Certified Programming with Dependent Types

Adam Chlipala

February 28, 2017

A print version of this book is available from the MIT Press. For more information, see the book's home page:

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

Contents

Chapter 1

Introduction

1.1 Whence This Book?

書いたプログラムが正しいことを検査してくれるプログラムが欲しいものです. 今日ソフトウェアを書く人々のほとんどが, 実践者であるか学者かにかかわらず, 形式的プログラム検証のコストが利益を上まわってしまうときめつけています. 計算機科学の歴史においてかつて語られ, そして失敗した, いくつかの強気な展望が, このきめつけの少なからぬ原因となっています. 本書の目的は, プログラム検証の技術が十分成熟したものであり, 計算機科学の各種研究プロジェクトの補助として用いる意味があるということを, あなたに納得してもらうことです. また, 納得を与えるにとどまらず, 認証されたプログラムを実用的にエンジニアリングするための手引き書を提供することも私の望みです. 本書のあつかう主題のほとんどは, 伝統的な数学の定理の扱いなどを含む, 計算機による対話的な定理証明一般と関係のあるものです. 実際に私は, 検証されたプログラムが, あらゆる種類の形式化における構成要素として役にたつのだということを, 実証してみせたいと考えています.

機械化された定理証明の研究は 20 世紀後半に始まりました. 最初期の実用的な成果の一つに, 「Boyer-Moore 定理証明器」Nqthm [3] があります. これはハードウェアとソフトウェア両方の層にわたる正しさを定理として証明する目的などに用いられました [24]. Nqthm の後継である ACL2 [16] は産業界で顕著に受けいれられ, 例えば AMD によって浮動小数除算の正しさを保証するために用いられました [25].

21 世紀初頭に対話的定理証明の実用的用法は著しく加速しつつ発展しました. 幾つかのよく知られた形式的開発が, 本書の扱うシステムである Coq を用いて行われました. 純粋数学の領域においては, Georges Gonthier が四色定理の機械的に検査された証明を構築しました [12]. 四色定理は百年以上前に提示された数学の問題で, これまでに得られていた証明は, 鍵となる事実を総当たりで確かめるための場当たり的なソフトウェアを信用しなければならないものでした. プログラム検証の領域においては, Xavier Leroy が CompCert プロジェクトを主導し, 本物の組み込みソフトウェアで使うのに十分な堅牢さを持つ, 検証された C コンパイラバックエンドを作り上げました [18].

他にも近頃多くのプロジェクトが, 計算機による証明支援系ソフトウェアを用いて重要な定理の証明を行い, 注目を集めています. 例えばGerwin Kleinが主導する L4.verified プロジェクト [17] は, 現実的なマイクロカーネルの正しさを, 証明支援系 Isabelle/HOL [26] を用いて

機械的に証明しました. この分野における進行中の事業は数多くあり, 最近の成功例を全てを挙げることはほぼ不可能です. 従ってこれ以降, 機械的に検査された証明を追求すべきであることと, それを作り出すことがどうやら可能であるということについて, 読者は納得したという仮定をすることにしたいと思います. (まだ納得していない読者には, “machine-checked proof” で Web を検索することをお勧めします.)

本書のタイトルでは**認証されたプログラム**という考えがはっきりと打ち出されています.「認証された」という語が意味するものは政府の規則**ではありません**. つまり作り上げられたシステムの信頼性が十分厳しい規格に従うことを, どのように実証するか定めた規則のことを言うのではありません. そうではなく, この認証という概念は, プログラミング言語や形式手法のコミュニティでは標準的に, **認証書**の考えと関連づけられます. 認証書とは, 言いかえると形式的で数学的な産物であり, プログラムが仕様に合っていることを証明するものです. 政府の認証手続きが強い数学的な保証を与えることはほとんどありませんが, 認証されたプログラムを書くことは, われわれの望むような保証をほとんどいくらでも強く与えてくれます. 基礎的な数理論理学を信頼し, その論理の実装のどれかを信頼し, そしてわれわれが非形式的な意図を正しく形式的仕様にエンコードできたことを信頼するならば, 正しくないソフトウェアを認証してしまう可能性は他にほとんど残りません. コンパイラや他のバッチ処理として走るプログラムについては, **認証を伴う**プログラムという概念もよく知られており, これは実行すると, 回答と共に回答が正しいことの証明を出力します. 認証を伴うプログラムを証明検査器と組み合わせることで認証されたプログラムが得られますが, 本書では認証されたプログラムの方に焦点をあてることにします. 同時に, Coq で定理を述べて証明するための, 一般的興味を惹くような原理や技術も紹介します.

