

  Theorem test_inster2 : ∀ x y, x ≠ y → P x → Q (f y) → Q (f x).
    inster 3.
  Qed.
End test_inster.
```

The style employed in the definition of *inster* can seem very counterintuitive to functional programmers. Usually, functional programs accumulate state changes in explicit arguments to recursive functions. In Ltac, the state of the current subgoal is always implicit. Nonetheless, in contrast to general imperative programming, it is easy to undo any changes to this state, and indeed such "undoing" happens automatically at failures within `match`es. In this way, Ltac programming is similar to programming in Haskell with a stateful failure monad that supports a composition operator along the lines of the *first* tactical.

Functional programming purists may react indignantly to the suggestion of programming this way. Nonetheless, as with other kinds of "monadic programming," many problems are much simpler to solve with Ltac than they would be with explicit, pure proof manipulation in ML or Haskell. To demonstrate, we will write a basic simplification procedure for logical

implications.

This procedure is inspired by one for separation logic, where conjuncts in formulas are thought of as "resources," such that we lose no completeness by "crossing out" equal conjuncts on the two sides of an implication. This process is complicated by the fact that, for reasons of modularity, our formulas can have arbitrary nested tree structure (branching at conjunctions) and may include existential quantifiers. It is helpful for the matching process to "go under" quantifiers and in fact decide how to instantiate existential quantifiers in the conclusion.

To distinguish the implications that our tactic handles from the implications that will show up as "plumbing" in various lemmas, we define a wrapper definition, a notation, and a tactic.

```
Definition imp (P1 P2 : Prop) := P1 → P2.
Infix "->" := imp (no associativity, at level 95).
Ltac imp := unfold imp; firstorder.
```

These lemmas about imp will be useful in the tactic that we will write.

```
Theorem and_True_prem : ∀ P Q,
  (P ∧ True -> Q)
  → (P -> Q).
  imp.
Qed.

Theorem and_True_conc : ∀ P Q,
  (P -> Q ∧ True)
  → (P -> Q).
  imp.
Qed.

Theorem assoc_prem1 : ∀ P Q R S,
  (P ∧ (Q ∧ R) -> S)
  → ((P ∧ Q) ∧ R -> S).
  imp.
Qed.

Theorem assoc_prem2 : ∀ P Q R S,
  (Q ∧ (P ∧ R) -> S)
  → ((P ∧ Q) ∧ R -> S).
  imp.
Qed.

Theorem comm_prem : ∀ P Q R,
  (P ∧ Q -> R)
  → (Q ∧ P -> R).
  imp.
Qed.

Theorem assoc_conc1 : ∀ P Q R S,
```

```
  (S -> P ∧ (Q ∧ R))
  → (S -> (P ∧ Q) ∧ R).
  imp.
Qed.

Theorem assoc_conc2 : ∀ P Q R S,
  (S -> Q ∧ (P ∧ R))
  → (S -> (P ∧ Q) ∧ R).
  imp.
Qed.

Theorem comm_conc : ∀ P Q R,
  (R -> P ∧ Q)
  → (R -> Q ∧ P).
  imp.
Qed.
```

The first order of business in crafting our *matcher* tactic will be auxiliary support for searching through formula trees. The *search_prem* tactic implements running its tactic argument *tac* on every subformula of an imp premise. As it traverses a tree, *search_prem* applies some of the above lemmas to rewrite the goal to bring different subformulas to the head of the goal. That is, for every subformula P of the implication premise, we want P to "have a turn," where the premise is rearranged into the form $P \wedge Q$ for some Q. The tactic *tac* should expect to see a goal in this form and focus its attention on the first conjunct of the premise.

```
Ltac search_prem tac :=
  let rec search P :=
    tac
    || (apply and_True_prem; tac)
    || match P with
         | ?P1 ∧ ?P2 ⇒
           (apply assoc_prem1; search P1)
           || (apply assoc_prem2; search P2)
       end
  in match goal with
       | [ ⊢ ?P ∧ _ -> _ ] ⇒ search P
       | [ ⊢ _ ∧ ?P -> _ ] ⇒ apply comm_prem; search P
       | [ ⊢ _ -> _ ] ⇒ progress (tac || (apply and_True_prem; tac))
     end.
```

To understand how *search_prem* works, we turn first to the final match. If the premise begins with a conjunction, we call the *search* procedure on each of the conjuncts, or only the first conjunct, if that already yields a case where *tac* does not fail. *search P* expects and maintains the invariant that the premise is of the form $P \wedge Q$ for some Q. We pass P explicitly as a kind of decreasing induction measure, to avoid looping forever when *tac*

always fails. The second `match` case calls a commutativity lemma to realize this invariant, before passing control to *search*. The final `match` case tries applying *tac* directly and then, if that fails, changes the form of the goal by adding an extraneous **True** conjunct and calls *tac* again.

search itself tries the same tricks as in the last case of the final `match`. Additionally, if neither works, it checks if P is a conjunction. If so, it calls itself recursively on each conjunct, first applying associativity lemmas to maintain the goal-form invariant.

We will also want a dual function *search_conc*, which does tree search through an `imp` conclusion.

```
Ltac search_conc tac :=
  let rec search P :=
    tac
    || (apply and_True_conc; tac)
    || match P with
         | ?P1 ∧ ?P2 ⇒
           (apply assoc_conc1; search P1)
           || (apply assoc_conc2; search P2)
       end
  in match goal with
       | [ ⊢ _ -> ?P ∧ _ ] ⇒ search P
       | [ ⊢ _ -> _ ∧ ?P ] ⇒ apply comm_conc; search P
       | [ ⊢ _ -> _ ] ⇒ progress (tac || (apply and_True_conc; tac))
     end.
```

Now we can prove a number of lemmas that are suitable for application by our search tactics. A lemma that is meant to handle a premise should have the form $P \wedge Q \rightarrow R$ for some interesting P, and a lemma that is meant to handle a conclusion should have the form $P \rightarrow Q \wedge R$ for some interesting Q.

```
Theorem False_prem : ∀ P Q,
  False ∧ P -> Q.
  imp.
Qed.

Theorem True_conc : ∀ P Q : Prop,
  (P -> Q)
  → (P -> True ∧ Q).
  imp.
Qed.

Theorem Match : ∀ P Q R : Prop,
  (Q -> R)
  → (P ∧ Q -> P ∧ R).
  imp.
Qed.
```

```
Theorem ex_prem : ∀ (T : Type) (P : T → Prop) (Q R : Prop),
  (∀ x, P x ∧ Q -> R)
  → (ex P ∧ Q -> R).
  imp.
Qed.

Theorem ex_conc : ∀ (T : Type) (P : T → Prop) (Q R : Prop) x,
  (Q -> P x ∧ R)
  → (Q -> ex P ∧ R).
  imp.
Qed.
```

We will also want a "base case" lemma for finishing proofs where cancelation has removed every constituent of the conclusion.

```
Theorem imp_True : ∀ P,
  P -> True.
  imp.
Qed.
```

Our final *matcher* tactic is now straightforward. First, we `intros` all variables into scope. Then we attempt simple premise simplifications, finishing the proof upon finding **False** and eliminating any existential quantifiers that we find. After that, we search through the conclusion. We remove **True** conjuncts, remove existential quantifiers by introducing unification variables for their bound variables, and search for matching premises to cancel. Finally, when no more progress is made, we see if the goal has become trivial and can be solved by imp_True. In each case, we use the tactic *simple* `apply` in place of `apply` to use a simpler, less expensive unification algorithm.

```
Ltac matcher :=
  intros;
    repeat search_prem ltac:(simple apply False_prem || (simple apply ex_prem; intro));
      repeat search_conc ltac:(simple apply True_conc || simple eapply ex_conc
        || search_prem ltac:(simple apply Match));
      try simple apply imp_True.
```

Our tactic succeeds at proving a simple example.

```
Theorem t2 : ∀ P Q : Prop,
  Q ∧ (P ∧ False) ∧ P -> P ∧ Q.
  matcher.
Qed.
```

In the generated proof, we find a trace of the workings of the search tactics.

```
Print t2.

t2 =
fun P Q : Prop ⇒
comm_prem (assoc_prem1 (assoc_prem2 (False_prem (P:=P ∧ P ∧ Q) (P ∧ Q))))
```

```
  : ∀ P Q : Prop, Q ∧ (P ∧ False) ∧ P -> P ∧ Q
```

We can also see that *matcher* is well-suited for cases where some human intervention is needed after the automation finishes.

```
Theorem t3 : ∀ P Q R : Prop,
  P ∧ Q -> Q ∧ R ∧ P.
  matcher.

  ============================
   True -> R
```

matcher canceled those conjuncts that it was able to cancel, leaving a simplified subgoal for us, much as `intuition` does.

```
Abort.
```

matcher even succeeds at guessing quantifier instantiations. It is the unification that occurs in uses of the Match lemma that does the real work here.

```
Theorem t4 : ∀ (P : nat → Prop) Q, (∃ x, P x ∧ Q) -> Q ∧ (∃ x, P x).
  matcher.
Qed.

Print t4.

t4 =
fun (P : nat → Prop) (Q : Prop) ⇒
and_True_prem
  (ex_prem (P:=fun x : nat ⇒ P x ∧ Q)
     (fun x : nat ⇒
      assoc_prem2
        (Match (P:=Q)
           (and_True_conc
              (ex_conc (fun x0 : nat ⇒ P x0) x
                 (Match (P:=P x) (imp_True (P:=True))))))))
     : ∀ (P : nat → Prop) (Q : Prop),
       (∃ x : nat, P x ∧ Q) -> Q ∧ (∃ x : nat, P x)
```

12.6 Creating Unification Variables

A final useful ingredient in tactic crafting is the ability to allocate new unification variables explicitly. Tactics like `eauto` introduce unification variable internally to support flexible proof search. While `eauto` and its relatives do *backward* reasoning, we often want to do similar *forward* reasoning, where unification variables can be useful for similar reasons.

For example, we can write a tactic that instantiates the quantifiers of a universally-quantified hypothesis. The tactic should not need to know what the appropriate instantiantiations are; rather, we want these choices filled with placeholders. We hope that, when we apply the specialized hypothesis later, syntactic unification will determine concrete values.

Before we are ready to write a tactic, we can try out its ingredients one at a time.

```
Theorem t5 : (∀ x : nat, S x > x) → 2 > 1.
  intros.

  H : ∀ x : nat, S x > x
  ============================
   2 > 1
```

To instantiate H generically, we first need to name the value to be used for x.

```
  evar (y : nat).

  H : ∀ x : nat, S x > x
  y := ?279 : nat
  ============================
   2 > 1
```

The proof context is extended with a new variable y, which has been assigned to be equal to a fresh unification variable ?279. We want to instantiate H with ?279. To get ahold of the new unification variable, rather than just its alias y, we perform a trivial call-by-value reduction in the expression y. In particular, we only request the use of one reduction rule, *delta*, which deals with definition unfolding. We pass a flag further stipulating that only the definition of y be unfolded. This is a simple trick for getting at the value of a synonym variable.

```
  let y' := eval cbv delta [y] in y in
    clear y; generalize (H y').

  H : ∀ x : nat, S x > x
  ============================
   S ?279 > ?279 → 2 > 1
```

Our instantiation was successful. We can finish by using the refined formula to replace the original.

```
  clear H; intro H.

  H : S ?281 > ?281
  ============================
```

```
  2 > 1
```

We can finish the proof by using `apply`'s unification to figure out the proper value of ?281. (The original unification variable was replaced by another, as often happens in the internals of the various tactics' implementations.)

```
  apply H.
Qed.
```

Now we can write a tactic that encapsulates the pattern we just employed, instantiating all quantifiers of a particular hypothesis.

```
Ltac insterU H :=
  repeat match type of H with
           | ∀ x : ?T, _ ⇒
             let x := fresh "x" in
               evar (x : T);
               let x' := eval cbv delta [x] in x in
                 clear x; generalize (H x'); clear H; intro H
         end.

Theorem t5' : (∀ x : nat, S x > x) → 2 > 1.
  intro H; insterU H; apply H.
Qed.
```

This particular example is somewhat silly, since `apply` by itself would have solved the goal originally. Separate forward reasoning is more useful on hypotheses that end in existential quantifications. Before we go through an example, it is useful to define a variant of *insterU* that does not clear the base hypothesis we pass to it.

```
Ltac insterKeep H :=
  let H' := fresh "H'" in
    generalize H; intro H'; insterU H'.

Section t6.
  Variables A B : Type.
  Variable P : A → B → Prop.
  Variable f : A → A → A.
  Variable g : B → B → B.

  Hypothesis H1 : ∀ v, ∃ u, P v u.
  Hypothesis H2 : ∀ v1 u1 v2 u2,
    P v1 u1
    → P v2 u2
    → P (f v1 v2) (g u1 u2).

  Theorem t6 : ∀ v1 v2, ∃ u1, ∃ u2, P (f v1 v2) (g u1 u2).
    intros.
```

Neither `eauto` nor `firstorder` is clever enough to prove this goal. We can help out by doing some of the work with quantifiers ourselves.

```
do 2 insterKeep H1.
```

Our proof state is extended with two generic instances of *H1*.

```
H' : ∃ u : B, P ?4289 u
H'0 : ∃ u : B, P ?4288 u
============================
 ∃ u1 : B, ∃ u2 : B, P (f v1 v2) (g u1 u2)
```

`eauto` still cannot prove the goal, so we eliminate the two new existential quantifiers.

```
repeat match goal with
         | [ H : ex _ ⊢ _ ] ⇒ destruct H
       end.
```

Now the goal is simple enough to solve by logic programming.

```
eauto.
Qed.
End t6.
```

Our *insterU* tactic does not fare so well with quantified hypotheses that also contain implications. We can see the problem in a slight modification of the last example. We introduce a new unary predicate Q and use it to state an additional requirement of our hypothesis *H1*.

```
Section t7.
  Variables A B : Type.
  Variable Q : A → Prop.
  Variable P : A → B → Prop.
  Variable f : A → A → A.
  Variable g : B → B → B.

  Hypothesis H1 : ∀ v, Q v → ∃ u, P v u.
  Hypothesis H2 : ∀ v1 u1 v2 u2,
    P v1 u1
    → P v2 u2
    → P (f v1 v2) (g u1 u2).

  Theorem t6 : ∀ v1 v2, Q v1 → Q v2 → ∃ u1, ∃ u2, P (f v1 v2) (g u1 u2).
    intros; do 2 insterKeep H1;
      repeat match goal with
               | [ H : ex _ ⊢ _ ] ⇒ destruct H
             end; eauto.
```

This proof script does not hit any errors until the very end, when an error message like this one is displayed.

No more *subgoals but non*-instantiated *existential variables* :
Existential 1 =
?4384 : [A : Type
B : Type
Q : $A \to$ Prop
P : $A \to B \to$ Prop
f : $A \to A \to A$
g : $B \to B \to B$
H1 : $\forall$ v : A, Q $v \to \exists$ u : B, P v u
H2 : $\forall$ (*v1* : A) (*u1* : B) (*v2* : A) (*u2* : B),
P *v1* *u1* $\to P$ *v2* *u2* $\to P$ (f *v1* *v2*) (g *u1* *u2*)
v1 : A
v2 : A
H : Q *v1*
H0 : Q *v2*
H' : Q *v2* $\to \exists$ u : B, P *v2* $u \vdash Q$ *v2*]

There is another similar line about a different existential variable. Here, "existential variable" means what we have also called "unification variable." In the course of the proof, some unification variable ?4384 was introduced but never unified. Unification variables are just a device to structure proof search; the language of Gallina proof terms does not include them. Thus, we cannot produce a proof term without instantiating the variable.

The error message shows that ?4384 is meant to be a proof of Q *v2* in a particular proof state, whose variables and hypotheses are displayed. It turns out that ?4384 was created by *insterU*, as the value of a proof to pass to *H1*. Recall that, in Gallina, implication is just a degenerate case of $\forall$ quantification, so the *insterU* code to match against $\forall$ also matched the implication. Since any proof of Q *v2* is as good as any other in this context, there was never any opportunity to use unification to determine exactly which proof is appropriate. We expect similar problems with any implications in arguments to *insterU*.

```
  Abort.
End t7.
```

Reset insterU.

We can redefine *insterU* to treat implications differently. In particular, we pattern-match on the type of the type T in $\forall$ x : ?T, ... If T has type Prop, then x's instantiation should be thought of as a proof. Thus, instead of picking a new unification variable for it, we instead apply a user-supplied tactic *tac*. It is important that we end this special Prop case with || fail 1, so that, if *tac* fails to prove T, we abort the instantiation, rather than continuing on to the default quantifier handling.

```
Ltac insterU tac H :=
  repeat match type of H with
```

```
    | ∀ x : ?T, _ ⇒
      match type of T with
        | Prop ⇒
          (let H' := fresh "H'" in
            assert (H' : T); [
              solve [ tac ]
              | generalize (H H'); clear H H'; intro H ])
          || fail 1
        | _ ⇒
          let x := fresh "x" in
            evar (x : T);
            let x' := eval cbv delta [x] in x in
              clear x; generalize (H x'); clear H; intro H
      end
  end.

Ltac insterKeep tac H :=
  let H' := fresh "H'" in
    generalize H; intro H'; insterU tac H'.

Section t7.
  Variables A B : Type.
  Variable Q : A → Prop.
  Variable P : A → B → Prop.
  Variable f : A → A → A.
  Variable g : B → B → B.

  Hypothesis H1 : ∀ v, Q v → ∃ u, P v u.
  Hypothesis H2 : ∀ v1 u1 v2 u2,
    P v1 u1
    → P v2 u2
    → P (f v1 v2) (g u1 u2).

  Theorem t6 : ∀ v1 v2, Q v1 → Q v2 → ∃ u1, ∃ u2, P (f v1 v2) (g u1 u2).
```

We can prove the goal by calling *insterKeep* with a tactic that tries to find and apply a Q hypothesis over a variable about which we do not yet know any P facts. We need to begin this tactic code with *idtac*; to get around a strange limitation in Coq's proof engine, where a first-class tactic argument may not begin with a `match`.

```
    intros; do 2 insterKeep ltac:(idtac; match goal with
                                           | [ H : Q ?v ⊢ _ ] ⇒
                                             match goal with
                                               | [ _ : context[P v _] ⊢ _ ] ⇒ fail 1
                                               | _ ⇒ apply H
                                             end
                                         end) H1;
```

```
    repeat match goal with
             | [ H : ex _ ⊢ _ ] ⇒ destruct H
           end; eauto.
  Qed.
End t7.
```

It is often useful to instantiate existential variables explicitly. A built-in tactic provides one way of doing so.

```
Theorem t8 : ∃ p : nat * nat, fst p = 3.
  econstructor; instantiate (1 := (3, 2)); reflexivity.
Qed.
```

The 1 above is identifying an existential variable appearing in the current goal, with the last existential appearing assigned number 1, the second last assigned number 2, and so on. The named existential is replaced everywhere by the term to the right of the :=.

The *instantiate* tactic can be convenient for exploratory proving, but it leads to very brittle proof scripts that are unlikely to adapt to changing theorem statements. It is often more helpful to have a tactic that can be used to assign a value to a term that is known to be an existential. By employing a roundabout implementation technique, we can build a tactic that generalizes this functionality. In particular, our tactic *equate* will assert that two terms are equal. If one of the terms happens to be an existential, then it will be replaced everywhere with the other term.

```
Ltac equate x y :=
  let H := fresh "H" in
    assert (H : x = y); [ reflexivity | clear H ].
```

equate fails if it is not possible to prove $x = y$ by `reflexivity`. We perform the proof only for its unification side effects, clearing the fact $x = y$ afterward. With *equate*, we can build a less brittle version of the prior example.

```
Theorem t9 : ∃ p : nat * nat, fst p = 3.
  econstructor; match goal with
                  | [ ⊢ fst ?x = 3 ] ⇒ equate x (3, 2)
                end; reflexivity.
Qed.
```

Chapter 13

Proof by Reflection

The last chapter highlighted a very heuristic approach to proving. In this chapter, we will study an alternative technique, *proof by reflection*. We will write, in Gallina, decision procedures with proofs of correctness, and we will appeal to these procedures in writing very short proofs. Such a proof is checked by running the decision procedure. The term *reflection* applies because we will need to translate Gallina propositions into values of inductive types representing syntax, so that Gallina programs may analyze them.

13.1 Proving Evenness

Proving that particular natural number constants are even is certainly something we would rather have happen automatically. The Ltac-programming techniques that we learned in the last chapter make it easy to implement such a procedure.

```
Inductive isEven : nat → Prop :=
| Even_O : isEven O
| Even_SS : ∀ n, isEven n → isEven (S (S n)).

Ltac prove_even := repeat constructor.

Theorem even_256 : isEven 256.
  prove_even.
Qed.

Print even_256.

even_256 =
Even_SS
  (Even_SS
     (Even_SS
        (Even_SS
```

...and so on. This procedure always works (at least on machines with infinite resources), but it has a serious drawback, which we see when we print the proof it generates that 256

is even. The final proof term has length linear in the input value. This seems like a shame, since we could write a trivial and trustworthy program to verify evenness of constants. The proof checker could simply call our program where needed.

It is also unfortunate not to have static typing guarantees that our tactic always behaves appropriately. Other invocations of similar tactics might fail with dynamic type errors, and we would not know about the bugs behind these errors until we happened to attempt to prove complex enough goals.

The techniques of proof by reflection address both complaints. We will be able to write proofs like this with constant size overhead beyond the size of the input, and we will do it with verified decision procedures written in Gallina.

For this example, we begin by using a type from the MoreSpecif module (included in the book source) to write a certified evenness checker.

```
Print partial.
Inductive partial (P : Prop) : Set := Proved : P → [P] | Uncertain : [P]
```

A *partial* P value is an optional proof of P. The notation $[P]$ stands for *partial* P.

```
Local Open Scope partial_scope.
```

We bring into scope some notations for the *partial* type. These overlap with some of the notations we have seen previously for specification types, so they were placed in a separate scope that needs separate opening.

```
Definition check_even (n : nat) : [isEven n].
  Hint Constructors isEven.

  refine (fix F (n : nat) : [isEven n] :=
    match n with
      | 0 ⇒ Yes
      | 1 ⇒ No
      | S (S n') ⇒ Reduce (F n')
    end); auto.
Defined.
```

We can use dependent pattern-matching to write a function that performs a surprising feat. When given a *partial* P, this function partialOut returns a proof of P if the *partial* value contains a proof, and it returns a (useless) proof of **True** otherwise. From the standpoint of ML and Haskell programming, it seems impossible to write such a type, but it is trivial with a `return` annotation.

```
Definition partialOut (P : Prop) (x : [P]) :=
  match x return (match x with
                    | Proved _ ⇒ P
                    | Uncertain ⇒ True
                  end) with
    | Proved pf ⇒ pf
```

```
    | Uncertain ⇒ I
  end.
```

It may seem strange to define a function like this. However, it turns out to be very useful in writing a reflective verison of our earlier *prove_even* tactic:

```
Ltac prove_even_reflective :=
  match goal with
    | [ ⊢ isEven ?N] ⇒ exact (partialOut (check_even N))
  end.
```

We identify which natural number we are considering, and we "prove" its evenness by pulling the proof out of the appropriate check_even call.

```
Theorem even_256' : isEven 256.
  prove_even_reflective.
Qed.

Print even_256'.

even_256' = partialOut (check_even 256)
     : isEven 256
```

We can see a constant wrapper around the object of the proof. For any even number, this form of proof will suffice. What happens if we try the tactic with an odd number?

```
Theorem even_255 : isEven 255.

  prove_even_reflective.

User error: No matching clauses for match goal
```

Thankfully, the tactic fails. To see more precisely what goes wrong, we can run manually the body of the `match`.

```
  exact (partialOut (check_even 255)).

  Error: The term "partialOut (check_even 255)" has type
 "match check_even 255 with
  | Yes ⇒ isEven 255
  | No ⇒ True
  end" while it is expected to have type "isEven 255"
```

As usual, the type-checker performs no reductions to simplify error messages. If we reduced the first term ourselves, we would see that check_even 255 reduces to a *No*, so that the first term is equivalent to **True**, which certainly does not unify with **isEven** 255.

```
Abort.
```

13.2 Reflecting the Syntax of a Trivial Tautology Language

We might also like to have reflective proofs of trivial tautologies like this one:

```
Theorem true_galore : (True ∧ True) → (True ∨ (True ∧ (True → True))).
  tauto.
Qed.

Print true_galore.

true_galore =
fun H : True ∧ True ⇒
and_ind (fun _ _ : True ⇒ or_introl (True ∧ (True → True)) I) H
     : True ∧ True → True ∨ True ∧ (True → True)
```

As we might expect, the proof that `tauto` builds contains explicit applications of natural deduction rules. For large formulas, this can add a linear amount of proof size overhead, beyond the size of the input.

To write a reflective procedure for this class of goals, we will need to get into the actual "reflection" part of "proof by reflection." It is impossible to case-analyze a `Prop` in any way in Gallina. We must *reflect* `Prop` into some type that we *can* analyze. This inductive type is a good candidate:

```
Inductive taut : Set :=
| TautTrue : taut
| TautAnd : taut → taut → taut
| TautOr : taut → taut → taut
| TautImp : taut → taut → taut.
```

We write a recursive function to "unreflect" this syntax back to `Prop`.

```
Fixpoint tautDenote (t : taut) : Prop :=
  match t with
    | TautTrue ⇒ True
    | TautAnd t1 t2 ⇒ tautDenote t1 ∧ tautDenote t2
    | TautOr t1 t2 ⇒ tautDenote t1 ∨ tautDenote t2
    | TautImp t1 t2 ⇒ tautDenote t1 → tautDenote t2
  end.
```

It is easy to prove that every formula in the range of tautDenote is true.

```
Theorem tautTrue : ∀ t, tautDenote t.
  induction t; crush.
Qed.
```

To use tautTrue to prove particular formulas, we need to implement the syntax reflection process. A recursive Ltac function does the job.

```
Ltac tautReflect P :=
  match P with
    | True ⇒ TautTrue
    | ?P1 ∧ ?P2 ⇒
      let t1 := tautReflect P1 in
      let t2 := tautReflect P2 in
        constr:(TautAnd t1 t2)
    | ?P1 ∨ ?P2 ⇒
      let t1 := tautReflect P1 in
      let t2 := tautReflect P2 in
        constr:(TautOr t1 t2)
    | ?P1 → ?P2 ⇒
      let t1 := tautReflect P1 in
      let t2 := tautReflect P2 in
        constr:(TautImp t1 t2)
  end.
```

With *tautReflect* available, it is easy to finish our reflective tactic. We look at the goal formula, reflect it, and apply tautTrue to the reflected formula.

```
Ltac obvious :=
  match goal with
    | [ ⊢ ?P ] ⇒
      let t := tautReflect P in
        exact (tautTrue t)
  end.
```

We can verify that obvious solves our original example, with a proof term that does not mention details of the proof.

```
Theorem true_galore' : (True ∧ True) → (True ∨ (True ∧ (True → True))).
  obvious.
Qed.

Print true_galore'.

true_galore' =
tautTrue
  (TautImp (TautAnd TautTrue TautTrue)
     (TautOr TautTrue (TautAnd TautTrue (TautImp TautTrue TautTrue))))
     : True ∧ True → True ∨ True ∧ (True → True)
```

It is worth considering how the reflective tactic improves on a pure-Ltac implementation. The formula reflection process is just as ad-hoc as before, so we gain little there. In general, proofs will be more complicated than formula translation, and the "generic proof rule" that we apply here *is* on much better formal footing than a recursive Ltac function. The dependent type of the proof guarantees that it "works" on any input formula. This is all in

addition to the proof-size improvement that we have already seen.

13.3 A Monoid Expression Simplifier

Proof by reflection does not require encoding of all of the syntax in a goal. We can insert "variables" in our syntax types to allow injection of arbitrary pieces, even if we cannot apply specialized reasoning to them. In this section, we explore that possibility by writing a tactic for normalizing monoid equations.

```
Section monoid.
  Variable A : Set.
  Variable e : A.
  Variable f : A → A → A.

  Infix "+" := f.

  Hypothesis assoc : ∀ a b c, (a + b) + c = a + (b + c).
  Hypothesis identl : ∀ a, e + a = a.
  Hypothesis identr : ∀ a, a + e = a.
```

We add variables and hypotheses characterizing an arbitrary instance of the algebraic structure of monoids. We have an associative binary operator and an identity element for it.

It is easy to define an expression tree type for monoid expressions. A Var constructor is a "catch-all" case for subexpressions that we cannot model. These subexpressions could be actual Gallina variables, or they could just use functions that our tactic is unable to understand.

```
  Inductive mexp : Set :=
  | Ident : mexp
  | Var : A → mexp
  | Op : mexp → mexp → mexp.
```

Next, we write an "un-reflect" function.

```
  Fixpoint mdenote (me : mexp) : A :=
    match me with
      | Ident ⇒ e
      | Var v ⇒ v
      | Op me1 me2 ⇒ mdenote me1 + mdenote me2
    end.
```

We will normalize expressions by flattening them into lists, via associativity, so it is helpful to have a denotation function for lists of monoid values.

```
  Fixpoint mldenote (ls : list A) : A :=
    match ls with
      | nil ⇒ e
      | x :: ls' ⇒ x + mldenote ls'
```

```
  end.
```

The flattening function itself is easy to implement.

```
Fixpoint flatten (me : mexp) : list A :=
  match me with
    | Ident ⇒ nil
    | Var x ⇒ x :: nil
    | Op me1 me2 ⇒ flatten me1 ++ flatten me2
  end.
```

flatten has a straightforward correctness proof in terms of our denote functions.