今日では, 機械的に確かめられた数学の証明や機械的に認証されたプログラムを構築するために, 広く利用されているツールがいくつも（決して「沢山」というわけではありませんが）存在します. いくつかの条件を満たす対話的な「証明支援系」を, 以下に列挙し尽くしてみようと思います. 条件の一つめとして, 作者がツールの用途として, ソフトウェアに関連した応用を意図していなければなりません. 二つめとして, ツール自体の研究者以外の利用者が有意義に利用できるよう, 十分な工学的努力がなされていなければなりません. 三つめの条件は二つめをより経験的に保証するものです. すなわち, ツールの開発チーム以外のユーザコミュニティがちゃんと存在していなければなりません.

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 は「証明支援系開発のフレームワーク」である Isabelle [29] を用いて実装されており, 論理体系 HOL のための証明支援系で最もよく利用されるものです. ここで議論した目的のためには, HOL の他の実装も同様に利用できます.

1.2 Why Coq?

この本は Coq を使った認証を伴うプログラミングについての本になる予定です．私はそれ（Coq）がこの仕事（certified programming）に最良のツールだと確信しています．Coq にはとても魅力的な性質が多く備わっていますので，ここ（以下）でまとめます．同時に他の候補となるツールがどの性質を欠いているかも説明します．

1.2.1 高階の関数型プログラミング言語に基づいている

認証を伴うプログラムを書くとき，関数型プログラミング言語のよく知られた便利さをあきらめる理由はありません．ここで挙げるツールは関数型プログラミング言語を基礎としているため，証明に関係する機能を使わなくても普通のプログラムを書いたり実行したりできます．*1* **階の**言語のみを基礎とするという点点で ACL2 は重要です．つまり，(1 階の言語のツールでは) 関数上の関数などの関数型プログラミングの便利な機能が使えないのです．この便利さの代わりに，自動証明がいかにうまく動くかについてより多様な仮定が ACL2 ではできるのですが，他の証明支援系でも 1 階のプログラムを書くときは同様の利点を回復できます．

1.2.2 依存型

依存型を持つ言語は型の内部への言及を含むことができます．例えば，配列をあらわす型にはその配列のサイズを与えるプログラム式を含むことができるので，配列の範囲外アクセスがないことを静的に確かめることができるようになります．型の正しさを表す性質を効果的に捉えることで，依存型はさらに先まで行けます．例えば，この本で後ほど，正しく型付けされたソースプログラムから正しく型付けされたターゲットプログラムに変換することを保証する型をコンパイラに与える方法を見ます．

公然のことですが，ACL2 と HOL では依存型は使えません．PVS と Twelf は Coq の依存型言語のそれぞれ別の真部分集合をサポートします．Twelf の型言語は bare-bones に制限されています．つまり単形のラムダ計算です．そのため型の内部での計算に重大な制約が置かれます．この制約は Twelf が証明を表現したり証明したりするアプローチの健全性を議論するときに重要です．

それに対して PVS の依存型はより一般的ですが，subset type という単一の仕組みに制限されています．つまり，通常の型は述語を付加することで定義されます．subset type の要素の 1 つ 1 つは base type の要素のうちその述語を満たすものです．この本の 6 章ではこの様式のプログラミングを Coq でする方法を紹介します．一方 Part II の他の章では Coq による依存型を扱い，PVS がサポートする範囲外です．

依存型が有用なのは，型の正しさを表現するのを助けるからだけではありません．依存型のおかげで，しばしば証明らしいものを書かずに certified program を書くことができるようになるんです．subset type だけだと，離れ業を十分駆使すれば妥当な性質を表現できるのですが，人間が操作するような証明支援機は通常，証明を明示的に構築する必要があります．形式的な証明を書くのは大変なので，なるべく避けたいものです．この目的のために，依存型には計り知れない価値があります．

1.2.3 確認しやすいカーネル証明言語

自動化された決定手続きが判定してくれるというのは実践的な定理証明では有用です．しかし，一つずつの手続きが正しく実装されているということを信頼しないと行けないのは残念です．複雑で拡張可能な手順を使って証明を探し出すのが先かどうかに関わらず，証明支援機が核となる小さな言語で表現された_証明項_を生成するとき，その証明支援機は de Bruijn criterion を満たすと言います．数学の形式的な基礎の提案の中で見られるのと同様の複雑さをこのようなコア言語も持っています．証明を_探す_際のバグの可能性は無視してもよく，証明を検証するのに必要な小さな部分だけだけによって，証明を信じることができます．

Coq は de Bruijn criterion を満たします．一方 ACL2 は満たしません．というのは，ACL2 は手の込んだ決定手続きを採用し，ACL2 の結果を正当化する証跡を生成しないからです．PVS は_戦略_をサポートします．戦略では手の込んだ証明手順を「原始的な」証明の段階で実装します．ここで「原始的な」といっても Coq ほど原始的ではないです．例えば，命題論理の恒真式ソルバは PVS では原始的とされるため，そういうものが de Bruijn criterion を満たすかどうかという好みの問題になります．HOL の実装が de Bruijn criterion に適合するのはより明らかです．Twelf に対してはより不明瞭です．

1.2.4 便利でプログラム可能な証明自動化

残りのツールについては，全て，新しい決定手続きのユーザー拡張をサポートします．それは，直接ツールの実装（Coq の場合は OCaml です）をハックする形になります．ACL2 と PVS は de Bruijn 条件を満たさないため，全体の正しさは新しい手順を作った人に左右されてしまいます．

ISabelle/HOL と Coq はどちらもあらあな証明操作を ML でコードすることをサポートします．それによって不正な証明が受け入れられることはありません．加えて，Coq はドメイン特化言語をもっていて決定手続きを通常の Coq ソースコードの中でコーディングすることができます．ML に抜け出す必要はありません．この言語（DSL）は Ltac と呼ばれていて，私が思うに，Ltac は証明支援系の世界における無名の英雄です．Ltac は深刻な間違い犯すことを防ぐだけではありません．多くの画期的なプログラム構成要素を持っていて，それを組み合わせることで「決定手続きによる証明」というスタイルをとても快適にしてくれます．私たちはこれらの機能を後の章で見て行きます．

1.2.5 リフレクションによる証明

計算についての概念を豊富に統合した証明言語を選ぶことで驚くほど多くの利点が得られます．Coq はプログラムと証明項を同じ構文クラスで表現します（？）これによって証明を計算するプログラムを作るのが簡単になります．十分豊富な依存型があることで，そのようなプログラムは**認証を伴う決定手続き**になります．そのような場合，それらの認証を伴う手続きは走らせないとしても有用なのです．もしそのプログラムをわざわざ実行した場合，適切な十分な根拠のある証明が得られる，ということをそれらの型が保証してくれます．

このテクニックは多くは**リフレクションによる証明**と呼ばれます．証明検査の際，論理的な命題の中に非自明な計算を取り入れるというものです．さらに，このようなもののほとんどの例では適切な定理を表現するために依存型を必要とします．先ほどより挙げている証

明支援機の中では，Coq だけが型レベルでの計算をするというスタイルのリフレクションを提供しています．一方 PVS ではとても似た機能を refinement type を使ってサポートしています．

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, to protect the sanity of the programmer. 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 space 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 those features.

[1] `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`
[2] `https://code.google.com/p/epigram/`

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 fan of some programming language and 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 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 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 understand Scheme deeply will probably be fine.

My background is in programming languages, formal semantics, and program verification.

I sometimes use examples from that domain. As a reference on these topics, I recommend *Types and Programming Languages* [33], by Benjamin C. Pierce; however, I have tried to choose examples so that they may be understood without background in semantics.

1.6 Using This Book

本書は coqdoc というプログラムを使って Coq のソースファイルから自動的に生成されています。識別子から対応する定義にハイパーリンクの貼られた PDF バージョンは以下で利用できます：

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

オンラインの HTML バージョンも利用できます。もちろんこちらでもハイパーリンクが付いています：

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

本書のソースファイルも無料で利用できます：

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

ソースファイルでは本書に載っているすべてのコードがあり、コメントの中に本書と同じ順番で説明がつけられています。後述する Coq のグラフィカルインターフェースを使えばコードを 1 ステップずつ対話的に読み進められます。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.

Coq を生産的に使うには優れたグラフィカルインターフェースが必要不可欠でしょう。筆者は Emacs の Proof General[3]モードを使っています。Proof General は Coq の他にもいくつかの証明支援系をサポートしています。Coq チームによって開発されているスタンドアローンの CoqIDE プログラムもあります。著者は同じエディタの中で認証付きプログラミングと証明を他の作業を両方行えることを好みます。本書の最初で Coq の使い方を紹介する際には Proof General の手順を明示的に参照しますが、本書のほとんどはインターフェースに依存しないので、もし CoqIDE を使いたければそちらを使っても構いません。CoqIDE のバージョン 8.4 以下で本書のソースを実行する際に生じる一つの問題は、始めた証明をキャンセルするときに用いる Coq の `Abort` や `Restart` コマンドがサポートされていないことです。これらのコマンドを現実の、完了した開発に残すのは良くないでしょうが、著者はこれらのコマンドを証明を設計するときのユーザの思考プロセスをたどるソースコードを書くのに役立つと考えます。