```
Lemma flatten_correct' : ∀ ml2 ml1,
  mldenote ml1 + mldenote ml2 = mldenote (ml1 ++ ml2).
  induction ml1; crush.
Qed.

Theorem flatten_correct : ∀ me, mdenote me = mldenote (flatten me).
  Hint Resolve flatten_correct'.

  induction me; crush.
Qed.
```

Now it is easy to prove a theorem that will be the main tool behind our simplification tactic.

```
Theorem monoid_reflect : ∀ me1 me2,
  mldenote (flatten me1) = mldenote (flatten me2)
  → mdenote me1 = mdenote me2.
  intros; repeat rewrite flatten_correct; assumption.
Qed.
```

We implement reflection into the **mexp** type.

```
Ltac reflect me :=
  match me with
    | e ⇒ Ident
    | ?me1 + ?me2 ⇒
      let r1 := reflect me1 in
      let r2 := reflect me2 in
        constr:(Op r1 r2)
    | _ ⇒ constr:(Var me)
  end.
```

The final monoid tactic works on goals that equate two monoid terms. We reflect each and change the goal to refer to the reflected versions, finishing off by applying monoid_reflect and simplifying uses of mldenote.

```
Ltac monoid :=
  match goal with
```

```
    | [ ⊢ ?me1 = ?me2 ] ⇒
      let r1 := reflect me1 in
      let r2 := reflect me2 in
        change (mdenote r1 = mdenote r2);
          apply monoid_reflect; simpl mldenote
  end.
```

We can make short work of theorems like this one:

```
Theorem t1 : ∀ a b c d, a + b + c + d = a + (b + c) + d.
  intros; monoid.

============================
 a + (b + (c + (d + e))) = a + (b + (c + (d + e)))
```

monoid has canonicalized both sides of the equality, such that we can finish the proof by reflexivity.

```
  reflexivity.
Qed.
```

It is interesting to look at the form of the proof.

```
Print t1.
```

```
t1 =
fun a b c d : A ⇒
monoid_reflect (Op (Op (Op (Var a) (Var b)) (Var c)) (Var d))
  (Op (Op (Var a) (Op (Var b) (Var c))) (Var d))
  (refl_equal (a + (b + (c + (d + e)))))
     : ∀ a b c d : A, a + b + c + d = a + (b + c) + d
```

The proof term contains only restatements of the equality operands in reflected form, followed by a use of reflexivity on the shared canonical form.

```
End monoid.
```

Extensions of this basic approach are used in the implementations of the `ring` and *field* tactics that come packaged with Coq.

13.4 A Smarter Tautology Solver

Now we are ready to revisit our earlier tautology solver example. We want to broaden the scope of the tactic to include formulas whose truth is not syntactically apparent. We will want to allow injection of arbitrary formulas, like we allowed arbitrary monoid expressions in the last example. Since we are working in a richer theory, it is important to be able to use equalities between different injected formulas. For instance, we cannot prove $P \rightarrow P$ by

translating the formula into a value like Imp (Var P) (Var P), because a Gallina function has no way of comparing the two Ps for equality.

To arrive at a nice implementation satisfying these criteria, we introduce the *quote* tactic and its associated library.

```
Require Import Quote.

Inductive formula : Set :=
| Atomic : index → formula
| Truth : formula
| Falsehood : formula
| And : formula → formula → formula
| Or : formula → formula → formula
| Imp : formula → formula → formula.
```

The type **index** comes from the *Quote* library and represents a countable variable type. The rest of **formula**'s definition should be old hat by now.

The *quote* tactic will implement injection from `Prop` into **formula** for us, but it is not quite as smart as we might like. In particular, it interprets implications incorrectly, so we will need to declare a wrapper definition for implication, as we did in the last chapter.

```
Definition imp (P1 P2 : Prop) := P1 → P2.
Infix "->" := imp (no associativity, at level 95).
```

Now we can define our denotation function.

```
Definition asgn := varmap Prop.

Fixpoint formulaDenote (atomics : asgn) (f : formula) : Prop :=
  match f with
    | Atomic v ⇒ varmap_find False v atomics
    | Truth ⇒ True
    | Falsehood ⇒ False
    | And f1 f2 ⇒ formulaDenote atomics f1 ∧ formulaDenote atomics f2
    | Or f1 f2 ⇒ formulaDenote atomics f1 ∨ formulaDenote atomics f2
    | Imp f1 f2 ⇒ formulaDenote atomics f1 -> formulaDenote atomics f2
  end.
```

The **varmap** type family implements maps from **index** values. In this case, we define an assignment as a map from variables to `Prop`s. formulaDenote works with an assignment, and we use the varmap_find function to consult the assignment in the Atomic case. The first argument to varmap_find is a default value, in case the variable is not found.

```
Section my_tauto.
  Variable atomics : asgn.

  Definition holds (v : index) := varmap_find False v atomics.
```

We define some shorthand for a particular variable being true, and now we are ready to define some helpful functions based on the *ListSet* module of the standard library, which (unsurprisingly) presents a view of lists as sets.

```
Require Import ListSet.

Definition index_eq : ∀ x y : index, {x = y} + {x ≠ y}.
  decide equality.
Defined.

Definition add (s : set index) (v : index) := set_add index_eq v s.

Definition In_dec : ∀ v (s : set index), {In v s} + {¬ In v s}.
  Local Open Scope specif_scope.

  intro; refine (fix F (s : set index) : {In v s} + {¬ In v s} :=
    match s with
      | nil ⇒ No
      | v' :: s' ⇒ index_eq v' v || F s'
    end); crush.
Defined.
```

We define what it means for all members of an index set to represent true propositions, and we prove some lemmas about this notion.

```
Fixpoint allTrue (s : set index) : Prop :=
  match s with
    | nil ⇒ True
    | v :: s' ⇒ holds v ∧ allTrue s'
  end.

Theorem allTrue_add : ∀ v s,
  allTrue s
  → holds v
  → allTrue (add s v).
  induction s; crush;
    match goal with
      | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
    end; crush.
Qed.

Theorem allTrue_In : ∀ v s,
  allTrue s
  → set_In v s
  → varmap_find False v atomics.
  induction s; crush.
Qed.

Hint Resolve allTrue_add allTrue_In.

Local Open Scope partial_scope.
```

Now we can write a function forward which implements deconstruction of hypotheses. It has a dependent type, in the style of Chapter 6, guaranteeing correctness. The arguments

to forward are a goal formula f, a set *known* of atomic formulas that we may assume are true, a hypothesis formula *hyp*, and a success continuation *cont* that we call when we have extended *known* to hold new truths implied by *hyp*.

```
  Definition forward (f : formula) (known : set index) (hyp : formula)
    (cont : ∀ known', [allTrue known' → formulaDenote atomics f])
    : [allTrue known → formulaDenote atomics hyp → formulaDenote atomics f].
    refine (fix F (f : formula) (known : set index) (hyp : formula)
      (cont : ∀ known', [allTrue known' → formulaDenote atomics f])
      : [allTrue known → formulaDenote atomics hyp → formulaDenote atomics f] :=
      match hyp with
        | Atomic v ⇒ Reduce (cont (add known v))
        | Truth ⇒ Reduce (cont known)
        | Falsehood ⇒ Yes
        | And h1 h2 ⇒
          Reduce (F (Imp h2 f) known h1 (fun known' ⇒
            Reduce (F f known' h2 cont)))
        | Or h1 h2 ⇒ F f known h1 cont && F f known h2 cont
        | Imp _ _ ⇒ Reduce (cont known)
      end); crush.
  Defined.
```

A backward function implements analysis of the final goal. It calls forward to handle implications.

```
  Definition backward (known : set index) (f : formula)
    : [allTrue known → formulaDenote atomics f].
    refine (fix F (known : set index) (f : formula)
      : [allTrue known → formulaDenote atomics f] :=
      match f with
        | Atomic v ⇒ Reduce (In_dec v known)
        | Truth ⇒ Yes
        | Falsehood ⇒ No
        | And f1 f2 ⇒ F known f1 && F known f2
        | Or f1 f2 ⇒ F known f1 || F known f2
        | Imp f1 f2 ⇒ forward f2 known f1 (fun known' ⇒ F known' f2)
      end); crush; eauto.
  Defined.
```

A simple wrapper around backward gives us the usual type of a partial decision procedure.

```
  Definition my_tauto (f : formula) : [formulaDenote atomics f].
    intro; refine (Reduce (backward nil f)); crush.
  Defined.
End my_tauto.
```

Our final tactic implementation is now fairly straightforward. First, we intro all quanti-

fiers that do not bind Props. Then we call the *quote* tactic, which implements the reflection for us. Finally, we are able to construct an exact proof via partialOut and the my_tauto Gallina function.

```
Ltac my_tauto :=
  repeat match goal with
           | [ ⊢ ∀ x : ?P, _ ] ⇒
             match type of P with
               | Prop ⇒ fail 1
               | _ ⇒ intro
             end
         end;
  quote formulaDenote;
  match goal with
    | [ ⊢ formulaDenote ?m ?f ] ⇒ exact (partialOut (my_tauto m f))
  end.
```

A few examples demonstrate how the tactic works.

```
Theorem mt1 : True.
  my_tauto.
Qed.

Print mt1.

mt1 = partialOut (my_tauto (Empty_vm Prop) Truth)
      : True
```

We see my_tauto applied with an empty **varmap**, since every subformula is handled by formulaDenote.

```
Theorem mt2 : ∀ x y : nat, x = y -> x = y.
  my_tauto.
Qed.

Print mt2.

mt2 =
fun x y : nat ⇒
partialOut
  (my_tauto (Node_vm (x = y) (Empty_vm Prop) (Empty_vm Prop))
     (Imp (Atomic End_idx) (Atomic End_idx)))
     : ∀ x y : nat, x = y -> x = y
```

Crucially, both instances of $x = y$ are represented with the same index, *End_idx*. The value of this index only needs to appear once in the **varmap**, whose form reveals that **varmap**s are represented as binary trees, where **index** values denote paths from tree roots to leaves.

```
Theorem mt3 : ∀ x y z,
  (x < y ∧ y > z) ∨ (y > z ∧ x < S y)
  -> y > z ∧ (x < y ∨ x < S y).
  my_tauto.
Qed.

Print mt3.

fun x y z : nat ⇒
partialOut
  (my_tauto
     (Node_vm (x < S y) (Node_vm (x < y) (Empty_vm Prop) (Empty_vm Prop))
        (Node_vm (y > z) (Empty_vm Prop) (Empty_vm Prop)))
     (Imp
        (Or (And (Atomic (Left_idx End_idx)) (Atomic (Right_idx End_idx)))
           (And (Atomic (Right_idx End_idx)) (Atomic End_idx)))
        (And (Atomic (Right_idx End_idx))
           (Or (Atomic (Left_idx End_idx)) (Atomic End_idx)))))
     : ∀ x y z : nat,
       x < y ∧ y > z ∨ y > z ∧ x < S y -> y > z ∧ (x < y ∨ x < S y)
```

Our goal contained three distinct atomic formulas, and we see that a three-element **varmap** is generated.

It can be interesting to observe differences between the level of repetition in proof terms generated by my_tauto and `tauto` for especially trivial theorems.

```
Theorem mt4 : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False -> False.
  my_tauto.
Qed.

Print mt4.

mt4 =
partialOut
  (my_tauto (Empty_vm Prop)
     (Imp
        (And Truth
           (And Truth
              (And Truth (And Truth (And Truth (And Truth Falsehood))))))
        Falsehood))
     : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False -> False

Theorem mt4' : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False → False.
  tauto.
Qed.

Print mt4'.
```

```
mt4' =
fun H : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False ⇒
and_ind
  (fun (_ : True) (H1 : True ∧ True ∧ True ∧ True ∧ True ∧ False) ⇒
   and_ind
     (fun (_ : True) (H3 : True ∧ True ∧ True ∧ True ∧ False) ⇒
      and_ind
        (fun (_ : True) (H5 : True ∧ True ∧ True ∧ False) ⇒
         and_ind
           (fun (_ : True) (H7 : True ∧ True ∧ False) ⇒
            and_ind
              (fun (_ : True) (H9 : True ∧ False) ⇒
               and_ind (fun (_ : True) (H11 : False) ⇒ False_ind False H11)
                 H9) H7) H5) H3) H1) H
     : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False → False
```

13.5 Exercises

1. Implement a reflective procedure for normalizing systems of linear equations over rational numbers. In particular, the tactic should identify all hypotheses that are linear equations over rationals where the equation righthand sides are constants. It should normalize each hypothesis to have a lefthand side that is a sum of products of constants and variables, with no variable appearing multiple times. Then, your tactic should add together all of these equations to form a single new equation, possibly clearing the original equations. Some coefficients may cancel in the addition, reducing the number of variables that appear.

   To work with rational numbers, import module *QArith* and use `Local Open Scope` *Q_scope*. All of the usual arithmetic operator notations will then work with rationals, and there are shorthands for constants 0 and 1. Other rationals must be written as *num* # *den* for numerator *num* and denominator *den*. Use the infix operator == in place of =, to deal with different ways of expressing the same number as a fraction. For instance, a theorem and proof like this one should work with your tactic:

   ```
   Theorem t2 : ∀ x y z, (2 # 1) * (x - (3 # 2) * y) == 15 # 1
     → z + (8 # 1) * x == 20 # 1
     → (-6 # 2) * y + (10 # 1) * x + z == 35 # 1.
     intros; reflectContext; assumption.
   Qed.
   ```

   Your solution can work in any way that involves reflecting syntax and doing most calculation with a Gallina function. These hints outline a particular possible solution.

Throughout, the `ring` tactic will be helpful for proving many simple facts about rationals, and tactics like `rewrite` are correctly overloaded to work with rational equality ==.

(a) Define an inductive type exp of expressions over rationals (which inhabit the Coq type Q). Include variables (represented as natural numbers), constants, addition, subtraction, and multiplication.

(b) Define a function lookup for reading an element out of a list of rationals, by its position in the list.

(c) Define a function expDenote that translates exps, along with lists of rationals representing variable values, to Q.

(d) Define a recursive function *eqsDenote* over **list** (exp * Q), characterizing when all of the equations are true.

(e) Fix a representation *lhs* of flattened expressions. Where *len* is the number of variables, represent a flattened equation as **ilist** Q *len*. Each position of the list gives the coefficient of the corresponding variable.

(f) Write a recursive function *linearize* that takes a constant k and an expression e and optionally returns an *lhs* equivalent to k * e. This function returns None when it discovers that the input expression is not linear. The parameter *len* of *lhs* should be a parameter of *linearize*, too. The functions *singleton*, *everywhere*, and *map2* from DepList will probably be helpful. It is also helpful to know that *Qplus* is the identifier for rational addition.

(g) Write a recursive function *linearizeEqs* : **list** (exp * Q) → **option** (*lhs* * Q). This function linearizes all of the equations in the list in turn, building up the sum of the equations. It returns None if the linearization of any constituent equation fails.

(h) Define a denotation function for *lhs*.

(i) Prove that, when exp linearization succeeds on constant k and expression e, the linearized version has the same meaning as k * e.

(j) Prove that, when *linearizeEqs* succeeds on an equation list *eqs*, then the final summed-up equation is true whenever the original equation list is true.

(k) Write a tactic *findVarsHyps* to search through all equalities on rationals in the context, recursing through addition, subtraction, and multiplication to find the list of expressions that should be treated as variables. This list should be suitable as an argument to expDenote and *eqsDenote*, associating a Q value to each natural number that stands for a variable.

(l) Write a tactic *reflect* to reflect a Q expression into exp, with respect to a given list of variable values.

(m) Write a tactic *reflectEqs* to reflect a formula that begins with a sequence of implications from linear equalities whose lefthand sides are expressed with expDenote.

This tactic should build a **list** (exp * Q) representing the equations. Remember to give an explicit type annotation when returning a nil list, as in *constr*:(@nil (exp * Q)).

(n) Now this final tactic should do the job:

```
Ltac reflectContext :=
  let ls := findVarsHyps in
    repeat match goal with
             | [ H : ?e == ?num # ?den ⊢ _ ] ⇒
               let r := reflect ls e in
                 change (expDenote ls r == num # den) in H;
                 generalize H
           end;
    match goal with
      | [ ⊢ ?g ] ⇒ let re := reflectEqs g in
          intros;
            let H := fresh "H" in
            assert (H : eqsDenote ls re); [ simpl in *; tauto
              | repeat match goal with
                         | [ H : expDenote _ _ == _ ⊢ _ ] ⇒ clear H
                       end;
                generalize (linearizeEqsCorrect ls re H); clear H; simpl;
                  match goal with
                    | [ ⊢ ?X == ?Y → _ ] ⇒
                      ring_simplify X Y; intro
                  end ]
    end.
```

Chapter 14

Proving in the Large

It is somewhat unfortunate that the term "theorem-proving" looks so much like the word "theory." Most researchers and practitioners in software assume that mechanized theorem-proving is profoundly impractical. Indeed, until recently, most advances in theorem-proving for higher-order logics have been largely theoretical. However, starting around the beginning of the 21st century, there was a surge in the use of proof assistants in serious verification efforts. That line of work is still quite new, but I believe it is not too soon to distill some lessons on how to work effectively with large formal proofs.

Thus, this chapter gives some tips for structuring and maintaining large Coq developments.

14.1 Ltac Anti-Patterns

In this book, I have been following an unusual style, where proofs are not considered finished until they are "fully automated," in a certain sense. SEach such theorem is proved by a single tactic. Since Ltac is a Turing-complete programming language, it is not hard to squeeze arbitrary heuristics into single tactics, using operators like the semicolon to combine steps. In contrast, most Ltac proofs "in the wild" consist of many steps, performed by individual tactics followed by periods. Is it really worth drawing a distinction between proof steps terminated by semicolons and steps terminated by periods?

I argue that this is, in fact, a very important distinction, with serious consequences for a majority of important verification domains. The more uninteresting drudge work a proof domain involves, the more important it is to work to prove theorems with single tactics. From an automation standpoint, single-tactic proofs can be extremely effective, and automation becomes more and more critical as proofs are populated by more uninteresting detail. In this section, I will give some examples of the consequences of more common proof styles.

As a running example, consider a basic language of arithmetic expressions, an interpreter for it, and a transformation that scales up every constant in an expression.

```
Inductive exp : Set :=
```

```
| Const : nat → exp
| Plus : exp → exp → exp.

Fixpoint eval (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Plus e1 e2 ⇒ eval e1 + eval e2
  end.

Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (k * n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
  end.
```

We can write a very manual proof that *double* really doubles an expression's value.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k * eval e.
  induction e.

  trivial.

  simpl.
  rewrite IHe1.
  rewrite IHe2.
  rewrite mult_plus_distr_l.
  trivial.
Qed.
```

We use spaces to separate the two inductive cases. The second case mentions automatically-generated hypothesis names explicitly. As a result, innocuous changes to the theorem statement can invalidate the proof.

Reset eval_times.

```
Theorem eval_double : ∀ k x,
  eval (times k x) = k * eval x.
  induction x.

  trivial.

  simpl.


  rewrite IHe1.
```

Error: *The reference IHe1 was* not *found* in *the current environment.*

The inductive hypotheses are named *IHx1* and *IHx2* now, not *IHe1* and *IHe2*.

```
Abort.
```

We might decide to use a more explicit invocation of `induction` to give explicit binders for all of the names that we will reference later in the proof.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k * eval e.
  induction e as [ | ? IHe1 ? IHe2 ].

  trivial.

  simpl.
  rewrite IHe1.
  rewrite IHe2.
  rewrite mult_plus_distr_l.
  trivial.
Qed.
```

We pass `induction` an *intro pattern*, using a | character to separate out instructions for the different inductive cases. Within a case, we write ? to ask Coq to generate a name automatically, and we write an explicit name to assign that name to the corresponding new variable. It is apparent that, to use intro patterns to avoid proof brittleness, one needs to keep track of the seemingly unimportant facts of the orders in which variables are introduced. Thus, the script keeps working if we replace `e` by x, but it has become more cluttered. Arguably, neither proof is particularly easy to follow.

That category of complaint has to do with understanding proofs as static artifacts. As with programming in general, with serious projects, it tends to be much more important to be able to support evolution of proofs as specifications change. Unstructured proofs like the above examples can be very hard to update in concert with theorem statements. For instance, consider how the last proof script plays out when we modify times to introduce a bug.

Reset times.

```
Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (1 + k * n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
  end.

Theorem eval_times : ∀ k e,
  eval (times k e) = k * eval e.
  induction e as [ | ? IHe1 ? IHe2 ].

  trivial.

  simpl.


  rewrite IHe1.
```

Error: *The reference IHe1 was* not *found* `in` *the current environment.*

```
Abort.
```

Can you spot what went wrong, without stepping through the script step-by-step? The problem is that `trivial` never fails. Originally, `trivial` had been succeeding in proving an equality that follows by reflexivity. Our change to times leads to a case where that equality is no longer true. `trivial` happily leaves the false equality in place, and we continue on to the span of tactics intended for the second inductive case. Unfortunately, those tactics end up being applied to the *first* case instead.

The problem with `trivial` could be "solved" by writing *solve* [`trivial`] instead, so that an error is signaled early on if something unexpected happens. However, the root problem is that the syntax of a tactic invocation does not imply how many subgoals it produces. Much more confusing instances of this problem are possible. For example, if a lemma L is modified to take an extra hypothesis, then uses of `apply` L will general more subgoals than before. Old unstructured proof scripts will become hopelessly jumbled, with tactics applied to inappropriate subgoals. Because of the lack of structure, there is usually relatively little to be gleaned from knowledge of the precise point in a proof script where an error is raised.

Reset times.

```
Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (k * n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
  end.
```

Many real developments try to make essentially unstructured proofs look structured by applying careful indentation conventions, idempotent case-marker tactics included soley to serve as documentation, and so on. All of these strategies suffer from the same kind of failure of abstraction that was just demonstrated. I like to say that if you find yourself caring about indentation in a proof script, it is a sign that the script is structured poorly.

We can rewrite the current proof with a single tactic.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k * eval e.
  induction e as [ | ? IHe1 ? IHe2 ]; [
    trivial
    | simpl; rewrite IHe1; rewrite IHe2; rewrite mult_plus_distr_l; trivial ].
Qed.
```

This is an improvement in robustness of the script. We no longer need to worry about tactics from one case being applied to a different case. Still, the proof script is not especially readable. Probably most readers would not find it helpful in explaining why the theorem is true.

The situation gets worse in considering extensions to the theorem we want to prove. Let us add multiplication nodes to our exp type and see how the proof fares.

Reset exp.

```
Inductive exp : Set :=
| Const : nat → exp
| Plus : exp → exp → exp
| Mult : exp → exp → exp.

Fixpoint eval (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Plus e1 e2 ⇒ eval e1 + eval e2
    | Mult e1 e2 ⇒ eval e1 * eval e2
  end.

Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (k * n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
    | Mult e1 e2 ⇒ Mult (times k e1) e2
  end.

Theorem eval_times : ∀ k e,
  eval (times k e) = k * eval e.

  induction e as [ | ? IHe1 ? IHe2 ]; [
    trivial
    | simpl; rewrite IHe1; rewrite IHe2; rewrite mult_plus_distr_l; trivial ].
```

Error: *Expects a disjunctive* `pattern with` 3 *branches.*

```
Abort.
```

Unsurprisingly, the old proof fails, because it explicitly says that there are two inductive cases. To update the script, we must, at a minimum, remember the order in which the inductive cases are generated, so that we can insert the new case in the appropriate place. Even then, it will be painful to add the case, because we cannot walk through proof steps interactively when they occur inside an explicit set of cases.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k * eval e.
  induction e as [ | ? IHe1 ? IHe2 | ? IHe1 ? IHe2 ]; [
    trivial
    | simpl; rewrite IHe1; rewrite IHe2; rewrite mult_plus_distr_l; trivial
    | simpl; rewrite IHe1; rewrite mult_assoc; trivial ].
Qed.
```

Now we are in a position to see how much nicer is the style of proof that we have followed in most of this book.

Reset eval_times.

```
Hint Rewrite mult_plus_distr_l : cpdt.

Theorem eval_times : ∀ k e,
  eval (times k e) = k * eval e.
  induction e; crush.
Qed.
```

This style is motivated by a hard truth: one person's manual proof script is almost always mostly inscrutable to most everyone else. I claim that step-by-step formal proofs are a poor way of conveying information. Thus, we had might as well cut out the steps and automate as much as possible.

What about the illustrative value of proofs? Most informal proofs are read to convey the big ideas of proofs. How can reading `induction e; crush` convey any big ideas? My position is that any ideas that standard automation can find are not very big after all, and the *real* big ideas should be expressed through lemmas that are added as hints.

An example should help illustrate what I mean. Consider this function, which rewrites an expression using associativity of addition and multiplication.

```
Fixpoint reassoc (e : exp) : exp :=
  match e with
    | Const _ ⇒ e
    | Plus e1 e2 ⇒
      let e1' := reassoc e1 in
      let e2' := reassoc e2 in
        match e2' with
          | Plus e21 e22 ⇒ Plus (Plus e1' e21) e22
          | _ ⇒ Plus e1' e2'
        end
    | Mult e1 e2 ⇒
      let e1' := reassoc e1 in
      let e2' := reassoc e2 in
        match e2' with
          | Mult e21 e22 ⇒ Mult (Mult e1' e21) e22
          | _ ⇒ Mult e1' e2'
        end
  end.

Theorem reassoc_correct : ∀ e, eval (reassoc e) = eval e.
  induction e; crush;
    match goal with
      | [ ⊢ context[match ?E with Const _ ⇒ _ | Plus _ _ ⇒ _ | Mult _ _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
```

One subgoal remains:

IHe2 : **eval** *e3* * **eval** *e4* = **eval** *e2*

```
============================
 eval e1 * eval e3 * eval e4 = eval e1 * eval e2
```

crush does not know how to finish this goal. We could finish the proof manually.

```
rewrite ← IHe2; crush.
```

However, the proof would be easier to understand and maintain if we separated this insight into a separate lemma.

```
Abort.

Lemma rewr : ∀ a b c d, b * c = d → a * b * c = a * d.
  crush.
Qed.

Hint Resolve rewr.

Theorem reassoc_correct : ∀ e, eval (reassoc e) = eval e.
  induction e; crush;
    match goal with
      | [ ⊢ context[match ?E with Const _ ⇒ _ | Plus _ _ ⇒ _ | Mult _ _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
Qed.
```

In the limit, a complicated inductive proof might rely on one hint for each inductive case. The lemma for each hint could restate the associated case. Compared to manual proof scripts, we arrive at more readable results. Scripts no longer need to depend on the order in which cases are generated. The lemmas are easier to digest separately than are fragments of tactic code, since lemma statements include complete proof contexts. Such contexts can only be extracted from monolithic manual proofs by stepping through scripts interactively.

The more common situation is that a large induction has several easy cases that automation makes short work of. In the remaining cases, automation performs some standard simplification. Among these cases, some may require quite involved proofs; such a case may deserve a hint lemma of its own, where the lemma statement may copy the simplified version of the case. Alternatively, the proof script for the main theorem may be extended with some automation code targeted at the specific case. Even such targeted scripting is more desirable than manual proving, because it may be read and understood without knowledge of a proof's hierarchical structure, case ordering, or name binding structure.

14.2 Debugging and Maintaining Automation

Fully-automated proofs are desirable because they open up possibilities for automatic adaptation to changes of specification. A well-engineered script within a narrow domain can survive many changes to the formulation of the problem it solves. Still, as we are work-

ing with higher-order logic, most theorems fall within no obvious decidable theories. It is inevitable that most long-lived automated proofs will need updating.

Before we are ready to update our proofs, we need to write them in the first place. While fully-automated scripts are most robust to changes of specification, it is hard to write every new proof directly in that form. Instead, it is useful to begin a theorem with exploratory proving and then gradually refine it into a suitable automated form.

Consider this theorem from Chapter 7, which we begin by proving in a mostly manual way, invoking *crush* after each steop to discharge any low-hanging fruit. Our manual effort involves choosing which expressions to case-analyze on.

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

  dep_destruct (cfold e1); crush.
  dep_destruct (cfold e2); crush.

  dep_destruct (cfold e1); crush.
  dep_destruct (cfold e2); crush.

  dep_destruct (cfold e1); crush.
  dep_destruct (cfold e2); crush.

  dep_destruct (cfold e1); crush.
  dep_destruct (expDenote e1); crush.

  dep_destruct (cfold e); crush.

  dep_destruct (cfold e); crush.
Qed.
```

In this complete proof, it is hard to avoid noticing a pattern. We rework the proof, abstracting over the patterns we find.

```
Reset cfold_correct.

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

The expression we want to destruct here turns out to be the discriminee of a `match`, and we can easily enough write a tactic that destructs all such expressions.

```
  Ltac t :=
    repeat (match goal with
              | [ ⊢ context[match ?E with NConst _ ⇒ _ | Plus _ _ ⇒ _
                                        | Eq _ _ ⇒ _ | BConst _ ⇒ _ | And _ _ ⇒ _
                                        | If _ _ _ ⇒ _ | Pair _ _ _ _ ⇒ _
                                        | Fst _ _ ⇒ _ | Snd _ _ ⇒ _ end] ] ⇒
                dep_destruct E
            end; crush).

  t.
```

This tactic invocation discharges the whole case. It does the same on the next two cases, but it gets stuck on the fourth case.