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

1.6.1 Reading This Book

関数型プログラミングか形式手法の熟練者には、ある意味で Coq の使い方を学ぶことは難しくありません。Coq のマニュアル [7] や Bertot and Castéran [1]、Pierce らの *Software Foundations*[4]は生産的に Coq を使うのに役立ちます。しかし、著者は重大な Coq 開発をなし遂げる最良の方法は慣れることでは決してないと考えます。本書では、著者のテクニックを提示し、それらを最後の一、二章の発展的な道具として扱うのではなく、始めからそれらを使うつもりです。最初の章で依存型で何ができるかを見せた後、本書の第一部に対してよりシンプルなプログラミングスタイルへ方向転換します。また、本書の他の主眼として、Ltac による証明の自動化をほとんど初歩から技術的に説明します。

読者は著者が Coq の異なる熟練度の人々に合わせて読む順番を与えることを提案しているかもしれません。本書の第一部ではほとんどの Coq ユーザは既によく知っている基礎概念に多くの説明を捧げているのは事実です。しかし、それらの概念を導入するにあたって著者の好む自動化された証明のスタイルを展開するので、基礎の章も経験のある Coq ハッカーにとって価値があると思います。

Coq の経験がない読者は前述の議論を無視して構いません！　読者がなぜ他がマニュアル的に証明のステップの列を入力するのに多くの時間を費すのか疑問を持つくらいに、著者の早くからの証明の自動化への強い信頼が最も自然な方法と思えることを望みます。

Coq はとえも複雑なシステムで、美しい原理を追求することより実用的な懸念から導出されたコマンドがたくさんあります。何か構造物を始めて使うときは、それが何を成し遂げるかの一文での直感的説明を与えますが、詳細は Coq のリファレンスマニュアル [7] に譲ります。完璧な理解を求める読者はリファレンスマニュアルを頻繁に参照することでしょう。この意味では、本書は完全にはスタンドアローンになるようには書かれていません。しばしばコードの中で構造物を先に説明することなしに使うことがありますが、説明はいつもそのコードの直後の段落に置かれるでしょう。

本書の前のバージョンでは章の終わりに演習問題が含まれていました。それから、演習問題をなくして解説に焦点を当てることに決めました。本書のさまざまな読者に向けた演習問題のデータベースは Web で利用できます[5]。しかし、Coq を学ぶための最良の方法は、人工的な演習問題を解くことよりも Coq を実際のプロジェクトに応用し始めることだと著者は提案します。

1.6.2 On the Tactic Library

Coq にあらかじめあるタクティクは十分高レベルな自動化をサポートしていないので、高機能な自動証明から始めるために、それらに取り組むよりも、本書のソースに証明を探すプログラムである**タクティク**のライブラリを含めました。これらのタクティクは最初の章からすでにコード例と共に用います。

このタクティクライブラリを開発で使うことについて何人かの読者に尋ねられたことがあります。著者としては、このタクティクライブラリは本書の特定の例と共に設計したつもりなので、他の場面で使うことは推奨しません。第三部でこれらのタクティクを再実装し、またそれを越えるのに十分な技術について添えます。一般に対話的定理証明では決定不可能な

[4] `http://www.cis.upenn.edu/~bcpierce/sf/`

[5] `http://adam.chlipala.net/cpdt/ex/`

問題も扱われるので、すべてのゴールを解くようなタクティクはないかもしれません。(すぐに出てくる *crush* タクティクは時々そのようなものだと感じるかもしれませんが！) *crush* やその兄弟のタクティクの実装にはとても便利な秘訣があるので、コメント付きのソースファイル `CpdtTactics.v` を調べてみるのも有益かもしれません。著者は新しいプロジェクトごとに新しいタクティクライブラリを実装しています。各プロジェクトは異なる決定不可能な理論の組み合わせを含んでいて、異なる種類のヒューリスティクスがうまくいくからです。そして皆さんにもそれを勧めます。

1.6.3 Installation and Emacs Set-Up

次の章の最初では、皆さんが Coq と Proof General をインストールした状態であることを仮定します。本書のコードは Coq バージョン 8.4pl5 と 8.5beta2 でテストされています。部分的には他のバージョンで動くかもしれませんが、本書のソースはより以前のバージョンではビルドに失敗すると予想されます。

次の章でソースを処理するために Proof General の設定をするには、以下のシンプルなステップが必要です。

1. 以下からソースを取得

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

2. tarball をディレクトリ `DIR` に展開

3. `DIR` 内で `make` を実行 (マルチコアがあるなら、それを使うために `-j` フラグを付けることが望まれます)