```
  t.
  t.
  t.
```

The subgoal's conclusion is:

```
  ============================
  (if expDenote e1 then expDenote (cfold e2) else expDenote (cfold e3)) =
  expDenote (if expDenote e1 then cfold e2 else cfold e3)
```

We need to expand our t tactic to handle this case.

```
  Ltac t' :=
    repeat (match goal with
              | [ ⊢ context[match ?E with NConst _ ⇒ _ | Plus _ _ ⇒ _
                                       | Eq _ _ ⇒ _ | BConst _ ⇒ _ | And _ _ ⇒ _
                                       | If _ _ _ ⇒ _ | Pair _ _ _ _ ⇒ _
                                       | Fst _ _ _ ⇒ _ | Snd _ _ _ ⇒ _ end] ] ⇒
                dep_destruct E
              | [ ⊢ (if ?E then _ else _) = _ ] ⇒ destruct E
            end; crush).

  t'.
```

Now the goal is discharged, but t' has no effect on the next subgoal.

```
  t'.
```

A final revision of t finishes the proof.

```
  Ltac t'' :=
    repeat (match goal with
              | [ ⊢ context[match ?E with NConst _ ⇒ _ | Plus _ _ ⇒ _
                                       | Eq _ _ ⇒ _ | BConst _ ⇒ _ | And _ _ ⇒ _
                                       | If _ _ _ ⇒ _ | Pair _ _ _ _ ⇒ _
                                       | Fst _ _ _ ⇒ _ | Snd _ _ _ ⇒ _ end] ] ⇒
                dep_destruct E
              | [ ⊢ (if ?E then _ else _) = _ ] ⇒ destruct E
              | [ ⊢ context[match pairOut ?E with Some _ ⇒ _
                                       | None ⇒ _ end] ] ⇒
                dep_destruct E
            end; crush).

  t''.
  t''.
Qed.
```

We can take the final tactic and move it into the initial part of the proof script, arriving at a nicely-automated proof.

Reset t.

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

Even after we put together nice automated proofs, we must deal with specification changes that can invalidate them. It is not generally possible to step through single-tactic proofs interactively. There is a command *Debug On* that lets us step through points in tactic execution, but the debugger tends to make counterintuitive choices of which points we would like to stop at, and per-point output is quite verbose, so most Coq users do not find this debugging mode very helpful. How are we to understand what has broken in a script that used to work?

An example helps demonstrate a useful approach. Consider what would have happened in our proof of reassoc_correct if we had first added an unfortunate rewriting hint.

Reset reassoc_correct.

```
Theorem confounder : ∀ e1 e2 e3,
  eval e1 * eval e2 * eval e3 = eval e1 * (eval e2 + 1 - 1) * eval e3.
  crush.
Qed.

Hint Rewrite confounder : cpdt.

Theorem reassoc_correct : ∀ e, eval (reassoc e) = eval e.
  induction e; crush;
    match goal with
      | [ ⊢ context[match ?E with Const _ ⇒ _ | Plus _ _ ⇒ _ | Mult _ _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
```

One subgoal remains:

```
============================
 eval e1 * (eval e3 + 1 - 1) * eval e4 = eval e1 * eval e2
```

The poorly-chosen rewrite rule fired, changing the goal to a form where another hint no longer applies. Imagine that we are in the middle of a large development with many hints. How would we diagnose the problem? First, we might not be sure which case of the inductive proof has gone wrong. It is useful to separate out our automation procedure and apply it manually.

Restart.

```
Ltac t := crush; match goal with
                   | [ ⊢ context[match ?E with Const _ ⇒ _ | Plus _ _ ⇒ _
                                                          | Mult _ _ ⇒ _ end] ] ⇒
                     destruct E; crush
                 end.

induction e.
```

Since we see the subgoals before any simplification occurs, it is clear that this is the case for constants. *t* makes short work of it.

```
t.
```

The next subgoal, for addition, is also discharged without trouble.

```
t.
```

The final subgoal is for multiplication, and it is here that we get stuck in the proof state summarized above.

```
t.
```

What is *t* doing to get us to this point? The *info* command can help us answer this kind of question.

```
Undo.
info t.

== simpl in *; intuition; subst; autorewrite with cpdt in *;
   simpl in *; intuition; subst; autorewrite with cpdt in *;
   simpl in *; intuition; subst; destruct (reassoc e2).
   simpl in *; intuition.

   simpl in *; intuition.

   simpl in *; intuition; subst; autorewrite with cpdt in *;
      refine (eq_ind_r
                (fun n : nat ⇒
                 n * (eval e3 + 1 - 1) * eval e4 = eval e1 * eval e2) _ IHe1);
```

```
    autorewrite with cpdt in *; simpl in *; intuition;
  subst; autorewrite with cpdt in *; simpl in *;
  intuition; subst.
```

A detailed trace of t's execution appears. Since we are using the very general *crush* tactic, many of these steps have no effect and only occur as instances of a more general strategy. We can copy-and-paste the details to see where things go wrong.

```
Undo.
```

We arbitrarily split the script into chunks. The first few seem not to do any harm.

```
simpl in *; intuition; subst; autorewrite with cpdt in *.
simpl in *; intuition; subst; autorewrite with cpdt in *.
simpl in *; intuition; subst; destruct (reassoc e2).
simpl in *; intuition.
simpl in *; intuition.
```

The next step is revealed as the culprit, bringing us to the final unproved subgoal.

```
simpl in *; intuition; subst; autorewrite with cpdt in *.
```

We can split the steps further to assign blame.

```
Undo.
```

```
simpl in *.
intuition.
subst.
autorewrite with cpdt in *.
```

It was the final of these four tactics that made the rewrite. We can find out exactly what happened. The *info* command presents hierarchical views of proof steps, and we can zoom down to a lower level of detail by applying *info* to one of the steps that appeared in the original trace.

```
Undo.
```

```
info autorewrite with cpdt in *.
```

```
== refine (eq_ind_r (fun n : nat ⇒ n = eval e1 * eval e2) _
             (confounder (reassoc e1) e3 e4)).
```

The way a rewrite is displayed is somewhat baroque, but we can see that theorem confounder is the final culprit. At this point, we could remove that hint, prove an alternate version of the key lemma rewr, or come up with some other remedy. Fixing this kind of problem tends to be relatively easy once the problem is revealed.

```
Abort.
```

Sometimes a change to a development has undesirable performance consequences, even if

it does not prevent any old proof scripts from completing. If the performance consequences are severe enough, the proof scripts can be considered broken for practical purposes.

Here is one example of a performance surprise.

```
Section slow.
  Hint Resolve trans_eq.
```

The central element of the problem is the addition of transitivity as a hint. With transitivity available, it is easy for proof search to wind up exploring exponential search spaces. We also add a few other arbitrary variables and hypotheses, designed to lead to trouble later.

```
  Variable A : Set.
  Variables P Q R S : A → A → Prop.
  Variable f : A → A.

  Hypothesis H1 : ∀ x y, P x y → Q x y → R x y → f x = f y.
  Hypothesis H2 : ∀ x y, S x y → R x y.
```

We prove a simple lemma very quickly, using the *Time* command to measure exactly how quickly.

```
  Lemma slow : ∀ x y, P x y → Q x y → S x y → f x = f y.
    Time eauto 6.
```

Finished transaction in 0. *secs* (0.068004*u*,0.*s*)

```
  Qed.
```

Now we add a different hypothesis, which is innocent enough; in fact, it is even provable as a theorem.

```
  Hypothesis H3 : ∀ x y, x = y → f x = f y.

  Lemma slow' : ∀ x y, P x y → Q x y → S x y → f x = f y.
    Time eauto 6.
```

Finished transaction in 2. *secs* (1.264079*u*,0.*s*)

Why has the search time gone up so much? The *info* command is not much help, since it only shows the result of search, not all of the paths that turned out to be worthless.

```
    Restart.
    info eauto 6.

== intro x; intro y; intro H; intro H0; intro H4;
       simple eapply trans_eq.
    simple apply refl_equal.

    simple eapply trans_eq.
```

simple `apply refl_equal`.

simple `eapply trans_eq`.
simple `apply refl_equal`.

simple `apply` *H1*.
eexact H.

eexact H0.

simple `apply` *H2*; *eexact H4*.

This output does not tell us why proof search takes so long, but it does provide a clue that would be useful if we had forgotten that we added transitivity as a hint. The `eauto` tactic is applying depth-first search, and the proof script where the real action is ends up buried inside a chain of pointless invocations of transitivity, where each invocation uses reflexivity to discharge one subgoal. Each increment to the depth argument to `eauto` adds another silly use of transitivity. This wasted proof effort only adds linear time overhead, as long as proof search never makes false steps. No false steps were made before we added the new hypothesis, but somehow the addition made possible a new faulty path. To understand which paths we enabled, we can use the *debug* command.

Restart.
debug `eauto` 6.

The output is a large proof tree. The beginning of the tree is enough to reveal what is happening:

```
1 depth=6
1.1 depth=6 intro
1.1.1 depth=6 intro
1.1.1.1 depth=6 intro
1.1.1.1.1 depth=6 intro
1.1.1.1.1.1 depth=6 intro
1.1.1.1.1.1.1 depth=5 apply H3
1.1.1.1.1.1.1.1 depth=4 eapply trans_eq
1.1.1.1.1.1.1.1.1 depth=4 apply refl_equal
1.1.1.1.1.1.1.1.1.1 depth=3 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1 depth=3 apply refl_equal
1.1.1.1.1.1.1.1.1.1.1.1 depth=2 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1.1.1 depth=2 apply refl_equal
1.1.1.1.1.1.1.1.1.1.1.1.1.1 depth=1 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 depth=1 apply refl_equal
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 depth=0 eapply trans_eq
```

```
1.1.1.1.1.1.1.1.1.1.1.1.1.1.2 depth=1 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1 depth=0 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1.1.1.1.3 depth=0 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1.1.1.2 depth=2 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.2.1 depth=1 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1.1.2.1.1 depth=1 apply refl_equal
1.1.1.1.1.1.1.1.1.1.1.1.2.1.1.1 depth=0 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1.1.2.1.2 depth=1 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.2.1.2.1 depth=0 eapply trans_eq
1.1.1.1.1.1.1.1.1.1.1.1.2.1.3 depth=0 eapply trans_eq
```

The first choice `eauto` makes is to apply *H3*, since *H3* has the fewest hypotheses of all of the hypotheses and hints that match. However, it turns out that the single hypothesis generated is unprovable. That does not stop `eauto` from trying to prove it with an exponentially-sized tree of applications of transitivity, reflexivity, and symmetry of equality. It is the children of the initial `apply` *H3* that account for all of the noticeable time in proof execution. In a more realistic development, we might use this output of *info* to realize that adding transitivity as a hint was a bad idea.

```
  Qed.
End slow.
```

It is also easy to end up with a proof script that uses too much memory. As tactics run, they avoid generating proof terms, since serious proof search will consider many possible avenues, and we do not want to built proof terms for subproofs that end up unused. Instead, tactic execution maintains *thunks* (suspended computations, represented with closures), such that a tactic's proof-producing thunk is only executed when we run `Qed`. These thunks can use up large amounts of space, such that a proof script exhausts available memory, even when we know that we could have used much less memory by forcing some thunks earlier.

The *abstract* tactical helps us force thunks by proving some subgoals as their own lemmas. For instance, a proof `induction` x; *crush* can in many cases be made to use significantly less peak memory by changing it to `induction` x; *abstract crush*. The main limitation of *abstract* is that it can only be applied to subgoals that are proved completely, with no undetermined unification variables remaining. Still, many large automated proofs can realize vast memory savings via *abstract*.

14.3 Modules

Last chapter's examples of proof by reflection demonstrate opportunities for implementing abstract proof strategies with stronger formal guarantees than can be had with Ltac scripting. Coq's *module system* provides another tool for more rigorous development of generic theorems. This feature is inspired by the module systems found in Standard ML and Objective Caml, and the discussion that follows assumes familiarity with the basics of one of

those systems.

ML modules facilitate the grouping of abstract types with operations over those types. Moreover, there is support for *functors*, which are functions from modules to modules. A canonical example of a functor is one that builds a data structure implementation from a module that describes a domain of keys and its associated comparison operations.

When we add modules to a base language with dependent types, it becomes possible to use modules and functors to formalize kinds of reasoning that are common in algebra. For instance, this module signature captures the essence of the algebraic structure known as a group. A group consists of a carrier set G, an associative binary operation f, a left identity element e for f, and an operation i that is a left inverse for f.

```
Module Type GROUP.
  Parameter G : Set.
  Parameter f : G → G → G.
  Parameter e : G.
  Parameter i : G → G.

  Axiom assoc : ∀ a b c, f (f a b) c = f a (f b c).
  Axiom ident : ∀ a, f e a = a.
  Axiom inverse : ∀ a, f (i a) a = e.
End GROUP.
```

Many useful theorems hold of arbitrary groups. We capture some such theorem statements in another module signature.

```
Module Type GROUP_THEOREMS.
  Declare Module M : GROUP.

  Axiom ident' : ∀ a, M.f a M.e = a.

  Axiom inverse' : ∀ a, M.f a (M.i a) = M.e.

  Axiom unique_ident : ∀ e', (∀ a, M.f e' a = a) → e' = M.e.
End GROUP_THEOREMS.
```

We implement generic proofs of these theorems with a functor, whose input is an arbitrary group M. The proofs are completely manual, since it would take some effort to build suitable generic automation; rather, these theorems can serve as a basis for an automated procedure for simplifying group expressions, along the lines of the procedure for monoids from the last chapter. We take the proofs from the Wikipedia page on elementary group theory.

```
Module Group (M : GROUP) : GROUP_THEOREMS with Module M := M.
  Module M := M.

  Import M.

  Theorem inverse' : ∀ a, f a (i a) = e.
    intro.
    rewrite ← (ident (f a (i a))).
    rewrite ← (inverse (f a (i a))) at 1.
```

```
    rewrite assoc.
    rewrite assoc.
    rewrite ← (assoc (i a) a (i a)).
    rewrite inverse.
    rewrite ident.
    apply inverse.
  Qed.

  Theorem ident' : ∀ a, f a e = a.
    intro.
    rewrite ← (inverse a).
    rewrite ← assoc.
    rewrite inverse'.
    apply ident.
  Qed.

  Theorem unique_ident : ∀ e', (∀ a, M.f e' a = a) → e' = M.e.
    intros.
    rewrite ← (H e).
    symmetry.
    apply ident'.
  Qed.
End GROUP.
```

We can show that the integers with + form a group.

```
Require Import ZArith.
Open Scope Z_scope.

Module INT.
  Definition G := Z.
  Definition f x y := x + y.
  Definition e := 0.
  Definition i x := -x.

  Theorem assoc : ∀ a b c, f (f a b) c = f a (f b c).
    unfold f; crush.
  Qed.
  Theorem ident : ∀ a, f e a = a.
    unfold f, e; crush.
  Qed.
  Theorem inverse : ∀ a, f (i a) a = e.
    unfold f, i, e; crush.
  Qed.
End INT.
```

Next, we can produce integer-specific versions of the generic group theorems.

```
Module INTTHEOREMS := GROUP(INT).
```

```
Check IntTheorems.unique_ident.
```

$$IntTheorems.unique\_ident : \forall\ e' : Int.G, (\forall\ a : Int.G, Int.f\ e'\ a = a) \rightarrow e' = Int.e$$

```
Theorem unique_ident : ∀ e', (∀ a, e' + a = a) → e' = 0.
  exact IntTheorems.unique_ident.
Qed.
```

As in ML, the module system provides an effective way to structure large developments. Unlike in ML, Coq modules add no expressiveness; we can implement any module as an inhabitant of a dependent record type. It is the second-class nature of modules that makes them easier to use than dependent records in many case. Because modules may only be used in quite restricted ways, it is easier to support convenient module coding through special commands and editing modes, as the above example demonstrates. An isomorphic implementation with records would have suffered from lack of such conveniences as module subtyping and importation of the fields of a module.

14.4 Build Processes

As in software development, large Coq projects are much more manageable when split across multiple files and when decomposed into libraries. Coq and Proof General provide very good support for these activities.

Consider a library that we will name *Lib*, housed in directory `LIB` and split between files `A.v`, `B.v`, and `C.v`. A simple Makefile will compile the library, relying on the standard Coq tool `coq_makefile` to do the hard work.

```
MODULES := A B C
VS      := $(MODULES:%=%.v)

.PHONY: coq clean

coq: Makefile.coq
        make -f Makefile.coq

Makefile.coq: Makefile $(VS)
        coq_makefile -R . Lib $(VS) -o Makefile.coq

clean:: Makefile.coq
        make -f Makefile.coq clean
        rm -f Makefile.coq
```

The Makefile begins by defining a variable `VS` holding the list of filenames to be included in the project. The primary target is `coq`, which depends on the construction of an auxiliary Makefile called `Makefile.coq`. Another rule explains how to build that file. We call

`coq_makefile`, using the `-R` flag to specify that files in the current directory should be considered to belong to the library *Lib*. This Makefile will build a compiled version of each module, such that `X.v` is compiled into `X.vo`.

Now code in `B.v` may refer to definitions in `A.v` after running

`Require Import` *Lib.A*.

Library *Lib* is presented as a module, containing a submodule *A*, which contains the definitions from `A.v`. These are genuine modules in the sense of Coq's module system, and they may be passed to functors and so on.

`Require Import` is a convenient combination of two more primitive commands. `Require` finds the `.vo` file containing the named module, ensuring that the module is loaded into memory. `Import` loads all top-level definitions of the named module into the current namespace, and it may be used with local modules that do not have corresponding `.vo` files. Another command, `Load`, is for inserting the contents of a named file verbatim. It is generally better to use the module-based commands, since they avoid rerunning proof scripts, and they facilitate reorganization of directory structure without the need to change code.

Now we would like to use our library from a different development, called *Client* and found in directory `CLIENT`, which has its own Makefile.

```
MODULES := D E
VS      := $(MODULES:%=%.v)

.PHONY: coq clean

coq: Makefile.coq
        make -f Makefile.coq

Makefile.coq: Makefile $(VS)
        coq_makefile -R LIB Lib -R . Client $(VS) -o Makefile.coq

clean:: Makefile.coq
        make -f Makefile.coq clean
        rm -f Makefile.coq
```

We change the `coq_makefile` call to indicate where the library *Lib* is found. Now `D.v` and `E.v` can refer to definitions from *Lib* module *A* after running

`Require Import` *Lib.A*.

and `E.v` can refer to definitions from `D.v` by running

`Require Import` *Client.D*.

It can be useful to split a library into several files, but it is also inconvenient for client code to import library modules individually. We can get the best of both worlds by, for example, adding an extra source file `Lib.v` to *Lib*'s directory and Makefile.

`Require Export` *Lib.A Lib.B Lib.C*.

Now client code can import all definitions from all of *Lib*'s modules simply by running

`Require Import` *Lib*.

The two Makefiles above share a lot of code, so, in practice, it is useful to define a common Makefile that is included by multiple library-specific Makefiles.

The remaining ingredient is the proper way of editing library code files in Proof General. Recall this snippet of `.emacs` code from Chapter 2, which tells Proof General where to find the library associated with this book.

```
(custom-set-variables
  ...
  '(coq-prog-args '("-I" "/path/to/cpdt/src"))
  ...
)
```

To do interactive editing of our current example, we just need to change the flags to point to the right places.

```
(custom-set-variables
  ...
; '(coq-prog-args '("-I" "/path/to/cpdt/src"))
  '(coq-prog-args '("-R" "LIB" "Lib" "-R" "CLIENT" "Client"))
  ...
)
```

When working on multiple projects, it is useful to leave multiple versions of this setting in your `.emacs` file, commenting out all but one of them at any moment in time. To switch between projects, change the commenting structure and restart Emacs.

Part IV

Formalizing Programming Languages and Compilers

Chapter 15

First-Order Abstract Syntax

Many people interested in interactive theorem-proving want to prove theorems about programming languages. That domain also provides a good setting for demonstrating how to apply the ideas from the earlier parts of this book. This part introduces some techniques for encoding the syntax and semantics of programming languages, along with some example proofs designed to be as practical as possible, rather than to illustrate basic Coq technique.

To prove anything about a language, we must first formalize the language's syntax. We have a broad design space to choose from, and it makes sense to start with the simplest options, so-called *first-order* syntax encodings that do not use dependent types. These encodings are first-order because they do not use Coq function types in a critical way. In this chapter, we consider the most popular first-order encodings, using each to prove a basic type soundness theorem.

15.1 Concrete Binding

The most obvious encoding of the syntax of programming languages follows usual context-free grammars literally. We represent variables as strings and include a variable in our AST definition wherever a variable appears in the informal grammar. Concrete binding turns out to involve a surprisingly large amount of menial bookkeeping, especially when we encode higher-order languages with nested binder scopes. This section's example should give a flavor of what is required.

```
Module Concrete.
```

We need our variable type and its decidable equality operation.

```
  Definition var := string.
  Definition var_eq := string_dec.
```

We will formalize basic simply-typed lambda calculus. The syntax of expressions and types follows what we would write in a context-free grammar.

```
  Inductive exp : Set :=
```

```
| Const : bool → exp
| Var : var → exp
| App : exp → exp → exp
| Abs : var → exp → exp.

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

It is useful to define a syntax extension that lets us write function types in more standard notation.

```
Infix "->" := Arrow (right associativity, at level 60).
```

Now we turn to a typing judgment. We will need to define it in terms of typing contexts, which we represent as lists of pairs of variables and types.

```
Definition ctx := list (var × type).
```

The definitions of our judgments will be prettier if we write them using mixfix syntax. To define a judgment for looking up the type of a variable in a context, we first *reserve* a notation for the judgment. Reserved notations enable mutually-recursive definition of a judgment and its notation; in this sense, the reservation is like a forward declaration in C.

```
Reserved Notation "G |-v x : t" (no associativity, at level 90, x at next level).
```

Now we define the judgment itself, for variable typing, using a `where` clause to associate a notation definition.

```
Inductive lookup : ctx → var → type → Prop :=
| First : ∀ x t G,
  (x, t) :: G |-v x : t
| Next : ∀ x t x' t' G,
  x ≠ x'
  → G |-v x : t
  → (x', t') :: G |-v x : t

  where "G |-v x : t" := (lookup G x t).

Hint Constructors lookup.
```

The same technique applies to defining the main typing judgment. We use an `at next level` clause to cause the argument `e` of the notation to be parsed at a low enough precedence level.

```
Reserved Notation "G |-e e : t" (no associativity, at level 90, e at next level).

Inductive hasType : ctx → exp → type → Prop :=
| TConst : ∀ G b,
  G |-e Const b : Bool
| TVar : ∀ G v t,
  G |-v v : t
```

```
    → G |-e Var v : t
  | TApp : ∀ G e1 e2 dom ran,
    G |-e e1 : dom -> ran
    → G |-e e2 : dom
    → G |-e App e1 e2 : ran
  | TAbs : ∀ G x e' dom ran,
    (x, dom) :: G |-e e' : ran
    → G |-e Abs x e' : dom -> ran

    where "G |-e e : t" := (hasType G e t).

  Hint Constructors hasType.
```

It is useful to know that variable lookup results are unchanged by adding extra bindings to the end of a context.

```
  Lemma weaken_lookup : ∀ x t G' G1,
    G1 |-v x : t
    → G1 ++ G' |-v x : t.
    induction G1 as [ | [? ?] ? ]; crush;
      match goal with
        | [ H : _ |-v _ : _ ⊢ _ ] ⇒ inversion H; crush
      end.
  Qed.

  Hint Resolve weaken_lookup.
```

The same property extends to the full typing judgment.

```
  Theorem weaken_hasType' : ∀ G' G e t,
    G |-e e : t
    → G ++ G' |-e e : t.
    induction 1; crush; eauto.
  Qed.

  Theorem weaken_hasType : ∀ e t,
    nil |-e e : t
    → ∀ G', G' |-e e : t.
    intros; change G' with (nil ++ G');
      eapply weaken_hasType'; eauto.
  Qed.

  Hint Resolve weaken_hasType.
```

Much of the inconvenience of first-order encodings comes from the need to treat capture-avoiding substitution explicitly. We must start by defining a substitution function.

```
  Section subst.
    Variable x : var.
    Variable e1 : exp.
```

We are substituting expression *e1* for every free occurrence of x. Note that this definition is specialized to the case where *e1* is closed; substitution is substantially more complicated otherwise, potentially involving explicit alpha-variation. Luckily, our example of type safety for a call-by-value semantics only requires this restricted variety of substitution.

```
Fixpoint subst (e2 : exp) : exp :=
  match e2 with
    | Const _ ⇒ e2
    | Var x' ⇒ if var_eq x' x then e1 else e2
    | App e1 e2 ⇒ App (subst e1) (subst e2)
    | Abs x' e' ⇒ Abs x' (if var_eq x' x then e' else subst e')
  end.
```

We can prove a few theorems about substitution in well-typed terms, where we assume that *e1* is closed and has type *xt*.

```
Variable xt : type.
Hypothesis Ht' : nil |-e e1 : xt.
```

It is helpful to establish a notation asserting the freshness of a particular variable in a context.

```
Notation "x # G" := (∀ t' : type, In (x, t') G → False) (no associativity, at level 90).
```

To prove type preservation, we will need lemmas proving consequences of variable lookup proofs.

```
Lemma subst_lookup' : ∀ x' t,
  x ≠ x'
  → ∀ G1, G1 ++ (x, xt) :: nil |-v x' : t
    → G1 |-v x' : t.
  induction G1 as [ | [? ?] ? ]; crush;
    match goal with
      | [ H : _ |-v _ : _ ⊢ _ ] ⇒ inversion H
    end; crush.
Qed.

Hint Resolve subst_lookup'.

Lemma subst_lookup : ∀ x' t G1,
  x' # G1
  → G1 ++ (x, xt) :: nil |-v x' : t
  → t = xt.
  induction G1 as [ | [? ?] ? ]; crush; eauto;
    match goal with
      | [ H : _ |-v _ : _ ⊢ _ ] ⇒ inversion H
    end; crush; (elimtype False; eauto;
      match goal with
        | [ H : nil |-v _ : _ ⊢ _ ] ⇒ inversion H
```

```
    end)
  || match goal with
       | [ H : _ ⊢ _ ] ⇒ apply H; crush; eauto
     end.
Qed.

Implicit Arguments subst_lookup [x' t G1].
```

Another set of lemmas allows us to remove provably unused variables from the ends of typing contexts.

```
Lemma shadow_lookup : ∀ v t t' G1,
  G1 |-v x : t'
  → G1 ++ (x, xt) :: nil |-v v : t
  → G1 |-v v : t.
  induction G1 as [ | [? ?] ? ]; crush;
    match goal with
      | [ H : nil |-v _ : _ ⊢ _ ] ⇒ inversion H
      | [ H1 : _ |-v _ : _, H2 : _ |-v _ : _ ⊢ _ ] ⇒
        inversion H1; crush; inversion H2; crush
    end.
Qed.

Lemma shadow_hasType' : ∀ G e t,
  G |-e e : t
  → ∀ G1, G = G1 ++ (x, xt) :: nil
    → ∀ t", G1 |-v x : t"
      → G1 |-e e : t.
  Hint Resolve shadow_lookup.

  induction 1; crush; eauto;
    match goal with
      | [ H : (?x0, _) :: _ ++ (?x, _) :: _ |-e _ : _ ⊢ _ ] ⇒
        destruct (var_eq x0 x); subst; eauto
    end.
Qed.

Lemma shadow_hasType : ∀ G1 e t t",
  G1 ++ (x, xt) :: nil |-e e : t
  → G1 |-v x : t"
  → G1 |-e e : t.
  intros; eapply shadow_hasType'; eauto.
Qed.

Hint Resolve shadow_hasType.
```

Disjointness facts may be extended to larger contexts when the appropriate obligations are met.

```
  Lemma disjoint_cons : ∀ x x' t (G : ctx),
    x # G
    → x' ≠ x
    → x # (x', t) :: G.
    firstorder;
      match goal with
        | [ H : (_, _) = (_, _) ⊢ _ ] ⇒ injection H
      end; crush.
  Qed.

  Hint Resolve disjoint_cons.
```

Finally, we arrive at the main theorem about substitution: it preserves typing.

```
  Theorem subst_hasType : ∀ G e2 t,
    G |-e e2 : t
    → ∀ G1, G = G1 ++ (x, xt) :: nil
      → x # G1
      → G1 |-e subst e2 : t.
    induction 1; crush;
      try match goal with
            | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
          end; crush; eauto 6;
      match goal with
        | [ H1 : x # _, H2 : _ |-v x : _ ⊢ _ ] ⇒
          rewrite (subst_lookup H1 H2)
      end; crush.
  Qed.
```

We wrap the last theorem into an easier-to-apply form specialized to closed expressions.

```
  Theorem subst_hasType_closed : ∀ e2 t,
    (x, xt) :: nil |-e e2 : t
    → nil |-e subst e2 : t.
    intros; eapply subst_hasType; eauto.
  Qed.
End subst.

Hint Resolve subst_hasType_closed.
```

A notation for substitution will make the operational semantics easier to read.

```
Notation "[ x ~> e1 ] e2" := (subst x e1 e2) (no associativity, at level 80).
```

To define a call-by-value small-step semantics, we rely on a standard judgment characterizing which expressions are values.

```
Inductive val : exp → Prop :=
| VConst : ∀ b, val (Const b)
| VAbs : ∀ x e, val (Abs x e).
```

```
Hint Constructors val.
```

Now the step relation is easy to define.

```
Reserved Notation "e1 ==> e2" (no associativity, at level 90).

Inductive step : exp → exp → Prop :=
| Beta : ∀ x e1 e2,
  val e2
  → App (Abs x e1) e2 ==> [x ⇝ e2] e1
| Cong1 : ∀ e1 e2 e1',
  e1 ==> e1'
  → App e1 e2 ==> App e1' e2
| Cong2 : ∀ e1 e2 e2',
  val e1
  → e2 ==> e2'
  → App e1 e2 ==> App e1 e2'

  where "e1 ==> e2" := (step e1 e2).

Hint Constructors step.
```

The progress theorem says that any well-typed expression can take a step. To deal with limitations of the `induction` tactic, we put most of the proof in a lemma whose statement uses the usual trick of introducing extra equality hypotheses.

```
Lemma progress' : ∀ G e t, G |-e e : t
  → G = nil
  → val e ∨ ∃ e', e ==> e'.
  induction 1; crush; eauto;
    try match goal with
          | [ H : _ |-e _ : _ -> _ ⊢ _ ] ⇒ inversion H
        end;
    match goal with
      | [ H : _ ⊢ _ ] ⇒ solve [ inversion H; crush; eauto ]
    end.
Qed.

Theorem progress : ∀ e t, nil |-e e : t
  → val e ∨ ∃ e', e ==> e'.
  intros; eapply progress'; eauto.
Qed.
```

A similar pattern works for the preservation theorem, which says that any step of execution preserves an expression's type.

```
Lemma preservation' : ∀ G e t, G |-e e : t
  → G = nil
  → ∀ e', e ==> e'
```

```
    → nil |-e e' : t.
  induction 1; inversion 2; crush; eauto;
    match goal with
      | [ H : _ |-e Abs _ _ : _ ⊢ _ ] ⇒ inversion H
    end; eauto.
Qed.

Theorem preservation : ∀ e t, nil |-e e : t
  → ∀ e', e ==> e'
    → nil |-e e' : t.
  intros; eapply preservation'; eauto.
Qed.

End Concrete.
```

This was a relatively simple example, giving only a taste of the proof burden associated with concrete syntax. We were helped by the fact that, with call-by-value semantics, we only need to reason about substitution in closed expressions. There was also no need to alpha-vary an expression.