4. Coq の対話的トップレベルを与える `coqtop` プログラムのコマンドライン引数を Proof General に渡す際にはいくつか小さい困った問題があります。たくさんのソースファイルに共有されるであろう設定を追加する方法の一つは、custom variable setting を `.emacs` ファイルに以下のように追加することです：

```
(custom-set-variables
  ...
  '(coq-prog-args '("-R" "DIR/src" "Cpdt"))
  ...
)
```

   ここで見せた特別な引数は本書のコードのための選択です。省略されているのは皆さんが既に設定しているかもしれない Emacs カスタマイズです。`.emacs` ファイルに、一つを除いて `custom-set-variables` ブロックにコメントアウトした複数の代替のフラグを保存しておくと便利かもしれません。

   あるいは、設定を適用したいソースファイルのディレクトリ内の `.dir-locals.el` ファイルを使うことで、Proof General の設定はディレクトリごとに指定できます。以下が本書のソースに使えるような設定ファイルの例です。Coq を Emacs サポートモードで開始するための引数を含める必要があることに注意してください。

```
((coq-mode . ((coq-prog-args . ("-emacs-U" "-R" "DIR/src" "Cpdt")))))
```

本書の各章はコメント付きの Coq ソースファイルから生成されています。Proof General でそれらをロードしてステップ毎に実行できます。Coq バイナリ `coqtop` をコマンドライン引数`-R DIR/src Cpdt` を必ずつけて実行してください。Proof General を正しくインストールしたならば、Coq モードは Emacs 内で`.v` バッファに入ったときに自動で始まり、`.emacs` 設定の上記のアドバイスは `coqtop` に適切な引数が Emacs により渡されることを保証するでしょう。

Proof General では、Coq が実行されたバッファの一部は青の背景などでハイライトされます。Coq のソースファイルをステップ毎に実行するには、実行したい場所にカーソルを置いて C-C C-RET を押します。これは、ハイライトされた領域の外側でも内側でも、通常のステップ毎のコーディングのために使えます。

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`
General Recursion	`GeneralRec.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 by Logic Programming	`LogicProg.v`
Proof Search in Ltac	`Match.v`
Proof by Reflection	`Reflection.v`
Proving in the Large	`Large.v`
A Taste of Reasoning About Programming Language Syntax	`ProgLang.v`

Chapter 2

Some Quick Examples

まずは実際に動く例として、ソース言語からスタックマシンへの証明付きコンパイラの構成から始めましょう。最初はシンプルなソース言語から始め、少しずつ複雑なソース言語も扱っていきます。証明に関しては、いくつかの便利なタクティクを紹介し、それらがどのように手動の証明で使われるか、またそれらがどれだけ簡単に自動化できるかを見ていきます。この章では使う機能の完全な説明を与えるつもりはありません。それよりはむしろ、Coq でできることは何なのかを述べるつもりです。後の章ですべての概念をボトムアップに紹介していきます。言い換えれば、ほとんどの読者にとってここで行われることを完璧に理解するのは難しいかもしれませんが、ここでのデモが残りの章への興味に繋がっていただければ十分です！

読者はいつでもこの章のソースファイル `StackMachine.v` を Proof General を使って対話的に 1 ステップずつ見ていくことができます。あるいは、Coq 開発の過程を手で書いて感じたければ、この章のソースコードの一つ一つを Emacs バッファ内で新規の `.v` ファイルに書き込んでいっても良いでしょう。後者の方法を取るなら、ファイルの先頭に以下の三行をコピーしてください。

```
Require Import Bool Arith List Cpdt.CpdtTactics.
Set Implicit Arguments.
Set Asymmetric Patterns.
```

今後、各章のソースコード内の似たコマンドは文章中では省略するので、省略された部分は以前与えたところからコピー&ペーストする必要があります。具体的には、どの章の始めにも上の三行が挿入されます。ただし、章ごとに `Require Import` の後を必要に合わせて書き換えなければいけません。二行目のコマンドは型推論に関して定義の標準的なふるまいに影響し、三行目はより簡潔なパターンマッチングの機能を与えます (三行目は Coq のバージョン 8.5 以降のコマンドで、バージョン 8.5 未満には機能しません)。

2.1 自然数の算術式

コンパイラの教科書にはおなじみの、数値型の上での算術式から始めましょう。

2.1.1　ソース言語

ソース言語のシンタックスから始めます。

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

私たちの初めての Coq コードとなるこの一行は、ML や Haskell のプログラマには意外なものではないでしょう。ソース言語の二項演算子を表すため、代数的データ型 (algebraic datatype) **binop** を定義しました。ここで、ML や Haskell と比較されるべき二つのポイントがあります。一つは、Coq は `data`、`datatype`、`type` の代わりに `Inductive` を使うことです。これは単なる表面上のシンタックスの違いではありません。この章ではごく簡潔にしか触れませんが、Coq の帰納的データ型 (inductive data types) はありふれた代数的データ型よりもずっと豊かな表現力を持っていて、とくに数学のすべてを表現することができます。二つ目は、`: Set` です。これは、プログラムの構成要素として考えられるべきデータ型を定義していることを宣言します。プログラムの構成要素ではなく、証明の世界のデータ型、さらにプログラムと証明の両方を包含する、無限の階層を持つ世界のデータ型を定義するときのキーワードも後に与えます。後者は、高階の構成をするときに役立ちます。