15.2 De Bruijn Indices

De Bruijn indices are much more popular than concrete syntax. This technique provides a *canonical* representation of syntax, where any two alpha-equivalent expressions have syntactically equal encodings, removing the need for explicit reasoning about alpha conversion. Variables are represented as natural numbers, where variable *n* denotes a reference to the *n*th closest enclosing binder. Because variable references in effect point to binders, there is no need to label binders, such as function abstraction, with variables.

```
Module DeBruijn.

  Definition var := nat.
  Definition var_eq := eq_nat_dec.

  Inductive exp : Set :=
  | Const : bool → exp
  | Var : var → exp
  | App : exp → exp → exp
  | Abs : exp → exp.

  Inductive type : Set :=
  | Bool : type
  | Arrow : type → type → type.

  Infix "->" := Arrow (right associativity, at level 60).
```

The definition of typing proceeds much the same as in the last section. Since variables are numbers, contexts can be simple lists of types. This makes it possible to write the lookup judgment without mentioning inequality of variables.

```
Definition ctx := list type.

Reserved Notation "G |-v x : t" (no associativity, at level 90, x at next level).

Inductive lookup : ctx → var → type → Prop :=
| First : ∀ t G,
  t :: G |-v O : t
| Next : ∀ x t t' G,
  G |-v x : t
  → t' :: G |-v S x : t

  where "G |-v x : t" := (lookup G x t).

Hint Constructors lookup.

Reserved Notation "G |-e e : t" (no associativity, at level 90, e at next level).

Inductive hasType : ctx → exp → type → Prop :=
| TConst : ∀ G b,
  G |-e Const b : Bool
| TVar : ∀ G v t,
  G |-v v : t
  → G |-e Var v : t
| TApp : ∀ G e1 e2 dom ran,
  G |-e e1 : dom -> ran
  → G |-e e2 : dom
  → G |-e App e1 e2 : ran
| TAbs : ∀ G e' dom ran,
  dom :: G |-e e' : ran
  → G |-e Abs e' : dom -> ran

  where "G |-e e : t" := (hasType G e t).
```

In the **hasType** case for function abstraction, there is no need to choose a variable name. We simply push the function domain type onto the context G.

```
Hint Constructors hasType.
```

We prove roughly the same weakening theorems as before.

```
Lemma weaken_lookup : ∀ G' v t G,
  G |-v v : t
  → G ++ G' |-v v : t.
  induction 1; crush.
Qed.

Hint Resolve weaken_lookup.

Theorem weaken_hasType' : ∀ G' G e t,
  G |-e e : t
```

```
    → G ++ G' |-e e : t.
    induction 1; crush; eauto.
  Qed.

  Theorem weaken_hasType : ∀ e t,
    nil |-e e : t
    → ∀ G', G' |-e e : t.
    intros; change G' with (nil ++ G');
      eapply weaken_hasType'; eauto.
  Qed.

  Hint Resolve weaken_hasType.

  Section subst.
    Variable e1 : exp.
```

Substitution is easier to define than with concrete syntax. While our old definition needed to use two comparisons for equality of variables, the de Bruijn substitution only needs one comparison.

```
    Fixpoint subst (x : var) (e2 : exp) : exp :=
      match e2 with
        | Const _ ⇒ e2
        | Var x' ⇒ if var_eq x' x then e1 else e2
        | App e1 e2 ⇒ App (subst x e1) (subst x e2)
        | Abs e' ⇒ Abs (subst (S x) e')
      end.

    Variable xt : type.
```

We prove similar theorems about inversion of variable lookup.

```
    Lemma subst_eq : ∀ t G1,
      G1 ++ xt :: nil |-v length G1 : t
      → t = xt.
      induction G1; inversion 1; crush.
    Qed.

    Implicit Arguments subst_eq [t G1].

    Lemma subst_eq' : ∀ t G1 x,
      G1 ++ xt :: nil |-v x : t
      → x ≠ length G1
      → G1 |-v x : t.
      induction G1; inversion 1; crush;
        match goal with
          | [ H : nil |-v _ : _ ⊢ _ ] ⇒ inversion H
        end.
    Qed.

    Hint Resolve subst_eq'.
```

```
  Lemma subst_neq : ∀ v t G1,
    G1 ++ xt :: nil |-v v : t
    → v ≠ length G1
    → G1 |-e Var v : t.
    induction G1; inversion 1; crush.
  Qed.

  Hint Resolve subst_neq.

  Hypothesis Ht' : nil |-e e1 : xt.
```

The next lemma is included solely to guide `eauto`, which will not apply computational equivalences automatically.

```
  Lemma hasType_push : ∀ dom G1 e' ran,
    dom :: G1 |-e subst (length (dom :: G1)) e' : ran
    → dom :: G1 |-e subst (S (length G1)) e' : ran.
    trivial.
  Qed.

  Hint Resolve hasType_push.
```

Finally, we are ready for the main theorem about substitution and typing.

```
  Theorem subst_hasType : ∀ G e2 t,
    G |-e e2 : t
    → ∀ G1, G = G1 ++ xt :: nil
      → G1 |-e subst (length G1) e2 : t.
    induction 1; crush;
      try match goal with
            | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
          end; crush; eauto 6;
      try match goal with
            | [ H : _ |-v _ : _ ⊢ _ ] ⇒
              rewrite (subst_eq H)
          end; crush.
  Qed.

  Theorem subst_hasType_closed : ∀ e2 t,
    xt :: nil |-e e2 : t
    → nil |-e subst O e2 : t.
    intros; change O with (length (@nil type)); eapply subst_hasType; eauto.
  Qed.
End subst.

Hint Resolve subst_hasType_closed.
```

We define the operational semantics much as before.

```
Notation "[ x ~> e1 ] e2" := (subst e1 x e2) (no associativity, at level 80).
```

```
Inductive val : exp → Prop :=
| VConst : ∀ b, val (Const b)
| VAbs : ∀ e, val (Abs e).

Hint Constructors val.

Reserved Notation "e1 ==> e2" (no associativity, at level 90).

Inductive step : exp → exp → Prop :=
| Beta : ∀ e1 e2,
  val e2
  → App (Abs e1) e2 ==> [0 ⇝ e2] e1
| Cong1 : ∀ e1 e2 e1',
  e1 ==> e1'
  → App e1 e2 ==> App e1' e2
| Cong2 : ∀ e1 e2 e2',
  val e1
  → e2 ==> e2'
  → App e1 e2 ==> App e1 e2'

  where "e1 ==> e2" := (step e1 e2).

Hint Constructors step.
```

Since we have added the right hints, the progress and preservation theorem statements and proofs are exactly the same as in the concrete encoding example.

```
Lemma progress' : ∀ G e t, G |-e e : t
  → G = nil
  → val e ∨ ∃ e', e ==> e'.
  induction 1; crush; eauto;
    try match goal with
          | [ H : _ |-e _ : _ -> _ ⊢ _ ] ⇒ inversion H
        end;
    repeat match goal with
             | [ H : _ ⊢ _ ] ⇒ solve [ inversion H; crush; eauto ]
           end.
Qed.

Theorem progress : ∀ e t, nil |-e e : t
  → val e ∨ ∃ e', e ==> e'.
  intros; eapply progress'; eauto.
Qed.

Lemma preservation' : ∀ G e t, G |-e e : t
  → G = nil
  → ∀ e', e ==> e'
    → nil |-e e' : t.
```

```
    induction 1; inversion 2; crush; eauto;
      match goal with
        | [ H : _ |-e Abs _ : _ ⊢ _ ] ⇒ inversion H
      end; eauto.
  Qed.

  Theorem preservation : ∀ e t, nil |-e e : t
    → ∀ e', e ==> e'
      → nil |-e e' : t.
    intros; eapply preservation'; eauto.
  Qed.

End DeBruijn.
```

15.3 Locally Nameless Syntax

The most popular Coq syntax encoding today is the *locally nameless* style, which has been around for a while but was popularized recently by Aydemir et al., following a methodology summarized in their paper "Engineering Formal Metatheory." A specialized tutorial by that group[1] explains the approach, based on a library. In this section, we will build up all of the necessary ingredients from scratch.

The one-sentence summary of locally nameless encoding is that we represent free variables as concrete syntax does, and we represent bound variables with de Bruijn indices. Many proofs involve reasoning about terms transplanted into different free variable contexts; concrete encoding of free variables means that, to perform such a transplanting, we need no fix-up operation to adjust de Bruijn indices. At the same time, use of de Bruijn indices for local variables gives us canonical representations of expressions, with respect to the usual convention of alpha-equivalence. This makes many operations, including substitution of open terms in open terms, easier to implement.

The "Engineering Formal Metatheory" methodology involves a number of subtle design decisions, which we will describe as they appear in the latest version of our running example.

```
Module LocallyNameless.

  Definition free_var := string.
  Definition bound_var := nat.

  Inductive exp : Set :=
  | Const : bool → exp
  | FreeVar : free_var → exp
  | BoundVar : bound_var → exp
  | App : exp → exp → exp
  | Abs : exp → exp.
```

[1] http://www.cis.upenn.edu/~plclub/oregon08/

Note the different constructors for free vs. bound variables, and note that the lack of a variable annotation on Abs nodes is inherited from the de Bruijn convention.

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

Infix "->" := Arrow (right associativity, at level 60).
```

As typing only depends on types of free variables, our contexts borrow their form from the concrete binding example.

```
Definition ctx := list (free_var × type).

Reserved Notation "G |-v x : t" (no associativity, at level 90, x at next level).

Inductive lookup : ctx → free_var → type → Prop :=
| First : ∀ x t G,
  (x, t) :: G |-v x : t
| Next : ∀ x t x' t' G,
  x ≠ x'
  → G |-v x : t
  → (x', t') :: G |-v x : t

  where "G |-v x : t" := (lookup G x t).

Hint Constructors lookup.
```

The first unusual operation we need is *opening*, where we replace a particular bound variable with a particular free variable. Whenever we "go under a binder," in the typing judgment or elsewhere, we choose a new free variable to replace the old bound variable of the binder. Opening implements the replacement of one by the other. It is like a specialized version of the substitution function we used for pure de Bruijn terms.

```
Section open.
  Variable x : free_var.

  Fixpoint open (n : bound_var) (e : exp) : exp :=
    match e with
      | Const _ ⇒ e
      | FreeVar _ ⇒ e
      | BoundVar n' ⇒
        if eq_nat_dec n' n
          then FreeVar x
          else if le_lt_dec n' n
            then e
            else BoundVar (pred n')
      | App e1 e2 ⇒ App (open n e1) (open n e2)
      | Abs e1 ⇒ Abs (open (S n) e1)
    end.
```

```
End open.
```

We will also need to reason about an expression's set of free variables. To keep things simple, we represent sets as lists that may contain duplicates. Note how much easier this operation is to implement than over pure de Bruijn terms, since we do not need to maintain a separate numeric argument that keeps track of how deeply we have descended into the input expression.

```
Fixpoint freeVars (e : exp) : list free_var :=
  match e with
    | Const _ ⇒ nil
    | FreeVar x ⇒ x :: nil
    | BoundVar _ ⇒ nil
    | App e1 e2 ⇒ freeVars e1 ++ freeVars e2
    | Abs e1 ⇒ freeVars e1
  end.
```

It will be useful to have a well-formedness judgment for our terms. This notion is called *local closure*. An expression may be declared to be closed, up to a particular maximum de Bruijn index.

```
Inductive lclosed : nat → exp → Prop :=
| CConst : ∀ n b, lclosed n (Const b)
| CFreeVar : ∀ n v, lclosed n (FreeVar v)
| CBoundVar : ∀ n v, v < n → lclosed n (BoundVar v)
| CApp : ∀ n e1 e2, lclosed n e1 → lclosed n e2 → lclosed n (App e1 e2)
| CAbs : ∀ n e1, lclosed (S n) e1 → lclosed n (Abs e1).

Hint Constructors lclosed.
```

Now we are ready to define the typing judgment.

```
Reserved Notation "G |-e e : t" (no associativity, at level 90, e at next level).

Inductive hasType : ctx → exp → type → Prop :=
| TConst : ∀ G b,
  G |-e Const b : Bool
| TFreeVar : ∀ G v t,
  G |-v v : t
  → G |-e FreeVar v : t
| TApp : ∀ G e1 e2 dom ran,
  G |-e e1 : dom -> ran
  → G |-e e2 : dom
  → G |-e App e1 e2 : ran
| TAbs : ∀ G e' dom ran L,
  (∀ x, ¬ In x L → (x, dom) :: G |-e open x O e' : ran)
  → G |-e Abs e' : dom -> ran
```

```
  where "G |-e e : t" := (hasType G e t).
```

Compared to the previous versions, only the TAbs rule is surprising. The rule uses *co-finite quantiifcation*. That is, the premise of the rule quantifies over all x values that are not members of a finite set L. A proof may choose any value of L when applying TAbs. An alternate, more intuitive version of the rule would fix L to be freeVars e'. It turns out that the greater flexibility of the rule above simplifies many proofs significantly. This typing judgment may be proved equivalent to the more intuitive version, though we will not carry out the proof here.

Specifially, what our version of TAbs says is that, to prove that Abs e' has a function type, we must prove that any opening of e' with a variable not in L has the proper type. For each x choice, we extend the context G in the usual way.

```
Hint Constructors hasType.
```

We prove a standard weakening theorem for typing, adopting a more general form than in the previous sections.

```
Lemma lookup_push : ∀ G G' x t x' t',
  (∀ x t, G |-v x : t → G' |-v x : t)
  → (x, t) :: G |-v x' : t'
  → (x, t) :: G' |-v x' : t'.
  inversion 2; crush.
Qed.

Hint Resolve lookup_push.

Theorem weaken_hasType : ∀ G e t,
  G |-e e : t
  → ∀ G', (∀ x t, G |-v x : t → G' |-v x : t)
    → G' |-e e : t.
  induction 1; crush; eauto.
Qed.

Hint Resolve weaken_hasType.
```

We define a simple extension of *crush* to apply in many of the lemmas that follow.

```
Ltac ln := crush;
  repeat (match goal with
            | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
            | [ _ : context[if ?E then _ else _] ⊢ _ ] ⇒ destruct E
          end; crush); eauto.
```

Two basic properties of local closure will be useful later.

```
Lemma lclosed_S : ∀ x e n,
  lclosed n (open x n e)
  → lclosed (S n) e.
  induction e; inversion 1; ln.
```

```
Qed.

Hint Resolve lclosed_S.

Lemma lclosed_weaken : ∀ n e,
  lclosed n e
  → ∀ n', n' ≥ n
    → lclosed n' e.
  induction 1; crush.
Qed.

Hint Resolve lclosed_weaken.
Hint Extern 1 (_ ≥ _) ⇒ omega.
```

To prove some further properties, we need the ability to choose a variable that is disjoint from a particular finite set. We implement a specific choice function `fresh`; its details do not matter, as all we need is the final theorem about it, freshOk. Concretely, to choose a variable disjoint from set L, we sum the lengths of the variable names in L and choose a new variable name that is one longer than that sum. This variable can be the string "x", followed by a number of primes equal to the sum.

```
Open Scope string_scope.

Fixpoint primes (n : nat) : string :=
  match n with
    | O ⇒ "x"
    | S n' ⇒ primes n' ++ "'"
  end.

Fixpoint sumLengths (L : list free_var) : nat :=
  match L with
    | nil ⇒ O
    | x :: L' ⇒ String.length x + sumLengths L'
  end.

Definition fresh (L : list free_var) := primes (sumLengths L).
```

A few lemmas suffice to establish the correctness theorem freshOk for `fresh`.

```
Theorem freshOk' : ∀ x L, String.length x > sumLengths L
  → ¬ In x L.
  induction L; crush.
Qed.

Lemma length_app : ∀ s2 s1,
  String.length (s1 ++ s2) = String.length s1 + String.length s2.
  induction s1; crush.
Qed.

Hint Rewrite length_app : cpdt.

Lemma length_primes : ∀ n, String.length (primes n) = S n.
```

```
  induction n; crush.
Qed.

Hint Rewrite length_primes : cpdt.

Theorem freshOk : ∀ L, ¬ In (fresh L) L.
  intros; apply freshOk'; unfold fresh; crush.
Qed.

Hint Resolve freshOk.
```

Now we can prove that well-typedness implies local closure. `fresh` will be used for us automatically by `eauto` in the Abs case, driven by the presence of freshOk as a hint.

```
Lemma hasType_lclosed : ∀ G e t,
  G |-e e : t
  → lclosed 0 e.
  induction 1; eauto.
Qed.
```

An important consequence of local closure is that certain openings are idempotent.

```
Lemma lclosed_open : ∀ n e, lclosed n e
  → ∀ x, open x n e = e.
  induction 1; ln.
Qed.

Hint Resolve lclosed_open hasType_lclosed.

Open Scope list_scope.
```

We are now almost ready to get down to the details of substitution. First, we prove six lemmas related to treating lists as sets.

```
Lemma In_cons1 : ∀ T (x x' : T) ls,
  x = x'
  → In x (x' :: ls).
  crush.
Qed.

Lemma In_cons2 : ∀ T (x x' : T) ls,
  In x ls
  → In x (x' :: ls).
  crush.
Qed.

Lemma In_app1 : ∀ T (x : T) ls2 ls1,
  In x ls1
  → In x (ls1 ++ ls2).
  induction ls1; crush.
Qed.

Lemma In_app2 : ∀ T (x : T) ls2 ls1,
```

```
    In x ls2
    → In x (ls1 ++ ls2).
    induction ls1; crush.
  Qed.

  Lemma freshOk_app1 : ∀ L1 L2,
    ¬ In (fresh (L1 ++ L2)) L1.
    intros; generalize (freshOk (L1 ++ L2)); crush.
  Qed.

  Lemma freshOk_app2 : ∀ L1 L2,
    ¬ In (fresh (L1 ++ L2)) L2.
    intros; generalize (freshOk (L1 ++ L2)); crush.
  Qed.

  Hint Resolve In_cons1 In_cons2 In_app1 In_app2.
```

Now we can define our simplest substitution function yet, thanks to the fact that we only subsitute for free variables, which are distinguished syntactically from bound variables.

```
  Section subst.
    Hint Resolve freshOk_app1 freshOk_app2.

    Variable x : free_var.
    Variable e1 : exp.

    Fixpoint subst (e2 : exp) : exp :=
      match e2 with
        | Const _ ⇒ e2
        | FreeVar x' ⇒ if string_dec x' x then e1 else e2
        | BoundVar _ ⇒ e2
        | App e1 e2 ⇒ App (subst e1) (subst e2)
        | Abs e' ⇒ Abs (subst e')
      end.

    Variable xt : type.
```

It comes in handy to define disjointness of a variable and a context differently than in previous examples. We use the standard list function map, as well as the function fst for projecting the first element of a pair. We write @fst _ _ rather than just fst to ask that fst's implicit arguments be instantiated with inferred values.

```
    Definition disj x (G : ctx) := In x (map (@fst _ _) G) → False.
    Infix "#" := disj (no associativity, at level 90).

    Ltac disj := crush;
      match goal with
        | [ _ : _ :: _ = ?G0 ++ _ ⊢ _ ] ⇒ destruct G0
      end; crush; eauto.
```

Some basic properties of variable lookup will be needed on the road to our usual theorem connecting substitution and typing.

```
Lemma lookup_disj' : ∀ t G1,
  G1 |-v x : t
  → ∀ G, x # G
    → G1 = G ++ (x, xt) :: nil
    → t = xt.
  unfold disj; induction 1; disj.
Qed.

Lemma lookup_disj : ∀ t G,
  x # G
  → G ++ (x, xt) :: nil |-v x : t
  → t = xt.
  intros; eapply lookup_disj'; eauto.
Qed.

Lemma lookup_ne' : ∀ G1 v t,
  G1 |-v v : t
  → ∀ G, G1 = G ++ (x, xt) :: nil
    → v ≠ x
    → G |-v v : t.
  induction 1; disj.
Qed.

Lemma lookup_ne : ∀ G v t,
  G ++ (x, xt) :: nil |-v v : t
  → v ≠ x
  → G |-v v : t.
  intros; eapply lookup_ne'; eauto.
Qed.

Hint Extern 1 (_ |-e _ : _) ⇒
  match goal with
    | [ H1 : _, H2 : _ ⊢ _ ] ⇒ rewrite (lookup_disj H1 H2)
  end.
Hint Resolve lookup_ne.

Hint Extern 1 (@eq exp _ _) ⇒ f_equal.
```

We need to know that substitution and opening commute under appropriate circumstances.

```
Lemma open_subst : ∀ x0 e' n,
  lclosed n e1
  → x ≠ x0
  → open x0 n (subst e') = subst (open x0 n e').
  induction e'; ln.
Qed.
```

We state a corollary of the last result which will work more smoothly with `eauto`.

```
Lemma hasType_open_subst : ∀ G x0 e t,
  G |-e subst (open x0 0 e) : t
  → x ≠ x0
  → lclosed 0 e1
  → G |-e open x0 0 (subst e) : t.
  intros; rewrite open_subst; eauto.
Qed.

Hint Resolve hasType_open_subst.
```

Another lemma establishes the validity of weakening variable lookup judgments with fresh variables.

```
Lemma disj_push : ∀ x0 (t : type) G,
  x # G
  → x ≠ x0
  → x # (x0, t) :: G.
  unfold disj; crush.
Qed.

Hint Resolve disj_push.

Lemma lookup_cons : ∀ x0 dom G x1 t,
  G |-v x1 : t
  → ¬ In x0 (map (@fst _ _) G)
  → (x0, dom) :: G |-v x1 : t.
  induction 1; crush;
    match goal with
      | [ H : _ |-v _ : _ ⊢ _ ] ⇒ inversion H
    end; crush.
Qed.

Hint Resolve lookup_cons.
Hint Unfold disj.
```

Finally, it is useful to state a version of the TAbs rule specialized to the choice of L that is useful in our main substitution proof.

```
Lemma TAbs_specialized : ∀ G e' dom ran L x1,
  (∀ x, ¬ In x (x1 :: L ++ map (@fst _ _) G) → (x, dom) :: G |-e open x 0 e' : ran)
  → G |-e Abs e' : dom -> ran.
  eauto.
Qed.
```

Now we can prove the main inductive lemma in a manner similar to what worked for concrete binding.

```
Lemma hasType_subst' : ∀ G1 e t,
  G1 |-e e : t
  → ∀ G, G1 = G ++ (x, xt) :: nil
```

```
      → x # G
      → G |-e e1 : xt
      → G |-e subst e : t.
    induction 1; ln;
      match goal with
        | [ L : list free_var, _ : ?x # _ ⊢ _ ] ⇒
          apply TAbs_specialized with L x; eauto 20
      end.
  Qed.
```

The main theorem about substitution of closed expressions follows easily.

```
  Theorem hasType_subst : ∀ e t,
    (x, xt) :: nil |-e e : t
    → nil |-e e1 : xt
    → nil |-e subst e : t.
    intros; eapply hasType_subst'; eauto.
  Qed.
End subst.

Hint Resolve hasType_subst.
```

We can define the operational semantics in almost the same way as in previous examples.

```
Notation "[ x ~> e1 ] e2" := (subst x e1 e2) (no associativity, at level 60).

Inductive val : exp → Prop :=
| VConst : ∀ b, val (Const b)
| VAbs : ∀ e, val (Abs e).

Hint Constructors val.

Reserved Notation "e1 ==> e2" (no associativity, at level 90).

Inductive step : exp → exp → Prop :=
| Beta : ∀ e1 e2 x,
  val e2
  → ¬ In x (freeVars e1)
  → App (Abs e1) e2 ==> [x ⇝ e2] (open x O e1)
| Cong1 : ∀ e1 e2 e1',
  e1 ==> e1'
  → App e1 e2 ==> App e1' e2
| Cong2 : ∀ e1 e2 e2',
  val e1
  → e2 ==> e2'
  → App e1 e2 ==> App e1 e2'

  where "e1 ==> e2" := (step e1 e2).

Hint Constructors step.
```

The only interesting change is that the Beta rule requires identifying a fresh variable x to use in opening the abstraction body. We could have avoided this by implementing a more general open that allows substituting expressions for variables, not just variables for variables, but it simplifies the proofs to have just one general substitution function.

Now we are ready to prove progress and preservation. The same proof script from the last examples suffices to prove progress, though significantly different lemmas are applied for us by `eauto`.

```
Lemma progress' : ∀ G e t, G |-e e : t
  → G = nil
  → val e ∨ ∃ e', e ==> e'.
  induction 1; crush; eauto;
    try match goal with
          | [ H : _ |-e _ : _ -> _ ⊢ _ ] ⇒ inversion H
        end;
    repeat match goal with
             | [ H : _ ⊢ _ ] ⇒ solve [ inversion H; crush; eauto ]
           end.
Qed.

Theorem progress : ∀ e t, nil |-e e : t
  → val e ∨ ∃ e', e ==> e'.
  intros; eapply progress'; eauto.
Qed.
```

To establish preservation, it is useful to formalize a principle of sound alpha-variation. In particular, when we open an expression with a particular variable and then immediately substitute for the same variable, we can replace that variable with any other that is not free in the body of the opened expression.

```
Lemma alpha_open : ∀ x1 x2 e1 e2 n,
  ¬ In x1 (freeVars e2)
  → ¬ In x2 (freeVars e2)
  → [x1 ⇝ e1](open x1 n e2) = [x2 ⇝ e1](open x2 n e2).
  induction e2; ln.
Qed.

Hint Resolve freshOk_app1 freshOk_app2.
```

Again it is useful to state a direct corollary which is easier to apply in proof search.

```
Lemma hasType_alpha_open : ∀ G L e0 e2 x t,
  ¬ In x (freeVars e0)
  → G |-e [fresh (L ++ freeVars e0) ⇝ e2](open (fresh (L ++ freeVars e0)) 0 e0) : t
  → G |-e [x ⇝ e2](open x 0 e0) : t.
  intros; rewrite (alpha_open x (fresh (L ++ freeVars e0))); auto.
Qed.

Hint Resolve hasType_alpha_open.
```

Now the previous sections' preservation proof scripts finish the job.

```
  Lemma preservation' : ∀ G e t, G |-e e : t
    → G = nil
    → ∀ e', e ==> e'
      → nil |-e e' : t.
    induction 1; inversion 2; crush; eauto;
      match goal with
        | [ H : _ |-e Abs _ : _ ⊢ _ ] ⇒ inversion H
      end; eauto.
  Qed.

  Theorem preservation : ∀ e t, nil |-e e : t
    → ∀ e', e ==> e'
      → nil |-e e' : t.
    intros; eapply preservation'; eauto.
  Qed.

End LocallyNameless.
```

Chapter 16

Dependent De Bruijn Indices

The previous chapter introduced the most common form of de Bruijn indices, without essential use of dependent types. In earlier chapters, we used dependent de Bruijn indices to illustrate tricks for working with dependent types. This chapter presents one complete case study with dependent de Bruijn indices, focusing on producing the most maintainable proof possible of a classic theorem about lambda calculus.

The proof that follows does not provide a complete guide to all kinds of formalization with de Bruijn indices. Rather, it is intended as an example of some simple design patterns that can make proving standard theorems much easier.

We will prove commutativity of capture-avoiding substitution for basic untyped lambda calculus:

$$x_1 \neq x_2 \Rightarrow [e_1/x_1][e_2/x_2]e = [e_2/x_2][[e_2/x_2]e_1/x_1]e$$

16.1 Defining Syntax and Its Associated Operations

Our definition of expression syntax should be unsurprising. An expression of type exp *n* may refer to *n* different free variables.