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

算術式を定義しました。定数 Const は一つの自然数値の引数から成り、二項演算子 Binop は一つの演算子と二つのオペランド式から成るものとして与えます。

本書を PDF 版で読んでいる読者への注意：coqdoc は Coq のソースコードを LaTeX や HTML 形式に変換します。この PDF 上の右矢印→はソース上では ASCII テキストの `->` です。この章では他に、二重の右矢印⇒を `=>`、記号∀を `forall`、デカルト積×を `*` にする置き換えがあります。ASCII テキストでどう書くのかが分からなくなったら、ソースコードを参照してください。

言語が定義されたので、次にこの言語のプログラムの意味を与えることができます。ここでは、プログラムの意味は、インタプリタを書いて与えることにします。これはごく単純な操作的/表示的意味論として考えることができます。(もしあなたがこれらの意味論的手法に不慣れでも、心配いりません。「あたりまえの」構成をしていきますから。)

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

二項演算子の意味は自然数の二引数関数です。ML や Haskell における `match` や `case` のようなパターンマッチングを使って定義し、Coq の標準ライブラリ内の関数 plus と mult を参照しています。`Definition` キーワードは、Coq の項を名前に束縛するための Coq で頻繁に使われる記法で、場合に応じて構文糖衣を持ちます。上の例でも関数を定義するための構文糖衣が用いられており、展開すると以下のようになります：

```
Definition binopDenote : binop → nat → nat → nat := fun (b : binop) ⇒
  match b with
```

```
    | Plus ⇒ plus
    | Times ⇒ mult
  end.
```

この例では、次のようにすべての型注釈を外すことができます：

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

ML や Haskell のような言語は *principal types* 性という有用な性質を持っています。principal types 性とは、型推論がいかに効果的であるかの強い保証を与えます。残念ながら、Coq の型システムは表現力がとても豊かであるがために「完全な」型推論は不可能で、この課題は実際的にも困難です。それでもやはり Coq はとても便利なヒューリスティクスを含んでおり、それらの多くは Coq の単純なコードに落ちるようなプログラムに対する ML や Haskell の型検査器の仕組みを模倣しています。

この機会に Coq に関連した様々な種類の言語について触れましょう。Coq の理論的基礎は *Calculus of Inductive Constructions* (CIC) [28] と呼ばれる形式システムに基いています。CIC は *Calculus of Constructions* (CoC) [8] という型システムの拡張です。CIC はメタ理論を証明するのに有用ですが実際の開発には少し厳格すぎるような基礎理論です。しかしながら、CIC は強正規化性 (strong normalizaiton) や Zermelo-Fraenkel 集合論の類似システムに対する相対無矛盾性 (relative consistency) のような性質が証明されていることは知っておいて良いでしょう。強正規化性は、すべてのプログラムが (さらに重要なことに、すべての証明項も) 停止するという性質で、相対無矛盾性は簡単に言えば Coq で書かれた証明は対応する数学的命題が「本当に正しい」ことを信じてよいという性質です。

Coq は本当は Gallina と呼ばれる CIC の拡張の上に基いています。上のコードの := から . までの中身は Gallina の項です。Gallina は CIC の拡張として考慮されなければいけない有用な特徴を含んでいます。CIC についての重要なメタ定理は形式言語の範囲を越えた特徴の一部にまでは拡張されていませんが、ほとんどの Coq ユーザはこの欠落をさほど気にしていません。

さらに、Coq は証明を書いたり手続きを決定するためのドメイン固有言語である Ltac を含みます。この章の後半でいくつかの基本的な Ltac の例を見ていき、本書のほとんどはさらに有用な Ltac の例を挙げることに捧げます。

最後に、`Inductive` や `Definition` のようなコマンドは Vernacular の一部です。Vernacular は Coq システムに対するあらゆる種類の有用な要求や命令を含みます。すべての Coq のソースファイルは Vernacular コマンドの列であり、たくさんのコマンドは Gallina や Ltac プログラムを引数に取ります (実際は、Coq のソースファイルはネストされたスコープ構造の影響で、列ではなく木に近い形をしています)。