```
Inductive exp : nat → Type :=
| Var : ∀ n, fin n → exp n
| App : ∀ n, exp n → exp n → exp n
| Abs : ∀ n, exp (S n) → exp n.
```

The classic implementation of substitution in de Bruijn terms requires an auxiliary operation, *lifting*, which increments the indices of all free variables in an expression. We need to lift whenever we "go under a binder." It is useful to write an auxiliary function liftVar that lifts a variable; that is, liftVar x y will return $y + 1$ if $y \geq x$, and it will return y otherwise. This simple description uses numbers rather than our dependent **fin** family, so the actual specification is more involved.

Combining a number of dependent types tricks, we wind up with this concrete realization.

```
Fixpoint liftVar n (x : fin n) : fin (pred n) → fin n :=
  match x with
    | First _ ⇒ fun y ⇒ Next y
    | Next _ x' ⇒ fun y ⇒
      match y in fin n' return fin n' → (fin (pred n') → fin n')
        → fin (S n') with
        | First _ ⇒ fun x' _ ⇒ First
        | Next _ y' ⇒ fun _ fx' ⇒ Next (fx' y')
      end x' (liftVar x')
  end.
```

Now it is easy to implement the main lifting operation.

```
Fixpoint lift n (e : exp n) : fin (S n) → exp (S n) :=
  match e with
    | Var _ f' ⇒ fun f ⇒ Var (liftVar f f')
    | App _ e1 e2 ⇒ fun f ⇒ App (lift e1 f) (lift e2 f)
    | Abs _ e1 ⇒ fun f ⇒ Abs (lift e1 (Next f))
  end.
```

To define substitution itself, we will need to apply some explicit type casts, based on equalities between types. A single equality will suffice for all of our casts. Its statement is somewhat strange: it quantifies over a variable f of type **fin** *n*, but then never mentions f. Rather, quantifying over f is useful because **fin** is a dependent type that is inhabited or not depending on its index. The body of the theorem, S (pred *n*) = *n*, is true only for $n > 0$, but we can prove it by contradiction when $n = 0$, because we have around a value f of the uninhabited type **fin** 0.

```
Theorem nzf : ∀ n (f : fin n), S (pred n) = n.
  destruct 1; trivial.
Qed.
```

Now we define a notation to streamline our cast expressions. The code [f `return` *n*, *r* `for` e] denotes a cast of expression e whose type can be obtained by substituting some number *n1* for *n* in *r*. f should be a proof that *n1* = *n2*, for any *n2*. In that case, the type of the cast expression is *r* with *n2* substituted for *n*.

```
Notation "[ f 'return' n , r 'for' e ]" :=
  match f in _ = n return r with
    | refl_equal ⇒ e
  end.
```

This notation is useful in defining a variable substitution operation. The idea is that substVar *x y* returns None if *x* = *y*; otherwise, it returns a "squished" version of *y* with a smaller **fin** index, reflecting that variable *x* has been substituted away. Without dependent

types, this would be a simple definition. With dependency, it is reasonably intricate, and our main task in automating proofs about it will be hiding that intricacy.

```
Fixpoint substVar n (x : fin n) : fin n → option (fin (pred n)) :=
  match x with
    | First _ ⇒ fun y ⇒
      match y in fin n' return option (fin (pred n')) with
        | First _ ⇒ None
        | Next _ f' ⇒ Some f'
      end
    | Next _ x' ⇒ fun y ⇒
      match y in fin n'
        return fin (pred n') → (fin (pred n') → option (fin (pred (pred n'))))
          → option (fin (pred n')) with
        | First _ ⇒ fun x' _ ⇒ Some [nzf x' return n, fin n for First]
        | Next _ y' ⇒ fun _ fx' ⇒
          match fx' y' with
            | None ⇒ None
            | Some f ⇒ Some [nzf y' return n, fin n for Next f]
          end
      end x' (substVar x')
  end.
```

It is now easy to define our final substitution function. The abstraction case involves two casts, where one uses the sym_eq function to convert a proof of *n1* = *n2* into a proof of *n2* = *n1*.

```
Fixpoint subst n (e : exp n) : fin n → exp (pred n) → exp (pred n) :=
  match e with
    | Var _ f' ⇒ fun f v ⇒ match substVar f f' with
                             | None ⇒ v
                             | Some f'' ⇒ Var f''
                           end
    | App _ e1 e2 ⇒ fun f v ⇒ App (subst e1 f v) (subst e2 f v)
    | Abs _ e1 ⇒ fun f v ⇒ Abs [sym_eq (nzf f) return n, exp n for
      subst e1 (Next f) [nzf f return n, exp n for lift v First]]
  end.
```

Our final commutativity theorem is about subst, but our proofs will rely on a few more auxiliary definitions. First, we will want an operation more that increments the index of a fin while preserving its interpretation as a number.

```
Fixpoint more n (f : fin n) : fin (S n) :=
  match f with
    | First _ ⇒ First
    | Next _ f' ⇒ Next (more f')
```

```
  end.
```

Second, we will want a kind of inverse to liftVar.

```
Fixpoint unliftVar n (f : fin n) : fin (pred n) → fin (pred n) :=
  match f with
    | First _ ⇒ fun g ⇒ [nzf g return n, fin n for First]
    | Next _ f' ⇒ fun g ⇒
      match g in fin n'
        return fin n' → (fin (pred n') → fin (pred n')) → fin n' with
        | First _ ⇒ fun f' _ ⇒ f'
        | Next _ g' ⇒ fun _ unlift ⇒ Next (unlift g')
      end f' (unliftVar f')
  end.
```

16.2 Custom Tactics

Less than a page of tactic code will be sufficient to automate our proof of commuativity. We start by defining a workhorse simplification tactic *simp*, which extends *crush* in a few ways.

```
Ltac simp := repeat progress (crush; try discriminate;
```

We enter an inner loop of applying hints specific to our domain.

```
  repeat match goal with
```

Our first two hints find places where equality proofs are pattern-matched on. The first hint matches pattern-matches in the conclusion, while the second hint matches pattern-matches in hypotheses. In each case, we apply the library theorem UIP_refl, which says that any proof of a fact like e = e is itself equal to refl_equal. Rewriting with this fact enables reduction of the pattern-match that we found.

```
    | [ ⊢ context[match ?pf with refl_equal ⇒ _ end] ] ⇒
    rewrite (UIP_refl _ _ pf)
    | [ _ : context[match ?pf with refl_equal ⇒ _ end] ⊢ _ ] ⇒
    rewrite (UIP_refl _ _ pf) in *
```

The next hint finds an opportunity to invert a fin equality hypothesis.

```
    | [ H : Next _ = Next _ ⊢ _ ] ⇒ injection H; clear H
```

If we have two equality hypotheses that share a lefthand side, we can use one to rewrite the other, bringing the hypotheses' righthand sides together in a single equation.

```
    | [ H : ?E = _, H' : ?E = _ ⊢ _ ] ⇒ rewrite H in H'
```

Finally, we would like automatic use of quantified equality hypotheses to perform rewriting. We pattern-match a hypothesis H asserting proposition P. We try to use H to perform rewriting everywhere in our goal. The rewrite succeeds if it generates no additional hypotheses, and, to prevent infinite loops in proof search, we clear H if it begins with universal quantification.

```
    | [ H : ?P ⊢ _ ] ⇒ rewrite H in *; [match P with
                                       | ∀ x, _ ⇒ clear H
                                       | _ ⇒ idtac
                                       end]
  end).
```

In implementing another level of automation, it will be useful to mark which free variables we generated with tactics, as opposed to which were present in the original theorem statement. We use a dummy marker predicate Generated to record that information. A tactic *not_generated* fails if and only if its argument is a generated variable, and a tactic *generate* records that its argument is generated.

```
Definition Generated n (_ : fin n) := True.

Ltac not_generated x :=
  match goal with
    | [ _ : Generated x ⊢ _ ] ⇒ fail 1
    | _ ⇒ idtac
  end.

Ltac generate x := assert (Generated x); [ constructor | ].
```

A tactic *destructG* performs case analysis on **fin** values. The built-in case analysis tactics are not smart enough to handle all situations, and we also want to mark new variables as generated, to avoid infinite loops of case analysis. Our *destructG* tactic will only proceed if its argument is not generated.

```
Theorem fin_inv : ∀ n (f : fin (S n)), f = First ∨ ∃ f', f = Next f'.
  intros; dep_destruct f; eauto.
Qed.

Ltac destructG E :=
  not_generated E; let x := fresh "x" in
    (destruct (fin_inv E) as [ | [x]] || destruct E as [ | ? x]);
    [ | generate x ].
```

Our most powerful workhorse tactic will be *dester*, which incorporates all of *simp*'s simplifications and adds heuristics for automatic case analysis and automatic quantifier instantiation.

```
Ltac dester := simp;
  repeat (match goal with
```

The first hint expresses our main insight into quantifier instantiation. We identify a hypothesis *IH* that begins with quantification over **fin** values. We also identify a free **fin** variable x and an arbitrary equality hypothesis H. Given these, we try instantiating *IH* with x. We know we chose correctly if the instantiated proposition includes an opportunity to rewrite using H.

```
| [ x : fin _, H : _ = _, IH : ∀ f : fin _, _ ⊢ _ ] ⇒
  generalize (IH x); clear IH; intro IH; rewrite H in IH
```

This basic idea suffices for all of our explicit quantifier instantiation. We add one more variant that handles cases where an opportunity for rewriting is only exposed if two different quantifiers are instantiated at once.

```
| [ x : fin _, y : fin _, H : _ = _,
    IH : ∀ (f : fin _) (g : fin _), _ ⊢ _ ] ⇒
  generalize (IH x y); clear IH; intro IH; rewrite H in IH
```

We want to case-analyze on any **fin** expression that is the discriminee of a `match` expression or an argument to more.

```
| [ ⊢ context[match ?E with First _ ⇒ _ | Next _ _ ⇒ _ end] ] ⇒
  destructG E
| [ _ : context[match ?E with First _ ⇒ _ | Next _ _ ⇒ _ end] ⊢ _ ] ⇒
  destructG E
| [ ⊢ context[more ?E] ] ⇒ destructG E
```

Recall that *simp* will simplify equality proof terms of facts like `e = e`. The proofs in question will either be of $n = S$ (pred n) or S (pred n) $= n$, for some n. These equations do not have syntactically equal sides. We can get to the point where they *do* have equal sides by performing case analysis on n. Whenever we do so, the $n = 0$ case will be contradictory, allowing us to discharge it by finding a free variable of type **fin** 0 and performing inversion on it. In the $n = S$ n' case, the sides of these equalities will simplify to equal values, as needed. The next two hints identify n values that are good candidates for such case analysis.

```
| [ x : fin ?n ⊢ _ ] ⇒
  match goal with
    | [ ⊢ context[nzf x] ] ⇒
      destruct n; [ inversion x | ]
```

```
    end
  | [ x : fin (pred ?n), y : fin ?n ⊢ _ ] ⇒
    match goal with
      | [ ⊢ context[nzf x] ] ⇒
        destruct n; [ inversion y | ]
    end
```

Finally, we find `match` discriminees of **option** type, enforcing that we do not destruct any discriminees that are themselves `match` expressions. Crucially, we do these case analyses with *case_eq* instead of `destruct`. The former adds equality hypotheses to record the relationships between old variables and their new deduced forms. These equalities will be used by our quantifier instantiation heuristic.

```
  | [ ⊢ context[match ?E with None ⇒ _ | Some _ ⇒ _ end] ] ⇒
    match E with
      | match _ with None ⇒ _ | Some _ ⇒ _ end ⇒ fail 1
      | _ ⇒ case_eq E; firstorder
    end
```

Each iteration of the loop ends by calling *simp* again, and, after no more progress can be made, we finish by calling `eauto`.

```
  end; simp); eauto.
```

16.3 Theorems

We are now ready to prove our main theorem, by way of a progression of lemmas.

The first pair of lemmas characterizes the interaction of substitution and lifting at the variable level.

```
Lemma substVar_unliftVar : ∀ n (f0 : fin n) f g,
  match substVar f0 f, substVar (liftVar f0 g) f with
    | Some f1, Some f2 ⇒ ∃ f', substVar g f1 = Some f'
      ∧ substVar (unliftVar f0 g) f2 = Some f'
    | Some f1, None ⇒ substVar g f1 = None
    | None, Some f2 ⇒ substVar (unliftVar f0 g) f2 = None
    | None, None ⇒ False
  end.
  induction f0; dester.
Qed.

Lemma substVar_liftVar : ∀ n (f0 : fin n) f,
```

```
  substVar f0 (liftVar f0 f) = Some f.
  induction f0; dester.
Qed.
```

Next, we define a notion of "greater-than-or-equal" for **fin** values, prove an inversion theorem for it, and add that theorem as a hint.

```
Inductive fin_ge : ∀ n1, fin n1 → ∀ n2, fin n2 → Prop :=
| GeO : ∀ n1 (f1 : fin n1) n2,
  fin_ge f1 (First : fin (S n2))
| GeS : ∀ n1 (f1 : fin n1) n2 (f2 : fin n2),
  fin_ge f1 f2
  → fin_ge (Next f1) (Next f2).

Hint Constructors fin_ge.

Lemma fin_ge_inv' : ∀ n1 n2 (f1 : fin n1) (f2 : fin n2),
  fin_ge f1 f2
  → match f1, f2 with
      | Next _ f1', Next _ f2' ⇒ fin_ge f1' f2'
      | _, _ ⇒ True
    end.
  destruct 1; dester.
Qed.

Lemma fin_ge_inv : ∀ n1 n2 (f1 : fin n1) (f2 : fin n2),
  fin_ge (Next f1) (Next f2)
  → fin_ge f1 f2.
  intros; generalize (fin_ge_inv' (f1 := Next f1) (f2 := Next f2)); dester.
Qed.

Hint Resolve fin_ge_inv.
```

A congruence lemma for the **fin** constructor `Next` is similarly useful.

```
Lemma Next_cong : ∀ n (f1 f2 : fin n),
  f1 = f2
  → Next f1 = Next f2.
  dester.
Qed.

Hint Resolve Next_cong.
```

We prove a crucial lemma about liftVar in terms of **fin_ge**.

```
Lemma liftVar_more : ∀ n (f : fin n) (f0 : fin (S n)) g,
  fin_ge g f0
  → match liftVar f0 f in fin n'
      return fin n' → (fin (pred n') → fin n') → fin (S n') with
      | First n0 ⇒ fun _ _ ⇒ First
```

```
      | Next n0 y' ⇒ fun _ fx' ⇒ Next (fx' y')
    end g (liftVar g) = liftVar (more f0) (liftVar g f).
  induction f; inversion 1; dester.
Qed.

Hint Resolve liftVar_more.
```

We suggest a particular way of changing the form of a goal, so that other hints are able to match.

```
Hint Extern 1 (_ = lift _ (Next (more ?f))) ⇒
  change (Next (more f)) with (more (Next f)).
```

We suggest applying the `f_equal` tactic to simplify equalities over expressions. For instance, this would reduce a goal App *f1 x1* = App *f2 x2* to two goals *f1* = *f2* and *x1* = *x2*.

```
Hint Extern 1 (eq (A := exp _) _ _) ⇒ f_equal.
```

Our consideration of lifting in isolation finishes with another hint lemma. The auxiliary lemma with a strengthened induction hypothesis is where we put **fin_ge** to use, and we do not need to mention that predicate again afteward.

```
Lemma double_lift' : ∀ n (e : exp n) f g,
  fin_ge g f
  → lift (lift e f) (Next g) = lift (lift e g) (more f).
  induction e; dester.
Qed.

Lemma double_lift : ∀ n (e : exp n) g,
  lift (lift e First) (Next g) = lift (lift e g) First.
  intros; apply double_lift'; dester.
Qed.

Hint Resolve double_lift.
```

Now we characterize the interaction of substitution and lifting on variables. We start with a more general form substVar_lift' of the final lemma substVar_lift, with the latter proved as a direct corollary of the former.

```
Lemma substVar_lift' : ∀ n (f0 : fin n) f g,
  substVar [nzf f0 return n, fin (S n) for
    liftVar (more g) [sym_eq (nzf f0) return n, fin n for f0]]
  (liftVar (liftVar (Next f0) [nzf f0 return n, fin n for g]) f)
  = match substVar f0 f with
      | Some f'' ⇒ Some [nzf f0 return n, fin n for liftVar g f'']
      | None ⇒ None
    end.
  induction f0; dester.
Qed.
```

```
Lemma substVar_lift : ∀ n (f0 f g : fin (S n)),
  substVar (liftVar (more g) f0) (liftVar (liftVar (Next f0) g) f)
  = match substVar f0 f with
      | Some f'' ⇒ Some (liftVar g f'')
      | None ⇒ None
    end.
  intros; generalize (substVar_lift' f0 f g); dester.
Qed.
```

We follow a similar decomposition for the expression-level theorem about substitution and lifting.

```
Lemma lift_subst' : ∀ n (e1 : exp n) f g e2,
  lift (subst e1 f e2) g
  = [sym_eq (nzf f) return n, exp n for
    subst
      (lift e1 (liftVar (Next f) [nzf f return n, fin n for g]))
      [nzf f return n, fin (S n) for
        liftVar (more g) [sym_eq (nzf f) return n, fin n for f]]
      [nzf f return n, exp n for lift e2 g]].
  induction e1; generalize substVar_lift; dester.
Qed.

Lemma lift_subst : ∀ n g (e2 : exp (S n)) e3,
  subst (lift e2 First) (Next g) (lift e3 First) = lift (n := n) (subst e2 g e3) First.
  intros; generalize (lift_subst' e2 g First e3); dester.
Qed.

Hint Resolve lift_subst.
```

Our last auxiliary lemma characterizes a situation where substitution can undo the effects of lifting.

```
Lemma undo_lift' : ∀ n (e1 : exp n) e2 f,
  subst (lift e1 f) f e2 = e1.
  induction e1; generalize substVar_liftVar; dester.
Qed.

Lemma undo_lift : ∀ n e2 e3 (f0 : fin (S (S n))) g,
  e3 = subst (lift e3 (unliftVar f0 g)) (unliftVar f0 g)
    (subst (n := S n) e2 g e3).
  generalize undo_lift'; dester.
Qed.

Hint Resolve undo_lift.
```

Finally, we arrive at the substitution commutativity theorem.

```
Lemma subst_comm' : ∀ n (e1 : exp n) f g e2 e3,
  subst (subst e1 f e2) g e3
```

```
  = subst
  (subst e1 (liftVar f g) [nzf g return n, exp n for
    lift e3 [sym_eq (nzf g) return n, fin n for unliftVar f g]])
  (unliftVar f g)
  (subst e2 g e3).
  induction e1; generalize (substVar_unliftVar (n := n)); dester.
Qed.

Theorem subst_comm : ∀ (e1 : exp 2) e2 e3,
  subst (subst e1 First e2) First e3
  = subst (subst e1 (Next First) (lift e3 First)) First (subst e2 First e3).
  intros; generalize (subst_comm' e1 First First e2 e3); dester.
Qed.
```

The final theorem is specialized to the case of substituting in an expression with exactly two free variables, which yields a statement that is readable enough, as statements about de Bruijn indices go.

This proof script is resilient to specification changes. It is easy to add new constructors to the language being treated. The proofs adapt automatically to the addition of any constructor whose subterms each involve zero or one new bound variables. That is, to add such a constructor, we only need to add it to the definition of exp and add (quite obvious) cases for it in the definitions of lift and `subst`.

Chapter 17

Higher-Order Abstract Syntax

In many cases, detailed reasoning about variable binders and substitution is a small annoyance; in other cases, it becomes the dominant cost of proving a theorem formally. No matter which of these possibilities prevails, it is clear that it would be very pragmatic to find a way to avoid reasoning about variable identity or freshness. A well-established alternative to first-order encodings is *higher-order abstract syntax*, or HOAS. In mechanized theorem-proving, HOAS is most closely associated with the LF meta logic and the tools based on it, including Twelf.

In this chapter, we will see that HOAS cannot be implemented directly in Coq. However, a few very similar encodings are possible and are in fact very effective in some important domains.

17.1 Classic HOAS

The motto of HOAS is simple: represent object language binders using meta language binders. Here, "object language" refers to the language being formalized, while the meta language is the language in which the formalization is done. Our usual meta language, Coq's Gallina, contains the standard binding facilities of functional programming, making it a promising base for higher-order encodings.

Recall the concrete encoding of basic untyped lambda calculus expressions.

```
Inductive uexp : Set :=
| UVar : string → uexp
| UApp : uexp → uexp → uexp
| UAbs : string → uexp → uexp.
```

The explicit presence of variable names forces us to think about issues of freshness and variable capture. The HOAS alternative would look like this.

Reset uexp.

```
Inductive uexp : Set :=
```

```
| UApp : uexp → uexp → uexp
| UAbs : (uexp → uexp) → uexp.
```

We have avoided any mention of variables. Instead, we encode the binding done by abstraction using the binding facilities associated with Gallina functions. For instance, we might represent the term $\lambda x.\ x\ x$ as UAbs (`fun` $x \Rightarrow$ UApp $x\ x$). Coq has built-in support for matching binders in anonymous `fun` expressions to their uses, so we avoid needing to implement our own binder-matching logic.

This definition is not quite HOAS, because of the broad variety of functions that Coq would allow us to pass as arguments to UAbs. We can thus construct many **uexp**s that do not correspond to normal lambda terms. These deviants are called *exotic terms*. In LF, functions may only be written in a very restrictive computational language, lacking, among other things, pattern matching and recursive function definitions. Thus, thanks to a careful balancing act of design decisions, exotic terms are not possible with usual HOAS encodings in LF.

Our definition of **uexp** has a more fundamental problem: it is invalid in Gallina.

```
Error: Non strictly positive occurrence of "uexp" in
 "(uexp → uexp) → uexp".
```

We have violated a rule that we considered before: an inductive type may not be defined in terms of functions over itself. Way back in Chapter 3, we considered this example and the reasons why we should be glad that Coq rejects it. Thus, we will need to use more cleverness to reap similar benefits.

The root problem is that our expressions contain variables representing expressions of the same kind. Many useful kinds of syntax involve no such cycles. For instance, it is easy to use HOAS to encode standard first-order logic in Coq.

```
Inductive prop : Type :=
| Eq : ∀ T, T → T → prop
| Not : prop → prop
| And : prop → prop → prop
| Or : prop → prop → prop
| Forall : ∀ T, (T → prop) → prop
| Exists : ∀ T, (T → prop) → prop.

Fixpoint propDenote (p : prop) : Prop :=
  match p with
    | Eq _ x y ⇒ x = y
    | Not p ⇒ ¬ (propDenote p)
    | And p1 p2 ⇒ propDenote p1 ∧ propDenote p2
    | Or p1 p2 ⇒ propDenote p1 ∨ propDenote p2
    | Forall _ f ⇒ ∀ x, propDenote (f x)
    | Exists _ f ⇒ ∃ x, propDenote (f x)
```

```
  end.
```

Unfortunately, there are other recursive functions that we might like to write but cannot. One simple example is a function to count the number of constructors used to build a **prop**. To look inside a Forall or Exists, we need to look inside the quantifier's body, which is represented as a function. In Gallina, as in most statically-typed functional languages, the only way to interact with a function is to call it. We have no hope of doing that here; the domain of the function in question has an arbitary type T, so T may even be uninhabited. If we had a universal way of constructing values to look inside functions, we would have uncovered a consistency bug in Coq!

We are still suffering from the possibility of writing exotic terms, such as this example:

```
Example true_prop := Eq 1 1.
Example false_prop := Not true_prop.
Example exotic_prop := Forall (fun b : bool ⇒ if b then true_prop else false_prop).
```

Thus, the idea of a uniform way of looking inside a binder to find another well-defined **prop** is hopelessly doomed.

A clever HOAS variant called *weak HOAS* manages to rule out exotic terms in Coq. Here is a weak HOAS version of untyped lambda terms.

```
Parameter var : Set.

Inductive uexp : Set :=
| UVar : var → uexp
| UApp : uexp → uexp → uexp
| UAbs : (var → uexp) → uexp.
```

We postulate the existence of some set *var* of variables, and variable nodes appear explicitly in our syntax. A binder is represented as a function over *variables*, rather than as a function over *expressions*. This breaks the cycle that led Coq to reject the literal HOAS definition. It is easy to encode our previous example, $\lambda x.\ x\ x$:

```
Example self_app := UAbs (fun x ⇒ UApp (UVar x) (UVar x)).
```

What about exotic terms? The problems they caused earlier came from the fact that Gallina is expressive enough to allow us to perform case analysis on the types we used as the domains of binder functions. With weak HOAS, we use an abstract type *var* as the domain. Since we assume the existence of no functions for deconstructing *var*s, Coq's type soundness enforces that no Gallina term of type **uexp** can take different values depending on the value of a *var* available in the typing context, *except* by incorporating those variables into a **uexp** value in a legal way.

Weak HOAS retains the other disadvantage of our previous example: it is hard to write recursive functions that deconstruct terms. As with the previous example, some functions *are* implementable. For instance, we can write a function to reverse the function and argument positions of every UApp node.

```
Fixpoint swap (e : uexp) : uexp :=
  match e with
```

```
    | UVar _ ⇒ e
    | UApp e1 e2 ⇒ UApp (swap e2) (swap e1)
    | UAbs e1 ⇒ UAbs (fun x ⇒ swap (e1 x))
  end.
```

However, it is still impossible to write a function to compute the size of an expression. We would still need to manufacture a value of type *var* to peer under a binder, and that is impossible, because *var* is an abstract type.

17.2 Parametric HOAS

In the context of Haskell, Washburn and Weirich introduced a technique called *parametric HOAS*, or PHOAS. By making a slight alteration in the spirit of weak HOAS, we arrive at an encoding that addresses all three of the complaints above: the encoding is legal in Coq, exotic terms are impossible, and it is possible to write any syntax-deconstructing function that we can write with first-order encodings. The last of these advantages is not even present with HOAS in Twelf. In a sense, we receive it in exchange for giving up a free implementation of capture-avoiding substitution.

The first step is to change the weak HOAS type so that *var* is a variable inside a section, rather than a global parameter.

```
Reset uexp.

Section uexp.
  Variable var : Set.

  Inductive uexp : Set :=
  | UVar : var → uexp
  | UApp : uexp → uexp → uexp
  | UAbs : (var → uexp) → uexp.
End uexp.
```

Next, we can encapsulate choices of *var* inside a polymorphic function type.

```
Definition Uexp := ∀ var, uexp var.
```

This type Uexp is our final, exotic-term-free representation of lambda terms. Inside the body of a Uexp function, *var* values may not be deconstructed illegaly, for much the same reason as with weak HOAS. We simply trade an abstract type for parametric polymorphism.

Our running example $\lambda x.\ x\ x$ is easily expressed:

```
Example self_app : Uexp := fun var ⇒ UAbs (var := var)
  (fun x : var ⇒ UApp (var := var) (UVar (var := var) x) (UVar (var := var) x)).
```

Including all mentions of *var* explicitly helps clarify what is happening here, but it is convenient to let Coq's local type inference fill in these occurrences for us.

```
Example self_app' : Uexp := fun _ ⇒ UAbs (fun x ⇒ UApp (UVar x) (UVar x)).
```

We can go further and apply the PHOAS technique to dependently-typed ASTs, where Gallina typing guarantees that only well-typed terms can be represented. For the rest of this chapter, we consider the example of simply-typed lambda calculus with natural numbers and addition. We start with a conventional definition of the type language.

```
Inductive type : Type :=
| Nat : type
| Arrow : type → type → type.

Infix "->" := Arrow (right associativity, at level 60).
```

Our definition of the expression type follows the definition for untyped lambda calculus, with one important change. Now our section variable *var* is not just a type. Rather, it is a *function* returning types. The idea is that a variable of object language type t is represented by a *var* t. Note how this enables us to avoid indexing the exp type with a representation of typing contexts.

```
Section exp.
  Variable var : type → Type.

  Inductive exp : type → Type :=
  | Const' : nat → exp Nat
  | Plus' : exp Nat → exp Nat → exp Nat

  | Var : ∀ t, var t → exp t
  | App' : ∀ dom ran, exp (dom -> ran) → exp dom → exp ran
  | Abs' : ∀ dom ran, (var dom → exp ran) → exp (dom -> ran).
End exp.

Implicit Arguments Const' [var].
Implicit Arguments Var [var t].
Implicit Arguments Abs' [var dom ran].
```

Our final representation type wraps exp as before.

```
Definition Exp t := ∀ var, exp var t.
```

We can define some smart constructors to make it easier to build Exps without using polymorphism explicitly.

```
Definition Const (n : nat) : Exp Nat :=
  fun _ ⇒ Const' n.
Definition Plus (E1 E2 : Exp Nat) : Exp Nat :=
  fun _ ⇒ Plus' (E1 _) (E2 _).
Definition App dom ran (F : Exp (dom -> ran)) (X : Exp dom) : Exp ran :=
  fun _ ⇒ App' (F _) (X _).
```

A case for function abstraction is not as natural, but we can implement one candidate in terms of a type family Exp1, such that Exp1 *free result* represents an expression of type *result* with one free variable of type *free*.

```
Definition Exp1 t1 t2 := ∀ var, var t1 → exp var t2.
Definition Abs dom ran (B : Exp1 dom ran) : Exp (dom -> ran) :=
  fun _ ⇒ Abs' (B _).
```

Now it is easy to encode a number of example programs.