式の意味の簡単な定義を与えましょう：

```
Fixpoint expDenote (e : exp) : nat :=
  match e with
    | Const n ⇒ n
```

```
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
end.
```

Fixpoint キーワードを使って、これは再帰的定義をしていることを明示的に宣言しています。残りの部分は関数型プログラマにとっては目新しいものではないでしょう。

これらの定義の性質の証明をする前に、テストができれば好都合です。コマンド Eval を使っていくつかの例を評価し、私たちのセマンティクスがもっともらしいことを確かめてみましょう。このコマンドは「簡約戦略」(reduction strategy)、別の言葉で「評価順序」(order of evaluation) を表す引数を取ります。ML の先行評価や、Haskell の遅延評価とは違い、Coq ではこれらや他の様々な評価順序を選べます。これが可能なのはすべての Coq プログラムが停止するからです。実は、Coq は内部で上の Fixpoint で定義した関数の停止性をチェックしています。Coq は再帰呼び出しごとに引数のサイズが単調減少していることを見て、停止性を判断しています。さらに言うと、再帰呼び出しは match 式によって分割された元々の引数によって作られていないといけません。(In Chapter 7, we will see some ways of getting around this restriction, though simply removing the restriction would leave Coq useless as a theorem proving tool, for reasons we will start to learn about in the next chapter.)

評価のテストをするために、評価戦略 simpl を使って Eval コマンドを実行しましょう。simpl の定義は Coq の基礎をもっと学むまで後回しにしますが、simpl は通常テストを終わらせてくれます。

```
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
```

どれも自然な結果でしょう。これで、私たちのコンパイラのターゲット言語の定義に移る準備ができました。

2.1.2　ターゲット言語

今まで定義してきたソースプログラムを簡単なスタックマシン上へコンパイルします。ターゲット言語のシンタックスは以下で与えます：

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

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

命令 **instr** はスタックの先頭に定数をプッシュする iConst か、引数二つをポップし二項演算子に適用した後スタックに結果をプッシュする *iBinon* から成ります。ここでのプログラム prog は命令 **instr** のリストで、スタック stack は自然数のリストです。

命令の意味をスタックからスタックのオプション型への関数として与えましょう。命令を実行してスタックアンダーフローに陥った場合は None、結果として新たなスタック *s'* を

得た場合は Some s' を返します。中置演算子 :: はリストの cons で、Coq の標準ライブラリで定義されています。

```
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.
```

instrDenote が定義されれば、関数 progDenote も簡単に定義できます。プログラム全体に対して instrDenote を繰り返し適用させます：

```
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.
```

こうして二つのプログラミング言語が定義されたので、コンパイラの定義に移りましょう。

2.1.3 変換

私たちのコンパイラは自然に定義されます。リストの結合 ++ は Coq の標準ライブラリにあります。

```
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.
```

このコンパイラが正しいことを証明する前に、先ほどのサンプルプログラムを使っていくつかテストを走らせてみましょう。

```
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

コンパイルされたプログラムも実行し、それらが正しい結果を返すこともチェックしましょう。

```
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
```

今のところ良いですが、どうすればすべての入力プログラムに対してコンパイラが正しく動作することを確かめればよいでしょうか？

2.1.4 変換の正しさ

コンパイラが正しく実装されたことを証明しましょう。証明を始めるためには新たな Vernacula コマンド Theorem を使います。先ほど定義したセマンティクスを用いて変換の正しさを証明しましょう。

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

紙と鉛筆の証明なら「e に関する帰納法より」と書いて終わらせるかもしれませんが、この証明は直接取り組むのは懸命ではありません。ここでは基本的な手法である帰納法の仮定の強化をする必要があります。そのために、Lemma コマンドを使って補題を示しましょう。Lemma コマンドは Theorem のシノニムで、慣習的に主定理の証明に必要となる補助的な定理に対して使います。

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

Lemma コマンドを読み込むと、対話的証明モード (interactive proof-editing mode) に入ります。スクリーンに何やら新しいテキストが表示されるのが見えるでしょう：

```
1 subgoal

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

Coq は補題の証明を始めようとしています。ここに見えているテキストは、私たちが証明のどこにいるのかを部分的に表しています。今、私たちには証明のゴールが一つあることを伝えられています。一般に、証明の途中で、複数の未証明の部分的なゴールが与えられることがあります。こういったゴールのことを サブゴールと呼び、それらは論理的な命題で

す。複数のサブゴールはどんな順番で証明してもよいですが、通常は Coq の与えた順番で証明するのが良いでしょう。

出力には私たちの一つのサブゴールが完全な詳細とともに書かれています。二重線の上には自由変数や仮定が (もしあれば) 示されます。二重線の下は一般的に、仮定を使って証明されるべき結論が書かれています。

証明の状態はタクティクと呼ばれるコマンドを実行することで操作できます。もっとも重要なタクティクの一つである `induction` から始めましょう。

`induction` e.