```
Example zero := Const 0.
Example one := Const 1.
Example one_again := Plus zero one.
Example ident : Exp (Nat -> Nat) := Abs (fun _ X ⇒ Var X).
Example app_ident := App ident one_again.
Example app : Exp ((Nat -> Nat) -> Nat -> Nat) := fun _ ⇒
  Abs' (fun f ⇒ Abs' (fun x ⇒ App' (Var f) (Var x))).
Example app_ident' := App (App app ident) one_again.
```

We can write syntax-deconstructing functions, such as CountVars, which counts how many Var nodes appear in an Exp. First, we write a version countVars for exps. The main trick is to specialize countVars to work over expressions where *var* is instantiated as `fun _ ⇒ unit`. That is, every variable is just a value of type **unit**, such that variables carry no information. The important thing is that we have a value tt of type **unit** available, to use in descending into binders.

```
Fixpoint countVars t (e : exp (fun _ ⇒ unit) t) : nat :=
  match e with
    | Const' _ ⇒ 0
    | Plus' e1 e2 ⇒ countVars e1 + countVars e2
    | Var _ _ ⇒ 1
    | App' _ _ e1 e2 ⇒ countVars e1 + countVars e2
    | Abs' _ _ e' ⇒ countVars (e' tt)
  end.
```

We turn countVars into CountVars with explicit instantiation of a polymorphic Exp value *E*. We can write an underscore for the paramter to *E*, because local type inference is able to infer the proper value.

```
Definition CountVars t (E : Exp t) : nat := countVars (E _).
```

A few evaluations establish that CountVars behaves plausibly.

```
Eval compute in CountVars zero.
     = 0
     : nat

Eval compute in CountVars one.
     = 0
     : nat

Eval compute in CountVars one_again.
```

```
    = 0
    : nat
Eval compute in CountVars ident.
    = 1
    : nat
Eval compute in CountVars app_ident.
    = 1
    : nat
Eval compute in CountVars app.
    = 2
    : nat
Eval compute in CountVars app_ident'.
    = 3
    : nat
```

We might want to go further and count occurrences of a single distinguished free variable in an expression. In this case, it is useful to instantiate *var* as `fun _ ⇒ bool`. We will represent the distinguished variable with true and all other variables with false.

```
Fixpoint countOne t (e : exp (fun _ ⇒ bool) t) : nat :=
  match e with
    | Const' _ ⇒ 0
    | Plus' e1 e2 ⇒ countOne e1 + countOne e2
    | Var _ true ⇒ 1
    | Var _ false ⇒ 0
    | App' _ _ e1 e2 ⇒ countOne e1 + countOne e2
    | Abs' _ _ e' ⇒ countOne (e' false)
  end.
```

We wrap countOne into CountOne, which we type using the Exp1 definition from before. CountOne operates on an expression E with a single free variable. We apply an instantiated E to true to mark this variable as the one countOne should look for. countOne itself is careful to instantiate all other variables with false.

```
Definition CountOne t1 t2 (E : Exp1 t1 t2) : nat :=
  countOne (E _ true).
```

We can check the behavior of CountOne on a few examples.

```
Example ident1 : Exp1 Nat Nat := fun _ X ⇒ Var X.
Example add_self : Exp1 Nat Nat := fun _ X ⇒ Plus' (Var X) (Var X).
Example app_zero : Exp1 (Nat -> Nat) Nat := fun _ X ⇒ App' (Var X) (Const' 0).
Example app_ident1 : Exp1 Nat Nat := fun _ X ⇒
  App' (Abs' (fun Y ⇒ Var Y)) (Var X).
```

```
Eval compute in CountOne ident1.
```
```
     = 1
     : nat
```
```
Eval compute in CountOne add_self.
```
```
     = 2
     : nat
```
```
Eval compute in CountOne app_zero.
```
```
     = 1
     : nat
```
```
Eval compute in CountOne app_ident1.
```
```
     = 1
     : nat
```

The PHOAS encoding turns out to be just as general as the first-order encodings we saw previously. To provide a taste of that generality, we implement a translation into concrete syntax, rendered in human-readable strings. This is as easy as representing variables as strings.

```
Section ToString.
  Open Scope string_scope.

  Fixpoint natToString (n : nat) : string :=
    match n with
      | O ⇒ "O"
      | S n' ⇒ "S(" ++ natToString n' ++ ")"
    end.
```

Function toString takes an extra argument *cur*, which holds the last variable name assigned to a binder. We build new variable names by extending *cur* with primes. The function returns a pair of the next available variable name and of the actual expression rendering.

```
  Fixpoint toString t (e : exp (fun _ ⇒ string) t) (cur : string) : string * string :=
    match e with
      | Const' n ⇒ (cur, natToString n)
      | Plus' e1 e2 ⇒
        let (cur', s1) := toString e1 cur in
        let (cur'', s2) := toString e2 cur' in
        (cur'', "(" ++ s1 ++ ") + (" ++ s2 ++ ")")
      | Var _ s ⇒ (cur, s)
      | App' _ _ e1 e2 ⇒
        let (cur', s1) := toString e1 cur in
        let (cur'', s2) := toString e2 cur' in
        (cur'', "(" ++ s1 ++ ") (" ++ s2 ++ ")")
      | Abs' _ _ e' ⇒
```

```
      let (cur', s) := toString (e' cur) (cur ++ "'") in
      (cur', "(\" ++ cur ++ ", " ++ s ++ ")")
  end.

  Definition ToString t (E : Exp t) : string := snd (toString (E _) "x").
End ToString.

Eval compute in ToString zero.
     = "O"%string
     : string

Eval compute in ToString one.
     = "S(O)"%string
     : string

Eval compute in ToString one_again.
     = "(O) + (S(O))"%string
     : string

Eval compute in ToString ident.
     = "(\x, x)"%string
     : string

Eval compute in ToString app_ident.
     = "((\x, x)) ((O) + (S(O)))"%string
     : string

Eval compute in ToString app.
     = "(\x, (\x', (x) (x')))"%string
     : string

Eval compute in ToString app_ident'.
     = "(((\x, (\x', (x) (x')))) ((\x'', x''))) ((O) + (S(O)))"%string
     : string
```

Our final example is crucial to using PHOAS to encode standard operational semantics. We define capture-avoiding substitution, in terms of a function flatten which takes in an expression that represents variables as expressions. flatten replaces every node Var e with e.

```
Section flatten.
  Variable var : type → Type.

  Fixpoint flatten t (e : exp (exp var) t) : exp var t :=
    match e with
      | Const' n ⇒ Const' n
      | Plus' e1 e2 ⇒ Plus' (flatten e1) (flatten e2)
      | Var _ e' ⇒ e'
      | App' _ _ e1 e2 ⇒ App' (flatten e1) (flatten e2)
```

```
    | Abs' _ _ e' ⇒ Abs' (fun x ⇒ flatten (e' (Var x)))
  end.
End flatten.
```

Flattening turns out to implement the heart of substitution. We apply *E2*, which has one free variable, to *E1*, replacing the occurrences of the free variable by copies of *E1*. flatten takes care of removing the extra Var applications around these copies.

```
Definition Subst t1 t2 (E1 : Exp t1) (E2 : Exp1 t1 t2) : Exp t2 := fun _ ⇒
  flatten (E2 _ (E1 _)).

Eval compute in Subst one ident1.
    = fun var : type → Type ⇒ Const' 1
    : Exp Nat

Eval compute in Subst one add_self.
    = fun var : type → Type ⇒ Plus' (Const' 1) (Const' 1)
    : Exp Nat

Eval compute in Subst ident app_zero.
    = fun var : type → Type ⇒
      App' (Abs' (fun X : var Nat ⇒ Var X)) (Const' 0)
    : Exp Nat

Eval compute in Subst one app_ident1.
    = fun var : type → Type ⇒
      App' (Abs' (fun x : var Nat ⇒ Var x)) (Const' 1)
    : Exp Nat
```

17.3 A Type Soundness Proof

With Subst defined, there are few surprises encountered in defining a standard small-step, call-by-value semantics for our object language. We begin by classifying a subset of expressions as values.

```
Inductive Val : ∀ t, Exp t → Prop :=
| VConst : ∀ n, Val (Const n)
| VAbs : ∀ dom ran (B : Exp1 dom ran), Val (Abs B).

Hint Constructors Val.
```

Since this language is more complicated than the one we considered in the chapter on first-order encodings, we will use explicit evaluation contexts to define the semantics. A value of type Ctx t u is a context that yields an expression of type u when filled by an expression of type t. We have one context for each position of the App and Plus constructors.

```
Inductive Ctx : type → type → Type :=
| AppCong1 : ∀ (dom ran : type),
```

```
  Exp dom → Ctx (dom -> ran) ran
| AppCong2 : ∀ (dom ran : type),
  Exp (dom -> ran) → Ctx dom ran
| PlusCong1 : Exp Nat → Ctx Nat Nat
| PlusCong2 : Exp Nat → Ctx Nat Nat.
```

A judgment characterizes when contexts are valid, enforcing the standard call-by-value restriction that certain positions must hold values.

```
Inductive isCtx : ∀ t1 t2, Ctx t1 t2 → Prop :=
| IsApp1 : ∀ dom ran (X : Exp dom), isCtx (AppCong1 ran X)
| IsApp2 : ∀ dom ran (F : Exp (dom -> ran)), Val F → isCtx (AppCong2 F)
| IsPlus1 : ∀ E2, isCtx (PlusCong1 E2)
| IsPlus2 : ∀ E1, Val E1 → isCtx (PlusCong2 E1).
```

A simple definition implements plugging a context with a specific expression.

```
Definition plug t1 t2 (C : Ctx t1 t2) : Exp t1 → Exp t2 :=
  match C with
    | AppCong1 _ _ X ⇒ fun F ⇒ App F X
    | AppCong2 _ _ F ⇒ fun X ⇒ App F X
    | PlusCong1 E2 ⇒ fun E1 ⇒ Plus E1 E2
    | PlusCong2 E1 ⇒ fun E2 ⇒ Plus E1 E2
  end.

Infix "@" := plug (no associativity, at level 60).
```

Finally, we have the step relation itself, which combines our ingredients in the standard way. In the congruence rule, we introduce the extra variable *E1* and its associated equality to make the rule easier for `eauto` to apply.

```
Reserved Notation "E1 ==> E2" (no associativity, at level 90).

Inductive Step : ∀ t, Exp t → Exp t → Prop :=
| Beta : ∀ dom ran (B : Exp1 dom ran) (X : Exp dom),
  Val X
  → App (Abs B) X ==> Subst X B
| Sum : ∀ n1 n2,
  Plus (Const n1) (Const n2) ==> Const (n1 + n2)
| Cong : ∀ t t' (C : Ctx t t') E E' E1,
  isCtx C
  → E1 = C @ E
  → E ==> E'
  → E1 ==> C @ E'

  where "E1 ==> E2" := (Step E1 E2).

Hint Constructors isCtx Step.
```

To prove type soundness for this semantics, we need to overcome one crucial obstacle.

Standard proofs use induction on the structure of typing derivations. Our encoding mixes typing derivations with expression syntax, so we want to induct over expression structure. Our expressions are represented as functions, which do not, in general, admit induction in Coq. However, because of our use of parametric polymorphism, we know that our expressions do, in fact, have inductive structure. In particular, every closed value of Exp type must belong to the following relation.

```
Inductive Closed : ∀ t, Exp t → Prop :=
| CConst : ∀ n,
  Closed (Const n)
| CPlus : ∀ E1 E2,
  Closed E1
  → Closed E2
  → Closed (Plus E1 E2)
| CApp : ∀ dom ran (E1 : Exp (dom -> ran)) E2,
  Closed E1
  → Closed E2
  → Closed (App E1 E2)
| CAbs : ∀ dom ran (E1 : Exp1 dom ran),
  Closed (Abs E1).
```

How can we prove such a fact? It probably cannot be established in Coq without axioms. Rather, one would have to establish it metatheoretically, reasoning informally outside of Coq. For now, we assert the fact as an axiom. The later chapter on intensional transformations shows one approach to removing the need for an axiom.

```
Axiom closed : ∀ t (E : Exp t), Closed E.
```

The usual progress and preservation theorems are now very easy to prove. In fact, preservation is implicit in our dependently-typed definition of **Step**. This is a huge win, because we avoid completely the theorem about substitution and typing that made up the bulk of each proof in the chapter on first-order encodings. The progress theorem yields to a few lines of automation.

We define a slight variant of *crush* which also looks for chances to use the theorem inj_pair2 on hypotheses. This theorem deals with an artifact of the way that `inversion` works on dependently-typed hypotheses.

```
Ltac my_crush' :=
  crush;
  repeat (match goal with
            | [ H : _ ⊢ _ ] ⇒ generalize (inj_pair2 _ _ _ _ _ H); clear H
          end; crush).

Hint Extern 1 (_ = _ @ _) ⇒ simpl.
```

This is the point where we need to do induction over functions, in the form of expressions E. The judgment **Closed** provides the perfect framework; we induct over **Closed** derivations.

```
Lemma progress' : ∀ t (E : Exp t),
```

```
  Closed E
  → Val E ∨ ∃ E', E ==> E'.
  induction 1; crush;
    repeat match goal with
             | [ H : Val _ ⊢ _ ] ⇒ inversion H; []; clear H; my_crush'
           end; eauto 6.
Qed.
```

Our final proof of progress makes one top-level use of the axiom *closed* that we asserted above.

```
Theorem progress : ∀ t (E : Exp t),
  Val E ∨ ∃ E', E ==> E'.
  intros; apply progress'; apply closed.
Qed.
```

17.4 Big-Step Semantics

Another standard exercise in operational semantics is proving the equivalence of small-step and big-step semantics. We can carry out this exercise for our PHOAS lambda calculus. Most of the steps are just as pleasant as in the previous section, but things get complicated near to the end.

We must start by defining the big-step semantics itself. The definition is completely standard.

```
Reserved Notation "E1 ===> E2" (no associativity, at level 90).

Inductive BigStep : ∀ t, Exp t → Exp t → Prop :=
| SConst : ∀ n,
  Const n ===> Const n
| SPlus : ∀ E1 E2 n1 n2,
  E1 ===> Const n1
  → E2 ===> Const n2
  → Plus E1 E2 ===> Const (n1 + n2)

| SApp : ∀ dom ran (E1 : Exp (dom -> ran)) E2 B V2 V,
  E1 ===> Abs B
  → E2 ===> V2
  → Subst V2 B ===> V
  → App E1 E2 ===> V
| SAbs : ∀ dom ran (B : Exp1 dom ran),
  Abs B ===> Abs B

  where "E1 ===> E2" := (BigStep E1 E2).
```

```
Hint Constructors BigStep.
```

To prove a crucial intermediate lemma, we will want to name the transitive-reflexive closure of the small-step relation.

```
Reserved Notation "E1 ==>* E2" (no associativity, at level 90).

Inductive MultiStep : ∀ t, Exp t → Exp t → Prop :=
| Done : ∀ t (E : Exp t), E ==>* E
| OneStep : ∀ t (E E' E'' : Exp t),
  E ==> E'
  → E' ==>* E''
  → E ==>* E''

  where "E1 ==>* E2" := (MultiStep E1 E2).

Hint Constructors MultiStep.
```

A few basic properties of evaluation and values admit easy proofs.

```
Theorem MultiStep_trans : ∀ t (E1 E2 E3 : Exp t),
  E1 ==>* E2
  → E2 ==>* E3
  → E1 ==>* E3.
  induction 1; eauto.
Qed.

Theorem Big_Val : ∀ t (E V : Exp t),
  E ===> V
  → Val V.
  induction 1; crush.
Qed.

Theorem Val_Big : ∀ t (V : Exp t),
  Val V
  → V ===> V.
  destruct 1; crush.
Qed.

Hint Resolve Big_Val Val_Big.
```

Another useful property deals with pushing multi-step evaluation inside of contexts.

```
Lemma Multi_Cong : ∀ t t' (C : Ctx t t'),
  isCtx C
  → ∀ E E', E ==>* E'
    → C @ E ==>* C @ E'.
  induction 2; crush; eauto.
Qed.

Lemma Multi_Cong' : ∀ t t' (C : Ctx t t') E1 E2 E E',
```

```
  isCtx C
  → E1 = C @ E
  → E2 = C @ E'
  → E ==>* E'
  → E1 ==>* E2.
  crush; apply Multi_Cong; auto.
Qed.

Hint Resolve Multi_Cong'.
```

Unrestricted use of transitivity of $==>^*$ can lead to very large `eauto` search spaces, which has very inconvenient efficiency consequences. Instead, we define a special tactic *mtrans* that tries applying transitivity with a particular intermediate expression.

```
Ltac mtrans E :=
  match goal with
    | [ ⊢ E ==>* _ ] ⇒ fail 1
    | _ ⇒ apply MultiStep_trans with E; [ solve [ eauto ] | eauto ]
  end.
```

With *mtrans*, we can give a reasonably short proof of one direction of the equivalence between big-step and small-step semantics. We include proof cases specific to rules of the big-step semantics, since leaving the details to `eauto` would lead to a very slow proof script. The use of *solve* in *mtrans*'s definition keeps us from going down unfruitful paths.

```
Theorem Big_Multi : ∀ t (E V : Exp t),
  E ===> V
  → E ==>* V.
  induction 1; crush; eauto;
    repeat match goal with
             | [ n1 : _, E2 : _ ⊢ _ ] ⇒ mtrans (Plus (Const n1) E2)
             | [ n1 : _, n2 : _ ⊢ _ ] ⇒ mtrans (Plus (Const n1) (Const n2))
             | [ B : _, E2 : _ ⊢ _ ] ⇒ mtrans (App (Abs B) E2)
           end.
Qed.
```

We are almost ready to prove the other direction of the equivalence. First, we wrap an earlier lemma in a form that will work better with `eauto`.

```
Lemma Big_Val' : ∀ t (V1 V2 : Exp t),
  Val V2
  → V1 = V2
  → V1 ===> V2.
  crush.
Qed.

Hint Resolve Big_Val'.
```

Now we build some quite involved tactic support for reasoning about equalities over

PHOAS terms. First, we will call *equate_conj F* G to determine the consequences of an equality $F = \mathsf{G}$. When $F = \mathsf{f}$ *e_1 ... e_n* and $\mathsf{G} = \mathsf{f}$ *e'_1 ... e'_n*, *equate_conj* will return a conjunction $e\_1 = e'\_1 \wedge ... \wedge e\_n = e'\_n$. We hardcode a pattern for each value of *n* from 1 to 5.

```
Ltac equate_conj F G :=
  match constr:(F, G) with
    | (_ ?x1, _ ?x2) ⇒ constr:(x1 = x2)
    | (_ ?x1 ?y1, _ ?x2 ?y2) ⇒ constr:(x1 = x2 ∧ y1 = y2)
    | (_ ?x1 ?y1 ?z1, _ ?x2 ?y2 ?z2) ⇒ constr:(x1 = x2 ∧ y1 = y2 ∧ z1 = z2)
    | (_ ?x1 ?y1 ?z1 ?u1, _ ?x2 ?y2 ?z2 ?u2) ⇒
      constr:(x1 = x2 ∧ y1 = y2 ∧ z1 = z2 ∧ u1 = u2)
    | (_ ?x1 ?y1 ?z1 ?u1 ?v1, _ ?x2 ?y2 ?z2 ?u2 ?v2) ⇒
      constr:(x1 = x2 ∧ y1 = y2 ∧ z1 = z2 ∧ u1 = u2 ∧ v1 = v2)
  end.
```

The main tactic is *my_crush*, which generalizes our earlier *my_crush'* by performing inversion on hypotheses that equate PHOAS terms. Coq's built-in `inversion` is only designed to be useful on equalities over inductive types. PHOAS terms are functions, so `inversion` is not very helpful on them. To perform the equivalent of `discriminate`, we instantiate the terms with *var* as `fun _ ⇒ unit` and then appeal to normal `discriminate`. This eliminates some contradictory cases. To perform the equivalent of `injection`, we must consider all possible *var* instantiations. Some fairly intricate logic strings together these elements. The details are not worth discussing, since our conclusion will be that one should avoid dealing with proofs of facts like this one.

```
Ltac my_crush :=
  my_crush';
  repeat (match goal with
            | [ H : ?F = ?G ⊢ _ ] ⇒
              (let H' := fresh "H'" in
                assert (H' : F (fun _ ⇒ unit) = G (fun _ ⇒ unit)); [ congruence
                  | discriminate || injection H'; clear H' ];
                my_crush';
                repeat match goal with
                         | [ H : context[fun _ ⇒ unit] ⊢ _ ] ⇒ clear H
                       end;
                match type of H with
                  | ?F = ?G ⇒
                    let ec := equate_conj F G in
                      let var := fresh "var" in
                        assert ec; [ intuition; unfold Exp; apply ext_eq; intro var;
                          assert (H' : F var = G var); try congruence;
                            match type of H' with
                              | ?X = ?Y ⇒
```

```
                                    let X := eval hnf in X in
                                      let Y := eval hnf in Y in
                                        change (X = Y) in H'
                                  end; injection H'; my_crush'; tauto
                                | intuition; subst ]
                    end);
                clear H
            end; my_crush');
  my_crush'.
```

With that complicated tactic available, the proof of the main lemma is straightforward.

```
Lemma Multi_Big' : ∀ t (E E' : Exp t),
  E ==> E'
  → ∀ E'', E' ===> E''
    → E ===> E''.
  induction 1; crush; eauto;
    match goal with
      | [ H : _ ===> _ ⊢ _ ] ⇒ inversion H; my_crush; eauto
    end;
    match goal with
      | [ H : isCtx _ ⊢ _ ] ⇒ inversion H; my_crush; eauto
    end.
Qed.

Hint Resolve Multi_Big'.
```

The other direction of the overall equivalence follows as an easy corollary.

```
Theorem Multi_Big : ∀ t (E V : Exp t),
  E ==>* V
  → Val V
  → E ===> V.
  induction 1; crush; eauto.
Qed.
```

The lesson here is that working directly with PHOAS terms can easily lead to extremely intricate proofs. It is usually a better idea to stick to inductive proofs about *instantiated* PHOAS terms; in the case of this example, that means proofs about exp instead of Exp. Such results can usually be wrapped into results about Exp without further induction. Different theorems demand different variants of this underlying advice, and we will consider several of them in the chapters to come.

Chapter 18

Type-Theoretic Interpreters

Throughout this book, we have given semantics for programming languages via executable interpreters written in Gallina. PHOAS is quite compatible with that model, when we want to formalize many of the wide variety of interesting non-Turing-complete programming languages. Most such languages have very straightforward elaborations into Gallina. In this chapter, we show how to extend our past approach to higher-order languages encoded with PHOAS, and we show how simple program transformations may be proved correct with respect to these elaborative semantics.

18.1 Simply-Typed Lambda Calculus

We begin with a copy of last chapter's encoding of the syntax of simply-typed lambda calculus with natural numbers and addition. The primes at the ends of constructor names are gone, since here our primary subject is exps instead of Exps.

```
Module STLC.
  Inductive type : Type :=
  | Nat : type
  | Arrow : type → type → type.

  Infix "->" := Arrow (right associativity, at level 60).

  Section vars.
    Variable var : type → Type.

    Inductive exp : type → Type :=
    | Var : ∀ t,
      var t
      → exp t

    | Const : nat → exp Nat
    | Plus : exp Nat → exp Nat → exp Nat
```

```
  | App : ∀ t1 t2,
    exp (t1 -> t2)
    → exp t1
    → exp t2
  | Abs : ∀ t1 t2,
    (var t1 → exp t2)
    → exp (t1 -> t2).
End vars.

Definition Exp t := ∀ var, exp var t.

Implicit Arguments Var [var t].
Implicit Arguments Const [var].
Implicit Arguments Plus [var].
Implicit Arguments App [var t1 t2].
Implicit Arguments Abs [var t1 t2].
```

The definitions that follow will be easier to read if we define some parsing notations for the constructors.

```
Notation "# v" := (Var v) (at level 70).

Notation "*) n" := (Const n) (at level 70).
Infix "+^" := Plus (left associativity, at level 79).

Infix "@" := App (left associativity, at level 77).
Notation "\ x , e" := (Abs (fun x ⇒ e)) (at level 78).
Notation "\ ! , e" := (Abs (fun _ ⇒ e)) (at level 78).
```

A few examples will be useful for testing the functions we will write.

```
Example zero : Exp Nat := fun _ ⇒ *)0.
Example one : Exp Nat := fun _ ⇒ *)1.
Example zpo : Exp Nat := fun _ ⇒ zero _ +^ one _.
Example ident : Exp (Nat -> Nat) := fun _ ⇒ \x, #x.
Example app_ident : Exp Nat := fun _ ⇒ ident _ @ zpo _.
Example app : Exp ((Nat -> Nat) -> Nat -> Nat) := fun _ ⇒ \f, \x, #f @ #x.
Example app_ident' : Exp Nat := fun _ ⇒ app _ @ ident _ @ zpo _.
```

To write our interpreter, we must first interpret object language types as meta language types.

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Nat ⇒ nat
    | t1 -> t2 ⇒ typeDenote t1 → typeDenote t2
  end.
```

The crucial trick of the expression interpreter is to represent variables using the typeDenote function. Due to limitations in Coq's syntax extension system, we cannot take advantage

of some of our notations when they appear in patterns, so, to be consistent, in patterns we avoid notations altogether.

```
Fixpoint expDenote t (e : exp typeDenote t) : typeDenote t :=
  match e with
    | Var _ v ⇒ v

    | Const n ⇒ n
    | Plus e1 e2 ⇒ expDenote e1 + expDenote e2

    | App _ _ e1 e2 ⇒ (expDenote e1) (expDenote e2)
    | Abs _ _ e' ⇒ fun x ⇒ expDenote (e' x)
  end.

Definition ExpDenote t (e : Exp t) := expDenote (e _).
```

Some tests establish that ExpDenote produces Gallina terms like we might write manually.

```
Eval compute in ExpDenote zero.
    = 0
    : typeDenote Nat

Eval compute in ExpDenote one.
    = 1
    : typeDenote Nat

Eval compute in ExpDenote zpo.
    = 1
    : typeDenote Nat

Eval compute in ExpDenote ident.
    = fun x : nat ⇒ x
    : typeDenote (Nat -> Nat)

Eval compute in ExpDenote app_ident.
    = 1
    : typeDenote Nat

Eval compute in ExpDenote app.
    = fun (x : nat → nat) (x0 : nat) ⇒ x x0
    : typeDenote ((Nat -> Nat) -> Nat -> Nat)

Eval compute in ExpDenote app_ident'.
    = 1
    : typeDenote Nat
```

We can update to the higher-order case our common example of a constant folding function. The workhorse function cfold is parameterized to apply to an exp that uses any

var type. An output of cfold uses the same *var* type as the input. As in the definition of expDenote, we cannot use most of our notations in patterns, but we use them freely to make the bodies of `match` cases easier to read.

```
  Section cfold.
    Variable var : type → Type.

    Fixpoint cfold t (e : exp var t) : exp var t :=
      match e with
        | Var _ v ⇒ #v

        | Const n ⇒ ^n
        | Plus e1 e2 ⇒
          let e1' := cfold e1 in
          let e2' := cfold e2 in
          match e1', e2' return _ with
            | Const n1, Const n2 ⇒ ^(n1 + n2)
            | _, _ ⇒ e1' +^ e2'
          end

        | App _ _ e1 e2 ⇒ cfold e1 @ cfold e2
        | Abs _ _ e' ⇒ \x, cfold (e' x)
      end.
  End cfold.

  Definition Cfold t (E : Exp t) : Exp t := fun _ ⇒ cfold (E _).
```

Now we would like to prove the correctness of Cfold, which follows from a simple inductive lemma about cfold.

```
  Lemma cfold_correct : ∀ t (e : exp _ t),
    expDenote (cfold e) = expDenote e.
    induction e; crush; try (ext_eq; crush);
      repeat (match goal with
                | [ ⊢ context[cfold ?E] ] ⇒ dep_destruct (cfold E)
              end; crush).
  Qed.

  Theorem Cfold_correct : ∀ t (E : Exp t),
    ExpDenote (Cfold E) = ExpDenote E.
    unfold ExpDenote, Cfold; intros; apply cfold_correct.
  Qed.
End STLC.
```

18.2 Adding Products and Sums

The example is easily adapted to support products and sums, the basis of non-recursive datatypes in ML and Haskell.

```
Module PSLC.

  Inductive type : Type :=
  | Nat : type
  | Arrow : type → type → type
  | Prod : type → type → type
  | Sum : type → type → type.

  Infix "->" := Arrow (right associativity, at level 62).
  Infix "**" := Prod (right associativity, at level 61).
  Infix "++" := Sum (right associativity, at level 60).

  Section vars.
    Variable var : type → Type.

    Inductive exp : type → Type :=
    | Var : ∀ t,
      var t
      → exp t

    | Const : nat → exp Nat
    | Plus : exp Nat → exp Nat → exp Nat

    | App : ∀ t1 t2,
      exp (t1 -> t2)
      → exp t1
      → exp t2
    | Abs : ∀ t1 t2,
      (var t1 → exp t2)
      → exp (t1 -> t2)

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

```
  | Inl : ∀ t1 t2,
    exp t1
    → exp (t1 ++ t2)
  | Inr : ∀ t1 t2,
    exp t2
    → exp (t1 ++ t2)
  | SumCase : ∀ t1 t2 t,
    exp (t1 ++ t2)
    → (var t1 → exp t)
    → (var t2 → exp t)
    → exp t.
End vars.

Definition Exp t := ∀ var, exp var t.

Implicit Arguments Var [var t].
Implicit Arguments Const [var].
Implicit Arguments Abs [var t1 t2].
Implicit Arguments Inl [var t1 t2].
Implicit Arguments Inr [var t1 t2].

Notation "# v" := (Var v) (at level 70).

Notation "*) n" := (Const n) (at level 70).
Infix "+^" := Plus (left associativity, at level 78).

Infix "@" := App (left associativity, at level 77).
Notation "\ x , e" := (Abs (fun x ⇒ e)) (at level 78).
Notation "\ ! , e" := (Abs (fun _ ⇒ e)) (at level 78).

Notation "[ e1 , e2 ]" := (Pair e1 e2).
Notation "#1 e" := (Fst e) (at level 75).
Notation "#2 e" := (Snd e) (at level 75).

Notation "'case' e 'of' x ⇒ e1 | y ⇒ e2" := (SumCase e (fun x ⇒ e1) (fun y ⇒ e2))
  (at level 79).
```

A few examples can be defined easily, using the notations above.

```
Example swap : Exp (Nat ** Nat -> Nat ** Nat) := fun _ ⇒ \p, [#2 #p, #1 #p].
Example zo : Exp (Nat ** Nat) := fun _ ⇒ [*)0, *)1].
Example swap_zo : Exp (Nat ** Nat) := fun _ ⇒ swap _ @ zo _.

Example natOut : Exp (Nat ++ Nat -> Nat) := fun _ ⇒
  \s, case #s of x ⇒ #x | y ⇒ #y +^ #y.
Example ns1 : Exp (Nat ++ Nat) := fun _ ⇒ Inl (*)3).
Example ns2 : Exp (Nat ++ Nat) := fun _ ⇒ Inr (*)5).
Example natOut_ns1 : Exp Nat := fun _ ⇒ natOut _ @ ns1 _.
Example natOut_ns2 : Exp Nat := fun _ ⇒ natOut _ @ ns2 _.
```

The semantics adapts without incident.

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Nat ⇒ nat
    | t1 -> t2 ⇒ typeDenote t1 → typeDenote t2
    | t1 ** t2 ⇒ typeDenote t1 * typeDenote t2
    | t1 ++ t2 ⇒ typeDenote t1 + typeDenote t2
  end%type.

Fixpoint expDenote t (e : exp typeDenote t) : typeDenote t :=
  match e with
    | Var _ v ⇒ v

    | Const n ⇒ n
    | Plus e1 e2 ⇒ expDenote e1 + expDenote e2

    | App _ _ e1 e2 ⇒ (expDenote e1) (expDenote e2)
    | Abs _ _ e' ⇒ fun x ⇒ expDenote (e' x)

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

    | Inl _ _ e' ⇒ inl _ (expDenote e')
    | Inr _ _ e' ⇒ inr _ (expDenote e')
    | SumCase _ _ _ e' e1 e2 ⇒
      match expDenote e' with
        | inl v ⇒ expDenote (e1 v)
        | inr v ⇒ expDenote (e2 v)
      end
  end.

Definition ExpDenote t (e : Exp t) := expDenote (e _).

Eval compute in ExpDenote swap.
    = fun x : nat * nat ⇒ (let (_, y) := x in y, let (x0, _) := x in x0)
    : typeDenote (Nat ** Nat -> Nat ** Nat)

Eval compute in ExpDenote zo.
    = (0, 1)
    : typeDenote (Nat ** Nat)

Eval compute in ExpDenote swap_zo.
    = (1, 0)
    : typeDenote (Nat ** Nat)
```

```
Eval cbv beta iota delta -[plus] in ExpDenote natOut.
    = fun x : nat + nat ⇒ match x with
                          | inl v ⇒ v
                          | inr v ⇒ v + v
                          end
    : typeDenote (Nat ++ Nat -> Nat)

Eval compute in ExpDenote ns1.
    = inl nat 3
    : typeDenote (Nat ++ Nat)

Eval compute in ExpDenote ns2.
    = inr nat 5
    : typeDenote (Nat ++ Nat)

Eval compute in ExpDenote natOut_ns1.
    = 3
    : typeDenote Nat

Eval compute in ExpDenote natOut_ns2.
    = 10
    : typeDenote Nat
```

We adapt the cfold function using the same basic dependent-types trick that we applied in an earlier chapter to a very similar function for a language without variables.

```
Section cfold.
  Variable var : type → Type.

  Definition pairOutType t :=
    match t return Type with
      | t1 ** t2 ⇒ option (exp var t1 * exp var t2)
      | _ ⇒ unit
    end.

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

  Definition pairOut t1 t2 (e : exp var (t1 ** t2))
    : option (exp var t1 * exp var t2) :=
    match e in exp _ t return pairOutType t with
      | Pair _ _ e1 e2 ⇒ Some (e1, e2)
      | _ ⇒ pairOutDefault _
    end.

  Fixpoint cfold t (e : exp var t) : exp var t :=
```

```
    match e with
      | Var _ v ⇒ #v

      | Const n ⇒ ^n
      | Plus e1 e2 ⇒
        let e1' := cfold e1 in
        let e2' := cfold e2 in
        match e1', e2' return _ with
          | Const n1, Const n2 ⇒ ^(n1 + n2)
          | _, _ ⇒ e1' +^ e2'
        end

      | App _ _ e1 e2 ⇒ cfold e1 @ cfold e2
      | Abs _ _ e' ⇒ \x, cfold (e' x)

      | Pair _ _ e1 e2 ⇒ [cfold e1, cfold e2]
      | Fst t1 _ e' ⇒
        let e'' := cfold e' in
          match pairOut e'' with
            | None ⇒ #1 e''
            | Some (e1, _) ⇒ e1
          end
      | Snd _ _ e' ⇒
        let e'' := cfold e' in
          match pairOut e'' with
            | None ⇒ #2 e''
            | Some (_, e2) ⇒ e2
          end

      | Inl _ _ e' ⇒ Inl (cfold e')
      | Inr _ _ e' ⇒ Inr (cfold e')
      | SumCase _ _ _ e' e1 e2 ⇒
        case cfold e' of
          x ⇒ cfold (e1 x)
        | y ⇒ cfold (e2 y)
    end.
End cfold.

Definition Cfold t (E : Exp t) : Exp t := fun _ ⇒ cfold (E _).
```

The proofs are almost as straightforward as before. We first establish two simple theorems about pairs and their projections.

```
Section pairs.
  Variables A B : Type.
```

```
    Variable v1 : A.
    Variable v2 : B.
    Variable v : A * B.

    Theorem pair_eta1 : (v1, v2) = v → v1 = fst v.
      destruct v; crush.
    Qed.

    Theorem pair_eta2 : (v1, v2) = v → v2 = snd v.
      destruct v; crush.
    Qed.
  End pairs.

  Hint Resolve pair_eta1 pair_eta2.
```

To the proof script for the main lemma, we add just one more `match` case, detecting when case analysis is appropriate on discriminees of matches over sum types.

```
  Lemma cfold_correct : ∀ t (e : exp _ t),
    expDenote (cfold e) = expDenote e.
    induction e; crush; try (ext_eq; crush);
      repeat (match goal with
                | [ ⊢ context[cfold ?E] ] ⇒ dep_destruct (cfold E)
                | [ ⊢ match ?E with inl _ ⇒ _ | inr _ ⇒ _ end = _ ] ⇒ destruct E
              end; crush); eauto.
  Qed.

  Theorem Cfold_correct : ∀ t (E : Exp t),
    ExpDenote (Cfold E) = ExpDenote E.
    unfold ExpDenote, Cfold; intros; apply cfold_correct.
  Qed.
End PSLC.
```

Chapter 19

Extensional Transformations

Last chapter's constant folding example was particularly easy to verify, because that transformation used the same source and target language. In this chapter, we verify a different translation, illustrating the added complexities in translating between languages.

Program transformations can be classified as *intensional*, when they require some notion of inequality between variables; or *extensional*, otherwise. This chapter's example is extensional, and the next chapter deals with the trickier intensional case.

19.1 CPS Conversion for Simply-Typed Lambda Calculus

A convenient method for compiling functional programs begins with conversion to *continuation-passing style*, or CPS. In this restricted form, function calls never return; instead, we pass explicit return pointers, much as in assembly language. Additionally, we make order of evaluation explicit, breaking complex expressions into sequences of primitive operations.

Our translation will operate over the same source language that we used in the first part of last chapter, so we omit most of the language definition. However, we do make one significant change: since we will be working with multiple languages that involve similar constructs, we use Coq's *notation scope* mechanism to disambiguate. For instance, the span of code dealing with type notations looks like this:

```
Notation "'Nat'" := TNat : source_scope.
Infix "->" := Arrow (right associativity, at level 60) : source_scope.

Open Scope source_scope.
Bind Scope source_scope with type.
Delimit Scope source_scope with source.
```

We explicitly place our notations inside a scope named *source_scope*, and we associate a delimiting key *source* with *source_scope*. Without further commands, our notations would only be used in expressions like (...)%source. We also open our scope locally within this

module, so that we avoid repeating %source in many places. Further, we *bind* our scope to **type**. In some circumstances where Coq is able to infer that some subexpression has type **type**, that subexpression will automatically be parsed in *source_scope*.

The other critical new ingredient is a generalization of the **Closed** relation from two chapters ago. The new relation **exp_equiv** characters when two expressions may be considered syntactically equal. We need to be able to handle cases where each expression uses a different *var* type. Intuitively, we will want to compare expressions that use their variables to store source-level and target-level values. We express pairs of equivalent variables using a list parameter to the relation; variable expressions will be considered equivalent if and only if their variables belong to this list. The rule for function abstraction extends the list in a higher-order way. The remaining rules just implement the obvious congruence over expressions.

```
Section exp_equiv.
  Variables var1 var2 : type → Type.

  Inductive exp_equiv : list { t : type & var1 t * var2 t }%type
    → ∀ t, exp var1 t → exp var2 t → Prop :=
  | EqVar : ∀ G t (v1 : var1 t) v2,
    In (existT _ t (v1, v2)) G
    → exp_equiv G (#v1) (#v2)

  | EqConst : ∀ G n,
    exp_equiv G (^n) (^n)
  | EqPlus : ∀ G x1 y1 x2 y2,
    exp_equiv G x1 x2
    → exp_equiv G y1 y2
    → exp_equiv G (x1 +^ y1) (x2 +^ y2)

  | EqApp : ∀ G t1 t2 (f1 : exp _ (t1 -> t2)) (x1 : exp _ t1) f2 x2,
    exp_equiv G f1 f2
    → exp_equiv G x1 x2
    → exp_equiv G (f1 @ x1) (f2 @ x2)
  | EqAbs : ∀ G t1 t2 (f1 : var1 t1 → exp var1 t2) f2,
    (∀ v1 v2, exp_equiv (existT _ t1 (v1, v2) :: G) (f1 v1) (f2 v2))
    → exp_equiv G (Abs f1) (Abs f2).
End exp_equiv.
```

It turns out that, for any parametric expression E, any two instantiations of E with particular *var* types must be equivalent, with respect to an empty variable list. The parametricity of Gallina guarantees this, in much the same way that it guaranteed the truth of the axiom about **Closed**. Thus, we assert an analogous axiom here.

```
Axiom Exp_equiv : ∀ t (E : Exp t) var1 var2,
  exp_equiv nil (E var1) (E var2).
```

```
End Source.
```

Now we need to define the CPS language, where binary function types are replaced with unary continuation types, and we add product types because they will be useful in our translation.

```
Module CPS.
  Inductive type : Type :=
  | TNat : type
  | Cont : type → type
  | Prod : type → type → type.

  Notation "'Nat'" := TNat : cps_scope.
  Notation "t —>" := (Cont t) (at level 61) : cps_scope.
  Infix "**" := Prod (right associativity, at level 60) : cps_scope.

  Bind Scope cps_scope with type.
  Delimit Scope cps_scope with cps.

  Section vars.
    Variable var : type → Type.
```

A CPS program is a series of bindings of primitive operations (primops), followed by either a halt with a final program result or by a call to a continuation. The arguments to these program-ending operations are enforced to be variables. To use the values of compound expressions instead, those expressions must be decomposed into bindings of primops. The primop language itself similarly forces variables for all arguments besides bodies of function abstractions.

```
    Inductive prog : Type :=
    | PHalt :
      var Nat
      → prog
    | App : ∀ t,
      var (t —>)
      → var t
      → prog
    | Bind : ∀ t,
      primop t
      → (var t → prog)
      → prog

    with primop : type → Type :=
    | Const : nat → primop Nat
    | Plus : var Nat → var Nat → primop Nat

    | Abs : ∀ t,
      (var t → prog)
```

```
      → primop (t —>)

  | Pair : ∀ t1 t2,
      var t1
      → var t2
      → primop (t1 ** t2)
  | Fst : ∀ t1 t2,
      var (t1 ** t2)
      → primop t1
  | Snd : ∀ t1 t2,
      var (t1 ** t2)
      → primop t2.
End vars.

Implicit Arguments PHalt [var].
Implicit Arguments App [var t].

Implicit Arguments Const [var].
Implicit Arguments Plus [var].
Implicit Arguments Abs [var t].
Implicit Arguments Pair [var t1 t2].
Implicit Arguments Fst [var t1 t2].
Implicit Arguments Snd [var t1 t2].

Notation "'Halt' x" := (PHalt x) (no associativity, at level 75) : cps_scope.
Infix "@@" := App (no associativity, at level 75) : cps_scope.
Notation "x ← p ; e" := (Bind p (fun x ⇒ e))
  (right associativity, at level 76, p at next level) : cps_scope.
Notation "! ← p ; e" := (Bind p (fun _ ⇒ e))
  (right associativity, at level 76, p at next level) : cps_scope.

Notation "*) n" := (Const n) (at level 70) : cps_scope.
Infix "+^" := Plus (left associativity, at level 79) : cps_scope.

Notation "\ x , e" := (Abs (fun x ⇒ e)) (at level 78) : cps_scope.
Notation "\ ! , e" := (Abs (fun _ ⇒ e)) (at level 78) : cps_scope.

Notation "[ x1 , x2 ]" := (Pair x1 x2) : cps_scope.
Notation "#1 x" := (Fst x) (at level 72) : cps_scope.
Notation "#2 x" := (Snd x) (at level 72) : cps_scope.

Bind Scope cps_scope with prog primop.

Open Scope cps_scope.
```

In interpreting types, we treat continuations as functions with codomain **nat**, choosing **nat** as our arbitrary program result type.

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

```
    | Nat ⇒ nat
    | t' —> ⇒ typeDenote t' → nat
    | t1 ** t2 ⇒ (typeDenote t1 * typeDenote t2)%type
  end.
```

A mutually-recursive definition establishes the meanings of programs and primops.

```
Fixpoint progDenote (e : prog typeDenote) : nat :=
  match e with
    | PHalt n ⇒ n
    | App _ f x ⇒ f x
    | Bind _ p x ⇒ progDenote (x (primopDenote p))
  end

with primopDenote t (p : primop typeDenote t) : typeDenote t :=
  match p with
    | Const n ⇒ n
    | Plus n1 n2 ⇒ n1 + n2

    | Abs _ e ⇒ fun x ⇒ progDenote (e x)

    | Pair _ _ v1 v2 ⇒ (v1, v2)
    | Fst _ _ v ⇒ fst v
    | Snd _ _ v ⇒ snd v
  end.

Definition Prog := ∀ var, prog var.
Definition Primop t := ∀ var, primop var t.
Definition ProgDenote (E : Prog) := progDenote (E _).
Definition PrimopDenote t (P : Primop t) := primopDenote (P _).
End CPS.

Import Source CPS.
```

The translation itself begins with a type-level compilation function. We change every function into a continuation whose argument is a pair, consisting of the translation of the original argument and of an explicit return pointer.

```
Fixpoint cpsType (t : Source.type) : CPS.type :=
  match t with
    | Nat ⇒ Nat%cps
    | t1 -> t2 ⇒ (cpsType t1 ** (cpsType t2 —>) —>)%cps
  end%source.
```

Now we can define the expression translation. The notation $x \longleftarrow e1; e2$ stands for translating source-level expression *e1*, binding x to the CPS-level result of running the translated program, and then evaluating CPS-level expression *e2* in that context.

```
Reserved Notation "x ⟵ e1 ; e2" (right associativity, at level 76, e1 at next level).
```

```
Section cpsExp.
  Variable var : CPS.type → Type.

  Import Source.
  Open Scope cps_scope.
```

We implement a well-known variety of higher-order, one-pass CPS translation. The translation cpsExp is parameterized not only by the expression e to translate, but also by a meta-level continuation. The idea is that cpsExp evaluates the translation of e and calls the continuation on the result. With this convention, cpsExp itself is a natural match for the notation we just reserved.

```
  Fixpoint cpsExp t (e : exp (fun t ⇒ var (cpsType t)) t)
    : (var (cpsType t) → prog var) → prog var :=
    match e with
      | Var _ v ⇒ fun k ⇒ k v

      | Const n ⇒ fun k ⇒
        x ← ^n;
        k x
      | Plus e1 e2 ⇒ fun k ⇒
        x1 ⟵ e1;
        x2 ⟵ e2;
        x ← x1 +^ x2;
        k x

      | App _ _ e1 e2 ⇒ fun k ⇒
        f ⟵ e1;
        x ⟵ e2;
        kf ← \r, k r;
        p ← [x, kf];
        f @@ p
      | Abs _ _ e' ⇒ fun k ⇒
        f ← CPS.Abs (var := var) (fun p ⇒
          x ← #1 p;
          kf ← #2 p;
          r ⟵ e' x;
          kf @@ r);
        k f
    end

    where "x ⟵ e1 ; e2" := (cpsExp e1 (fun x ⇒ e2)).
End cpsExp.
```

Since notations do not survive the closing of sections, we redefine the notation associated

with cpsExp.

```
Notation "x ⟵ e1 ; e2" := (cpsExp e1 (fun x ⇒ e2)) : cps_scope.

Implicit Arguments cpsExp [var t].
```

We wrap cpsExp into the parametric version CpsExp, passing an always-halt continuation at the root of the recursion.

```
Definition CpsExp (E : Exp Nat) : Prog :=
  fun _ ⇒ cpsExp (E _) (PHalt (var := _)).

Eval compute in CpsExp zero.
     = fun var : type → Type ⇒ x ← *)0; Halt x
     : Prog

Eval compute in CpsExp one.
     = fun var : type → Type ⇒ x ← *)1; Halt x
     : Prog

Eval compute in CpsExp zpo.
     = fun var : type → Type ⇒ x ← *)0; x0 ← *)1; x1 ← (x +^ x0); Halt x1
     : Prog

Eval compute in CpsExp app_ident.
     = fun var : type → Type ⇒
       f ← (\ p, x ← #1 p; kf ← #2 p; kf @@ x);
       x ← *)0;
       x0 ← *)1; x1 ← (x +^ x0); kf ← (\ r, Halt r); p ← [x1, kf]; f @@ p
     : Prog

Eval compute in CpsExp app_ident'.
     = fun var : type → Type ⇒
       f ←
       (\ p,
        x ← #1 p;
        kf ← #2 p;
        f ←
        (\ p0,
         x0 ← #1 p0;
         kf0 ← #2 p0; kf1 ← (\ r, kf0 @@ r); p1 ← [x0, kf1]; x @@ p1);
        kf @@ f);
       f0 ← (\ p, x ← #1 p; kf ← #2 p; kf @@ x);
       kf ←
       (\ r,
        x ← *)0;
        x0 ← *)1;
        x1 ← (x +^ x0); kf ← (\ r0, Halt r0); p ← [x1, kf]; r @@ p);
```

```
    p ← [f0, kf]; f @@ p
  : Prog

Eval compute in ProgDenote (CpsExp zero).
  = 0
  : nat

Eval compute in ProgDenote (CpsExp one).
  = 1
  : nat

Eval compute in ProgDenote (CpsExp zpo).
  = 1
  : nat

Eval compute in ProgDenote (CpsExp app_ident).
  = 1
  : nat

Eval compute in ProgDenote (CpsExp app_ident').
  = 1
  : nat
```

Our main inductive lemma about cpsExp needs a notion of compatibility between source-level and CPS-level values. We express compatibility with a *logical relation*; that is, we define a binary relation by recursion on type structure, and the function case of the relation considers functions related if they map related arguments to related results. In detail, the function case is slightly more complicated, since it must deal with our continuation-based calling convention.

```
Fixpoint lr (t : Source.type)
  : Source.typeDenote t → CPS.typeDenote (cpsType t) → Prop :=
  match t with
    | Nat ⇒ fun n1 n2 ⇒ n1 = n2
    | t1 -> t2 ⇒ fun f1 f2 ⇒
      ∀ x1 x2, lr _ x1 x2
        → ∀ k, ∃ r,
          f2 (x2, k) = k r
          ∧ lr _ (f1 x1) r
  end%source.
```

The main lemma is now easily stated and proved. The most surprising aspect of the statement is the presence of *two* versions of the expression to be compiled. The first, *e1*, uses a *var* choice that makes it a suitable argument to expDenote. The second expression, *e2*, uses a *var* choice that makes its compilation, cpsExp *e2 k*, a suitable argument to progDenote. We use **exp_equiv** to assert that *e1* and *e2* have the same underlying structure, up to a variable correspondence list G. A hypothesis about G ensures that all of its pairs of variables

belong to the logical relation lr. We also use lr, in concert with some quantification over continuations and program results, in the conclusion of the lemma.

The lemma's proof should be unsurprising by now. It uses our standard bag of Ltac tricks to help out with quantifier instantiation; *crush* and `eauto` can handle the rest.

```
Lemma cpsExp_correct : ∀ G t (e1 : exp _ t) (e2 : exp _ t),
  exp_equiv G e1 e2
  → (∀ t v1 v2, lr (existT _ t (v1, v2)) G → lr t v1 v2)
  → ∀ k, ∃ r,
    progDenote (cpsExp e2 k) = progDenote (k r)
    ∧ lr t (expDenote e1) r.
  induction 1; crush;
    repeat (match goal with
              | [ H : ∀ k, ∃ r, progDenote (cpsExp ?E k) = _ ∧ _
                  ⊢ context[cpsExp ?E ?K] ] ⇒
                generalize (H K); clear H
              | [ ⊢ ∃ r, progDenote (_ ?R) = progDenote (_ r) ∧ _ ] ⇒
                ∃ R
              | [ t1 : Source.type ⊢ _ ] ⇒
                match goal with
                  | [ Hlr : lr t1 ?X1 ?X2, IH : ∀ v1 v2, _ ⊢ _ ] ⇒
                    generalize (IH X1 X2); clear IH; intro IH;
                      match type of IH with
                        | ?P → _ ⇒ assert P
                      end
                end
            end; crush); eauto.
Qed.
```

A simple lemma establishes the degenerate case of cpsExp_correct's hypothesis about G.

```
Lemma vars_easy : ∀ t v1 v2,
  lr (existT (fun t0 ⇒ (Source.typeDenote t0 * typeDenote (cpsType t0))%type) t
    (v1, v2)) nil → lr t v1 v2.
  crush.
Qed.
```

A manual application of cpsExp_correct proves a version applicable to CpsExp. This is where we use the axiom *Exp_equiv*.

```
Theorem CpsExp_correct : ∀ (E : Exp Nat),
  ProgDenote (CpsExp E) = ExpDenote E.
  unfold ProgDenote, CpsExp, ExpDenote; intros;
    generalize (cpsExp_correct (e1 := E _) (e2 := E _)
      (Exp_equiv _ _ _) vars_easy (PHalt (var := _))); crush.
Qed.
```

19.2 Exercises

1. When in the last chapter we implemented constant folding for simply-typed lambda calculus, it may have seemed natural to try applying beta reductions. This would have been a lot more trouble than is apparent at first, because we would have needed to convince Coq that our normalizing function always terminated.

   It might also seem that beta reduction is a lost cause because we have no effective way of substituting in the `exp` type; we only managed to write a substitution function for the parametric `Exp` type. This is not as big of a problem as it seems. For instance, for the language we built by extending simply-typed lambda calculus with products and sums, it also appears that we need substitution for simplifying `case` expressions whose discriminees are known to be `inl` or `inr`, but the function is still implementable.

   For this exercise, extend the products and sums constant folder from the last chapter so that it simplifies `case` expressions as well, by checking if the discriminee is a known `inl` or known `inr`. Also extend the correctness theorem to apply to your new definition. You will probably want to assert an axiom relating to an expression equivalence relation like the one defined in this chapter. Any such axiom should only mention syntax; it should not mention any compilation or denotation functions. Following the format of the axiom from the last chapter is the safest bet to avoid proving a worthless theorem.

Chapter 20

Intensional Transformations

The essential benefit of higher-order encodings is that variable contexts are implicit. We represent the object language's contexts within the meta language's contexts. For translations like CPS conversion, this is a clear win, since such translations do not need to keep track of details of which variables are available. Other important translations, including closure conversion, need to work with variables as first-class, analyzable values.

Another example is conversion from PHOAS terms to de Bruijn terms. The output format makes the structure of variables explicit, so the translation requires explicit reasoning about variable identity. In this chapter, we implement verified translations in both directions between last chapter's PHOAS language and a de Bruijn version of it. Along the way, we show one approach to avoiding the use of axioms with PHOAS.

The de Bruijn version of simply-typed lambda calculus is defined in a manner that should be old hat by now.

```
Module DeBruijn.
  Inductive exp : list type → type → Type :=
    | Var : ∀ G t,
      member t G
      → exp G t

    | Const : ∀ G, nat → exp G Nat
    | Plus : ∀ G, exp G Nat → exp G Nat → exp G Nat

    | App : ∀ G t1 t2,
      exp G (t1 -> t2)
      → exp G t1
      → exp G t2
    | Abs : ∀ G t1 t2,
      exp (t1 :: G) t2
      → exp G (t1 -> t2).
  Implicit Arguments Const [G].
```

```
  Fixpoint expDenote G t (e : exp G t) : hlist typeDenote G → typeDenote t :=
    match e with
      | Var _ _ v ⇒ fun s ⇒ hget s v

      | Const _ n ⇒ fun _ ⇒ n
      | Plus _ e1 e2 ⇒ fun s ⇒ expDenote e1 s + expDenote e2 s

      | App _ _ _ e1 e2 ⇒ fun s ⇒ (expDenote e1 s) (expDenote e2 s)
      | Abs _ _ _ e' ⇒ fun s x ⇒ expDenote e' (x ::: s)
    end.
End DeBruijn.

Import Phoas DeBruijn.
```

20.1 From De Bruijn to PHOAS

The heart of the translation into PHOAS is this function phoasify, which is parameterized by an **hlist** that represents a mapping from de Bruijn variables to PHOAS variables.

```
Section phoasify.
  Variable var : type → Type.

  Fixpoint phoasify G t (e : DeBruijn.exp G t) : hlist var G → Phoas.exp var t :=
    match e with
      | Var _ _ v ⇒ fun s ⇒ #(hget s v)

      | Const _ n ⇒ fun _ ⇒ ^n
      | Plus _ e1 e2 ⇒ fun s ⇒ phoasify e1 s +^ phoasify e2 s

      | App _ _ _ e1 e2 ⇒ fun s ⇒ phoasify e1 s @ phoasify e2 s
      | Abs _ _ _ e' ⇒ fun s ⇒ \x, phoasify e' (x ::: s)
    end.
End phoasify.

Definition Phoasify t (e : DeBruijn.exp nil t) : Phoas.Exp t :=
  fun _ ⇒ phoasify e HNil.
```

It turns out to be trivial to establish the translation's soundness.

```
Theorem phoasify_sound : ∀ G t (e : DeBruijn.exp G t) s,
  Phoas.expDenote (phoasify e s) = DeBruijn.expDenote e s.
  induction e; crush; ext_eq; crush.
Qed.
```

We can prove that any output of Phoasify is well-formed, in a sense strong enough to let us avoid asserting last chapter's axiom.

```
Print Wf.
```

```
Wf =
fun (t : type) (E : Exp t) ⇒
∀ var1 var2 : type → Type, exp_equiv nil (E var1) (E var2)
    : ∀ t : type, Exp t → Prop

Section vars.
  Variables var1 var2 : type → Type.
```

In the course of proving well-formedness, we will need to translate back and forth between the de Bruijn and PHOAS representations of free variable information. The function zip combines two de Bruijn substitutions into a single PHOAS context.

```
  Fixpoint zip G (s1 : hlist var1 G)
    : hlist var2 G → list {t : type & var1 t * var2 t}%type :=
    match s1 with
      | HNil ⇒ fun _ ⇒ nil
      | HCons _ _ v1 s1' ⇒ fun s2 ⇒ existT _ _ (v1, hhd s2) :: zip s1' (htl s2)
    end.
```

Two simple lemmas about zip will make useful hints.

```
  Lemma In_zip : ∀ t G (s1 : hlist _ G) s2 (m : member t G),
    In (existT _ t (hget s1 m, hget s2 m)) (zip s1 s2).
    induction s1; intro s2; dep_destruct s2; intro m; dep_destruct m; crush.
  Qed.

  Lemma unsimpl_zip : ∀ t (v1 : var1 t) (v2 : var2 t)
    G (s1 : hlist _ G) s2 t' (e1 : Phoas.exp _ t') e2,
   exp_equiv (zip (v1 ::: s1) (v2 ::: s2)) e1 e2
   → exp_equiv (existT _ _ (v1, v2) :: zip s1 s2) e1 e2.
    trivial.
  Qed.

  Hint Resolve In_zip unsimpl_zip.
```

Now it is trivial to prove the main inductive lemma about well-formedness.

```
  Lemma phoasify_wf : ∀ G t (e : DeBruijn.exp G t) s1 s2,
    exp_equiv (zip s1 s2) (phoasify e s1) (phoasify e s2).
    Hint Constructors exp_equiv.

    induction e; crush.
  Qed.
End vars.
```

We apply phoasify_wf manually to prove the final theorem.

```
Theorem Phoasify_wf : ∀ t (e : DeBruijn.exp nil t),
  Wf (Phoasify e).
  unfold Wf, Phoasify; intros;
```

```
    apply (phoasify_wf e (HNil (B := var1)) (HNil (B := var2))).
Qed.
```

Now, if we compose Phoasify with any translation over PHOAS terms, we can verify the composed translation without relying on axioms. The conclusion of Phoasify_wf is robustly useful in verifying a wide variety of translations that use a wide variety of *var* instantiations.

20.2 From PHOAS to De Bruijn

The translation to de Bruijn terms is more involved. We will essentially be instantiating and using a PHOAS term following a convention isomorphic to *de Bruijn levels*, which are different from the de Bruijn indices that we have treated so far. With levels, a given bound variable is referred to by the same number at each of its occurrences. In any expression, the binders that are not enclosed by other binders are assigned level 0, a binder with just one enclosing binder is assigned level 1, and so on. The uniformity of references to any binder will be critical to our translation, since it is compatible with the pattern of filling in all of a PHOAS variable's locations at once by applying a function.

We implement a special lookup function, for reading a numbered variable's type out of a de Bruijn level typing context. The last variable in the list is taken to have level 0, the next-to-last level 1, and so on.

```
Fixpoint lookup (ts : list type) (n : nat) : option type :=
  match ts with
    | nil ⇒ None
    | t :: ts' ⇒ if eq_nat_dec n (length ts') then Some t else lookup ts' n
  end.

Infix "##" := lookup (left associativity, at level 1).
```

With lookup, we can define a notion of well-formedness for PHOAS expressions that we are treating according to the de Bruijn level convention.

```
Fixpoint wf (ts : list type) t (e : Phoas.exp (fun _ ⇒ nat) t) : Prop :=
  match e with
    | Phoas.Var t n ⇒ ts ## n = Some t
    | Phoas.Const _ ⇒ True
    | Phoas.Plus e1 e2 ⇒ wf ts e1 ∧ wf ts e2
    | Phoas.App _ _ e1 e2 ⇒ wf ts e1 ∧ wf ts e2
    | Phoas.Abs t1 _ e1 ⇒ wf (t1 :: ts) (e1 (length ts))
  end.
```

20.2.1 Connecting Notions of Well-Formedness

Our first order of business now is to prove that any well-formed Exp instantiates to a well-formed de Bruijn level expression. We start by characterizing, as a function of de Bruijn

level contexts, the set of PHOAS contexts that will occur in the proof, where we will be inducting over an **exp_equiv** derivation.

```
Fixpoint makeG (ts : list type) : list { t : type & nat * nat }%type :=
  match ts with
    | nil ⇒ nil
    | t :: ts' ⇒ existT _ t (length ts', length ts') :: makeG ts'
  end.
```

Now we prove a connection between lookup and makeG, by way of a lemma about lookup.

```
Opaque eq_nat_dec.
Hint Extern 1 (_ ≥ _) ⇒ omega.

Lemma lookup_contra' : ∀ t ts n,
  ts ## n = Some t
  → n ≥ length ts
  → False.
  induction ts; crush;
    match goal with
      | [ _ : context[if ?E then _ else _] ⊢ _ ] ⇒ destruct E; crush
    end; eauto.
Qed.

Lemma lookup_contra : ∀ t ts,
  ts ## (length ts) = Some t
  → False.
  intros; eapply lookup_contra'; eauto.
Qed.

Hint Resolve lookup_contra.

Lemma lookup_In : ∀ t v1 v2 ts,
  In (existT (fun _ : type ⇒ (nat * nat)%type) t (v1, v2)) (makeG ts)
  → ts ## v1 = Some t.
  induction ts; crush;
    match goal with
      | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E; crush
    end; elimtype False; eauto.
Qed.

Hint Resolve lookup_In.
```

We can prove the main inductive lemma by induction over **exp_equiv** derivations.

```
Hint Extern 1 (_ :: _ = makeG (_ :: _)) ⇒ reflexivity.

Lemma Wf_wf' : ∀ G t e1 (e2 : Phoas.exp (fun _ ⇒ nat) t),
  exp_equiv G e1 e2
  → ∀ ts, G = makeG ts
```

```
    → wf ts e1.
  induction 1; crush; eauto.
Qed.

Lemma Wf_wf : ∀ t (E : Exp t),
  Wf E
  → wf nil (E (fun _ ⇒ nat)).
  intros; eapply Wf_wf'; eauto.
Qed.
```

20.2.2 The Translation

Implementing the translation itself will require some proofs. Our main function dbify will take `wf` proofs as arguments, and these proofs will be critical to constructing de Bruijn index terms. First, we use `congruence` to prove two basic theorems about **option**s.

```
Theorem None_Some : ∀ T (x : T),
  None = Some x
  → False.
  congruence.
Qed.

Theorem Some_Some : ∀ T (x y : T),
  Some x = Some y
  → x = y.
  congruence.
Qed.
```

We can use these theorems to implement makeVar, which translates a proof about lookup into a de Bruijn index variable with a closely related type.

```
Fixpoint makeVar {ts n t} : ts ## n = Some t → member t ts :=
  match ts with
    | nil ⇒ fun Heq ⇒ match None_Some Heq with end
    | t' :: ts' ⇒ if eq_nat_dec n (length ts') as b return (if b then _ else _) = _ → _
      then fun Heq ⇒ match Some_Some Heq with refl_equal ⇒ HFirst end
      else fun Heq ⇒ HNext (makeVar Heq)
  end.
```

Now dbify is straightforward to define. We use the functions proj1 and proj2 to decompose proofs of conjunctions.

```
Fixpoint dbify {ts} t (e : Phoas.exp (fun _ ⇒ nat) t) : wf ts e → DeBruijn.exp ts t :=
  match e in Phoas.exp _ t return wf ts e → DeBruijn.exp ts t with
    | Phoas.Var _ n ⇒ fun wf ⇒ DeBruijn.Var (makeVar wf)

    | Phoas.Const n ⇒ fun _ ⇒ DeBruijn.Const n
```

```
    | Phoas.Plus e1 e2 ⇒ fun wf ⇒
      DeBruijn.Plus (dbify e1 (proj1 wf)) (dbify e2 (proj2 wf))

    | Phoas.App _ _ e1 e2 ⇒ fun wf ⇒
      DeBruijn.App (dbify e1 (proj1 wf)) (dbify e2 (proj2 wf))
    | Phoas.Abs _ _ e1 ⇒ fun wf ⇒ DeBruijn.Abs (dbify (e1 (length ts)) wf)
  end.
```

We define the parametric translation Dbify by appealing to the well-formedness translation theorem Wf_wf that we proved earlier.

```
Definition Dbify t (E : Phoas.Exp t) (W : Wf E) : DeBruijn.exp nil t :=
  dbify (E _) (Wf_wf W).
```

To prove soundness, it is helpful to classify a set of contexts which depends on a de Bruijn index substitution.

```
Fixpoint makeG' ts (s : hlist typeDenote ts)
  : list { t : type & nat * typeDenote t }%type :=
  match s with
    | HNil ⇒ nil
    | HCons _ ts' v s' ⇒ existT _ _ (length ts', v) :: makeG' s'
  end.
```

We prove an analogous lemma to the one we proved connecting makeG and lookup. This time, we connect makeG' and hget.

```
Lemma In_makeG'_contra' : ∀ t v2 ts (s : hlist _ ts) n,
  In (existT _ t (n, v2)) (makeG' s)
  → n ≥ length ts
  → False.
  induction s; crush; eauto.
Qed.

Lemma In_makeG'_contra : ∀ t v2 ts (s : hlist _ ts),
  In (existT _ t (length ts, v2)) (makeG' s)
  → False.
  intros; eapply In_makeG'_contra'; eauto.
Qed.

Hint Resolve In_makeG'_contra.

Lemma In_makeG' : ∀ t v1 v2 ts s (w : ts ## v1 = Some t),
  In (existT _ t (v1, v2)) (makeG' s)
  → hget s (makeVar w) = v2.
  induction s; crush;
    match goal with
      | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E; crush
    end;
```

```
    repeat match goal with
             | [ ⊢ context[match ?pf with refl_equal ⇒ _ end] ] ⇒
               rewrite (UIP_refl _ _ pf)
           end; crush; elimtype False; eauto.
Qed.

Hint Resolve ln_makeG'.
```

Now the main inductive lemma can be stated and proved simply.

```
Lemma dbify_sound : ∀ G t (e1 : Phoas.exp _ t) (e2 : Phoas.exp _ t),
  exp_equiv G e1 e2
  → ∀ ts (w : wf ts e1) s,
    G = makeG' s
    → DeBruijn.expDenote (dbify e1 w) s = Phoas.expDenote e2.
  induction 1; crush; ext_eq; crush.
Qed.
```

In the usual way, we wrap dbify_sound into the final soundness theorem, formally establishing the expressive equivalence of PHOAS and de Bruijn index terms.

```
Theorem Dbify_sound : ∀ t (E : Exp t) (W : Wf E),
  DeBruijn.expDenote (Dbify W) HNil = Phoas.ExpDenote E.
  unfold Dbify, Phoas.ExpDenote; intros; eapply dbify_sound; eauto.
Qed.
```

Chapter 21

Higher-Order Operational Semantics

The last few chapters have shown how PHOAS can make it relatively painless to reason about program transformations. Each of our example languages so far has had a semantics that is easy to implement with an interpreter in Gallina. Since Gallina is designed to rule out non-termination, we cannot hope to give interpreter-based semantics to Turing-complete programming languages. Falling back on standard operational semantics leaves us with the old bureaucratic concerns about capture-avoiding substitution. Can we encode Turing-complete, higher-order languages in Coq without sacrificing the advantages of higher-order encoding?

Any approach that applies to basic untyped lambda calculus is likely to extend to most object languages of interest. We can attempt the "obvious" way of equipping a PHOAS definition for use in an operational semantics, without mentioning substitution explicitly. Specifically, we try to work with expressions with *var* instantiated with a type of values.

```
Section exp.
  Variable var : Type.

  Inductive exp : Type :=
  | Var : var → exp
  | App : exp → exp → exp
  | Abs : (var → exp) → exp.
End exp.


Inductive val : Type :=
| VAbs : (val → exp val) → val.
```

Error: *Non strictly positive occurrence of* "val" in
"(val → exp val) → val".

We would like to represent values (which are all function abstractions) as functions from variables to expressions, where we represent variables as the same value type that we are defining. That way, a value can be substituted in a function body simply by applying the

body to the value. Unfortunately, the positivity restriction rejects this definition, for much the same reason that we could not use the classical HOAS encoding.

We can try an alternate approach based on defining **val** like a usual class of syntax.

```
Section val.
  Variable var : Type.

  Inductive val : Type :=
  | VAbs : (var → exp var) → val.
End val.
```

Now the puzzle is how to write the type of an expression whose variables are represented as values. We would like to be able to write a recursive definition like this one:

```
Fixpoint expV := exp (val expV).
```

Of course, this kind of definition is not structurally recursive, so Coq will not allow it. Getting "substitution for free" seems to require some similar kind of self-reference.

In this chapter, we will consider an alternate take on the problem. We add a level of indirection, introducing more explicit syntax to break the cycle in type definitions. Specifically, we represent function values as numbers that index into a *closure heap* that our operational semantics maintains alongside the expression being evaluated.

21.1 Closure Heaps

The essence of the technique is to store function bodies in lists that are extended monotonically as function abstractions are evaluated. We can define a set of functions and theorems that implement the core functionality generically.

```
Section lookup.
  Variable A : Type.
```

We start with a lookup function that generalizes last chapter's function of the same name. It selects the element at a particular position in a list, where we number the elements starting from the end of the list, so that prepending new elements does not change the indices of old elements.

```
  Fixpoint lookup (ls : list A) (n : nat) : option A :=
    match ls with
      | nil ⇒ None
      | v :: ls' ⇒ if eq_nat_dec n (length ls') then Some v else lookup ls' n
    end.

  Infix "##" := lookup (left associativity, at level 1).
```

The second of our two definitions expresses when one list extends another. We will write *ls1* $\rightsquigarrow$ *ls2* to indicate that *ls1* could evolve into *ls2*; that is, *ls1* is a suffix of *ls2*.

```
Definition extends (ls1 ls2 : list A) := ∃ ls, ls2 = ls ++ ls1.
Infix "⇝" := extends (no associativity, at level 80).
```

We prove and add as hints a few basic theorems about lookup and extends.

```
Theorem lookup1 : ∀ x ls,
  (x :: ls) ## (length ls) = Some x.
  crush; match goal with
           | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
         end; crush.
Qed.

Theorem extends_refl : ∀ ls, ls ⇝ ls.
  ∃ nil; reflexivity.
Qed.

Theorem extends1 : ∀ v ls, ls ⇝ v :: ls.
  intros; ∃ (v :: nil); reflexivity.
Qed.

Theorem extends_trans : ∀ ls1 ls2 ls3,
  ls1 ⇝ ls2
  → ls2 ⇝ ls3
  → ls1 ⇝ ls3.
  intros ? ? ? [l1 ?] [l2 ?]; ∃ (l2 ++ l1); crush.
Qed.

Lemma lookup_contra : ∀ n v ls,
  ls ## n = Some v
  → n ≥ length ls
  → False.
  induction ls; crush;
    match goal with
      | [ _ : context[if ?E then _ else _] ⊢ _ ] ⇒ destruct E
    end; crush.
Qed.

Hint Resolve lookup_contra.

Theorem extends_lookup : ∀ ls1 ls2 n v,
  ls1 ⇝ ls2
  → ls1 ## n = Some v
  → ls2 ## n = Some v.
  intros ? ? ? ? [l ?]; crush; induction l; crush;
    match goal with
      | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
    end; crush; elimtype False; eauto.
Qed.
```

```
End lookup.

Infix "##" := lookup (left associativity, at level 1).
Infix "⇝" := extends (no associativity, at level 80).

Hint Resolve lookup1 extends_refl extends1 extends_trans extends_lookup.
```

We are dealing explicitly with the nitty-gritty of closure heaps. Why is this better than dealing with the nitty-gritty of variables? The inconvenience of modeling lambda calculus-style binders comes from the presence of nested scopes. Program evaluation will only involve one *global* closure heap. Also, the short development that we just finished can be reused for many different object languages. None of these definitions or theorems needs to be redone to handle specific object language features. By adding the theorems as hints, no per-object-language effort is required to apply the critical facts as needed.

21.2 Languages and Translation

For the rest of this chapter, we will consider the example of CPS translation for untyped lambda calculus with boolean constants. It is convenient to include these constants, because their presence makes it easy to state a final translation correctness theorem.

```
Module Source.
```

We define the syntax of source expressions in our usual way.

```
  Section exp.
    Variable var : Type.

    Inductive exp : Type :=
    | Var : var → exp
    | App : exp → exp → exp
    | Abs : (var → exp) → exp
    | Bool : bool → exp.
  End exp.

  Implicit Arguments Bool [var].

  Definition Exp := ∀ var, exp var.
```

We will implement a big-step operational semantics, where expressions are mapped to values. A value is either a function or a boolean. We represent a function as a number that will be interpreted as an index into the global closure heap.

```
  Inductive val : Set :=
  | VFun : nat → val
  | VBool : bool → val.
```

A closure, then, follows the usual representation of function abstraction bodies, where we represent variables as values.

```
  Definition closure := val → exp val.
```

```
Definition closures := list closure.
```

Our evaluation relation has four places. We map an initial closure heap and an expression into a final closure heap and a value. The interesting cases are for Abs, where we push the body onto the closure heap; and for App, where we perform a lookup in a closure heap, to find the proper function body to execute next.

```
Inductive eval : closures → exp val → closures → val → Prop :=
| EvVar : ∀ cs v,
  eval cs (Var v) cs v

| EvApp : ∀ cs1 e1 e2 cs2 v1 c cs3 v2 cs4 v3,
  eval cs1 e1 cs2 (VFun v1)
  → eval cs2 e2 cs3 v2
  → cs2 ## v1 = Some c
  → eval cs3 (c v2) cs4 v3
  → eval cs1 (App e1 e2) cs4 v3

| EvAbs : ∀ cs c,
  eval cs (Abs c) (c :: cs) (VFun (length cs))

| EvBool : ∀ cs b,
  eval cs (Bool b) cs (VBool b).
```

A simple wrapper produces an evaluation relation suitable for use on the main expression type Exp.

```
Definition Eval (cs1 : closures) (E : Exp) (cs2 : closures) (v : val) :=
  eval cs1 (E _) cs2 v.
```

To prove our translation's correctness, we will need the usual notions of expression equivalence and well-formedness.

```
Section exp_equiv.
  Variables var1 var2 : Type.

  Inductive exp_equiv : list (var1 * var2) → exp var1 → exp var2 → Prop :=
  | EqVar : ∀ G v1 v2,
    In (v1, v2) G
    → exp_equiv G (Var v1) (Var v2)

  | EqApp : ∀ G f1 x1 f2 x2,
    exp_equiv G f1 f2
    → exp_equiv G x1 x2
    → exp_equiv G (App f1 x1) (App f2 x2)
  | EqAbs : ∀ G f1 f2,
    (∀ v1 v2, exp_equiv ((v1, v2) :: G) (f1 v1) (f2 v2))
    → exp_equiv G (Abs f1) (Abs f2)
```

```
    | EqBool : ∀ G b,
      exp_equiv G (Bool b) (Bool b).
  End exp_equiv.

  Definition Wf (E : Exp) := ∀ var1 var2, exp_equiv nil (E var1) (E var2).
End SOURCE.
```

Our target language can be defined without introducing any additional tricks.

```
Module CPS.
  Section exp.
    Variable var : Type.

    Inductive prog : Type :=
    | Halt : var → prog
    | App : var → var → prog
    | Bind : primop → (var → prog) → prog

    with primop : Type :=
    | Abs : (var → prog) → primop

    | Bool : bool → primop

    | Pair : var → var → primop
    | Fst : var → primop
    | Snd : var → primop.
  End exp.

  Implicit Arguments Bool [var].

  Notation "x ← p ; e" := (Bind p (fun x ⇒ e))
    (right associativity, at level 76, p at next level).

  Definition Prog := ∀ var, prog var.
  Definition Primop := ∀ var, primop var.

  Inductive val : Type :=
  | VFun : nat → val
  | VBool : bool → val
  | VPair : val → val → val.
  Definition closure := val → prog val.
  Definition closures := list closure.

  Inductive eval : closures → prog val → val → Prop :=
  | EvHalt : ∀ cs v,
    eval cs (Halt v) v

  | EvApp : ∀ cs n v2 c v3,
```

```
    cs ## n = Some c
    → eval cs (c v2) v3
    → eval cs (App (VFun n) v2) v3

  | EvBind : ∀ cs1 p e cs2 v1 v2,
    evalP cs1 p cs2 v1
    → eval cs2 (e v1) v2
    → eval cs1 (Bind p e) v2

  with evalP : closures → primop val → closures → val → Prop :=
  | EvAbs : ∀ cs c,
    evalP cs (Abs c) (c :: cs) (VFun (length cs))

  | EvPair : ∀ cs v1 v2,
    evalP cs (Pair v1 v2) cs (VPair v1 v2)
  | EvFst : ∀ cs v1 v2,
    evalP cs (Fst (VPair v1 v2)) cs v1
  | EvSnd : ∀ cs v1 v2,
    evalP cs (Snd (VPair v1 v2)) cs v2

  | EvBool : ∀ cs b,
    evalP cs (Bool b) cs (VBool b).
  Definition Eval (cs : closures) (P : Prog) (v : val) := eval cs (P _) v.
End CPS.
```

```
Import Source CPS.
```

Finally, we define a CPS translation in the same way as in our previous example for simply-typed lambda calculus.

```
Reserved Notation "x ⟵ e1 ; e2" (right associativity, at level 76, e1 at next level).

Section cpsExp.
  Variable var : Type.

  Import Source.

  Fixpoint cpsExp (e : exp var)
    : (var → prog var) → prog var :=
    match e with
      | Var v ⇒ fun k ⇒ k v

      | App e1 e2 ⇒ fun k ⇒
        f ⟵ e1;
        x ⟵ e2;
        kf ← CPS.Abs k;
        p ← Pair x kf;
```

```
      CPS.App f p
    | Abs e' ⇒ fun k ⇒
      f ← CPS.Abs (var := var) (fun p ⇒
        x ← Fst p;
        kf ← Snd p;
        r ⟵ e' x;
        CPS.App kf r);
      k f

    | Bool b ⇒ fun k ⇒
      x ← CPS.Bool b;
      k x
  end

  where "x ⟵ e1 ; e2" := (cpsExp e1 (fun x ⇒ e2)).
End cpsExp.

Notation "x ⟵ e1 ; e2" := (cpsExp e1 (fun x ⇒ e2)).

Definition CpsExp (E : Exp) : Prog := fun _ ⇒ cpsExp (E _) (Halt (var := _)).
```

21.3 Correctness Proof

Our proof for simply-typed lambda calculus relied on a logical relation to state the key induction hypothesis. Since logical relations proceed by recursion on type structure, we cannot apply them directly in an untyped setting. Instead, we will use an inductive judgment to relate source-level and CPS-level values. First, it is helpful to define an abbreviation for the compiled version of a function body.

```
Definition cpsFunc var (e' : var → Source.exp var) :=
  fun p : var ⇒
    x ← Fst p;
    kf ← Snd p;
    r ⟵ e' x;
    CPS.App kf r.
```

Now we can define our correctness relation **cr**, which is parameterized by source-level and CPS-level closure heaps.

```
Section cr.
  Variable s1 : Source.closures.
  Variable s2 : CPS.closures.

  Import Source.
```

Only equal booleans are related. For two function addresses *l1* and *l2* to be related, they must point to valid functions in their respective closure heaps. The address *l1* must point

to a function *f1*, and *l2* must point to the result of compiling function *f2*. Further, *f1* and *f2* must be equivalent syntactically in some variable environment G, and every variable pair in G must itself belong to the relation we are defining.

```
Inductive cr : Source.val → CPS.val → Prop :=
| CrBool : ∀ b,
  cr (Source.VBool b) (CPS.VBool b)

| CrFun : ∀ l1 l2 G f1 f2,
  (∀ x1 x2, exp_equiv ((x1, x2) :: G) (f1 x1) (f2 x2))
  → (∀ x1 x2, In (x1, x2) G → cr x1 x2)
  → s1 ## l1 = Some f1
  → s2 ## l2 = Some (cpsFunc f2)
  → cr (Source.VFun l1) (CPS.VFun l2).
End cr.

Notation "s1 & s2 |- v1 ~~ v2" := (cr s1 s2 v1 v2) (no associativity, at level 70).

Hint Constructors cr.
```

To prove our main lemma, it will be useful to know that source-level evaluation never removes old closures from a closure heap.

```
Lemma eval_monotone : ∀ cs1 e cs2 v,
  Source.eval cs1 e cs2 v
  → cs1 ⇝ cs2.
  induction 1; crush; eauto.
Qed.
```

Further, cr continues to hold when its closure heap arguments are evolved in legal ways.

```
Lemma cr_monotone : ∀ cs1 cs2 cs1' cs2',
  cs1 ⇝ cs1'
  → cs2 ⇝ cs2'
  → ∀ v1 v2, cs1 & cs2 |- v1 ~~ v2
    → cs1' & cs2' |- v1 ~~ v2.
  induction 3; crush; eauto.
Qed.

Hint Resolve eval_monotone cr_monotone.
```

We state a trivial fact about the validity of variable environments, so that we may add this fact as a hint that `eauto` will apply.

```
Lemma push : ∀ G s1 s2 v1' v2',
  (∀ v1 v2, In (v1, v2) G → s1 & s2 |- v1 ~~ v2)
  → s1 & s2 |- v1' ~~ v2'
  → (∀ v1 v2, (v1', v2') = (v1, v2) ∨ In (v1, v2) G → s1 & s2 |- v1 ~~ v2).
  crush.
Qed.
```

```
Hint Resolve push.
```

Our final preparation for the main lemma involves adding effective hints about the CPS language's operational semantics. The following tactic performs one step of evaluation. It uses the Ltac code **eval** *hnf* `in e` to compute the *head normal form* of `e`, where the head normal form of an expression in an inductive type is an application of one of that inductive type's constructors. The final line below uses *solve* to ensure that we only take a `Bind` step if a full evaluation derivation for the associated primop may be found before proceeding.

```
Ltac evalOne :=
  match goal with
    | [ ⊢ CPS.eval ?cs ?e ?v ] ⇒
      let e := eval hnf in e in
        change (CPS.eval cs e v);
          econstructor; [ solve [ eauto ] | ]
  end.
```

For primops, we rely on `eauto`'s usual approach. For goals that evaluate programs, we instead ask to treat one or more applications of *evalOne* as a single step, which helps us avoid passing `eauto` an excessively large bound on proof tree depth.

```
Hint Constructors evalP.
Hint Extern 1 (CPS.eval _ _ _) ⇒ evalOne; repeat evalOne.
```

The final lemma proceeds by induction on an evaluation derivation for an expression *e1* that is equivalent to some *e2* in some environment G. An initial closure heap for each language is quantified over, such that all variable pairs in G are compatible. The lemma's conclusion applies to an arbitrary continuation k, asserting that a final CPS-level closure heap *s2* and a CPS-level program result value *r2* exist.

Three conditions establish that *s2* and *r2* are chosen properly: Evaluation of *e2*'s compilation with continuation k must be equivalent to evaluation of k *r2*. The original program result *r1* must be compatible with *r2* in the final closure heaps. Finally, *s2'* must be a proper evolution of the original CPS-level heap *s2*.

```
Lemma cpsExp_correct : ∀ s1 e1 s1' r1,
  Source.eval s1 e1 s1' r1
  → ∀ G (e2 : exp CPS.val),
    exp_equiv G e1 e2
    → ∀ s2,
      (∀ v1 v2, In (v1, v2) G → s1 & s2 |- v1 ~~ v2)
      → ∀ k, ∃ s2', ∃ r2,
        (∀ r, CPS.eval s2' (k r2) r
          → CPS.eval s2 (cpsExp e2 k) r)
        ∧ s1' & s2' |- r1 ~~ r2
        ∧ s2 ⇝ s2'.
```

The proof script follows our standard approach. Its main loop applies three hints. First, we perform inversion on any derivation of equivalence between a source-level function value

and some other value. Second, we eliminate redundant equality hypotheses. Finally, we look for opportunities to instantiate inductive hypotheses.

We identify an IH by its syntactic form, noting the expression E that it applies to. It is important to instantiate IHes in the right order, since existentially-quantified variables in the conclusion of one IH may need to be used in instantiating the universal quantifiers of a different IH. Thus, we perform a quick check to `fail 1` if the IH we found applies to an expression that was evaluated after another expression E' whose IH we did not yet instantiate. The flow of closure heaps through source-level evaluation is used to implement the check.

If the hypothesis H is indeed the right IH to handle next, we use the *guess* tactic to guess values for its universal quantifiers and prove its hypotheses with `eauto`. This tactic is very similar to *inster* from Chapter 12. It takes two arguments: the first is a value to use for any properly-typed universal quantifier, and the second is the hypothesis to instantiate. The final inner `match` deduces if we are at the point of executing the body of a called function. If so, we help *guess* by saying that the initial closure heap will be the current closure heap *cs* extended with the current continuation k. In all other cases, *guess* is smart enough to operate alone.

```
  induction 1; inversion 1; crush;
    repeat (match goal with
              | [ H : _ & _ |- Source.VFun _ ~~ _ ⊢ _ ] ⇒ inversion H; clear H
              | [ H1 : ?E = _, H2 : ?E = _ ⊢ _ ] ⇒ rewrite H1 in H2; clear H1
              | [ H : ∀ G e2, exp_equiv G ?E e2 → _ ⊢ _ ] ⇒
                match goal with
                  | [ _ : Source.eval ?CS E _ _, _ : Source.eval _ ?E' ?CS _,
                      _ : ∀ G e2, exp_equiv G ?E' e2 → _ ⊢ _ ] ⇒ fail 1
                  | _ ⇒ match goal with
                          | [ k : val → prog val, _ : _ & ?cs |- _ ~~ _,
                              _ : context[VFun] ⊢ _ ] ⇒
                            guess (k :: cs) H
                          | _ ⇒ guess tt H
                        end
                end
            end; crush); eauto 13.
Qed.
```

The final theorem follows easily from this lemma.

```
Theorem CpsExp_correct : ∀ E cs b,
  Source.Eval nil E cs (Source.VBool b)
  → Wf E
  → CPS.Eval nil (CpsExp E) (CPS.VBool b).
  Hint Constructors CPS.eval.

  unfold Source.Eval, CPS.Eval, CpsExp; intros ? ? ? H1 H2;
```

```
    generalize (cpsExp_correct H1 (H2 _ _) (s2 := nil)
      (fun _ _ pf ⇒ match pf with end) (Halt (var := _))); crush;
    match goal with
      | [ H : _ & _ |- _ ~~ _ ⊢ _ ] ⇒ inversion H
    end; crush.
Qed.
```