今、式 e の構造の帰納法によってこの証明を始めることが宣言されました。始めのサブゴールは、帰納法による証明のための二つの新しいサブゴールに変わりました：

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)

一つ目のサブゴールには二重線と、その上に自由変数や仮定も表示されますが、それ以降のサブゴールは結論だけが表示されます。今自由変数の例が一つ目のサブゴールに見えますね。**nat** 型の自由変数 n です。結論は、元の定理内の e が Const n に置き換えられています。同様に、二つ目のサブゴールの e はコンストラクタ Binop の一般的な形に置き換えられています。この両方のサブゴールを証明することは、構造的帰納法による標準的な証明に対応します。

一つ目のサブゴールの証明を新しいタクティクから始めましょう。次のタクティクは非常によく使われます。

`intros`.

サブゴールは次のように変わります：

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

intros は、ゴールの先頭にあった ∀ によって束縛された変数を自由変数に変えました。
さらに証明を進めるためには、ゴール内のいくつかの関数の定義を使う必要があります。unfold タクティクは識別子をその定義に置き換えます。

```
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)
```

ゴールを証明するには一つ目の progDenote を展開 (unfold) する必要があります。at 節は unfold と共に使われ、識別子を特定の箇所のみを展開したい場合にその場所を指定します。場所は左から右に数えます。

```
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)
```

今回の unfold は progDenote を無名の再帰関数に変えました (一般に fun や “lambda” が再帰しない無名関数を与えるのと同様に)。これは、再帰的定義を展開するときに一般に

起こります。ここで、Coq は引数 p, s を $p0$, $s0$ へ自動的に変えたことに注意してください。局所的な自由変数と名前の衝突を避けるためです。また、他にも None (A:=stack) という部分項が見えますね。この項は自身が **option** stack 型を持つということを指示する注釈を含んでいます。このことを **option** の定義内の型変数 A の明示的具体化と呼びます。

幸いなことに、今のケースではこの複雑な無名再帰関数をすぐに除くことができます。これは、引数である (iConst n :: nil) ++ p の構造が、`simpl` タクティクを使って内部のパターンマッチを簡約することで明らかになるからです。`simpl` タクティクは先ほど `Eval` と共に使ったものと同じ簡約戦略を適用します (詳細はまだ触れません)。

```
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)

これで progDenote の定義を折り畳むことができます：

```
fold progDenote.
```

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

自明な等式になったので、このケースの証明はこれで終わりのように見えます。実際、次のタクティクを使えば証明は終わります：

```
reflexivity.
```

二つ目のサブゴールに入ります：

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)
```

初めての「仮定」の例が二重線の上に見えますね。部分項 *e1*, *e2* に対応する帰納法の仮定 *IHe1*, *IHe2* です。

前回と同じように、自由変数を導入 (`intro`duce) し、適切な定義を展開 (`unfold`) し折り畳み (`fold`) ます。`unfold`/`fold` は一見つまらないことをやっているように見えますが、実は `unfold` は時折簡単な簡約を行うので、実に有益に働きます。

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

今、私たちはこれまで見てきたタクティクでは不十分な地点に着きました。もう定義の展開は不要なので、他のことを試す必要があります。

```
  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)
```

今必要なのは、リストの結合に関する結合律 (associative law) です。これは標準ライブラリで定理 app_assoc_reverse として利用できます。(Here and elsewhere, it is possible to tell the difference between inputs and outputs to Coq by periods at the ends of the inputs.)

```
Check app_assoc_reverse.
```

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

もし使いたい定理の名前を知らなければ、`SearchRewrite` コマンドを使って検索できま

す。SearchRewrite は以下のように書き換えたいパターンを入力して使います：

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

app_assoc_reverse:
  $\forall$ (A : Type) (l m n : **list** A), $(l ++ m) ++ n = l ++ m ++ n$

app_assoc: $\forall$ (A : Type) (l m n : **list** A), $l ++ m ++ n = (l ++ m) ++ n$

app_assoc_reverse で書き換えを行いましょう：

```
rewrite app_assoc_reverse.
```

結論は以下のように変わります：

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

今、等式の左辺は二つ目の帰納法の仮定内の等式の左辺に一致していることが分かります。よってその仮定も書き換えに使えます。

```
rewrite IHe2.
```

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

同様のプロセスで残りの仮定も適用できます。

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

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

これで、先ほど終わらせた一つ目の証明と同様のタクティクを適用していくことができます。

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

これで、以下のメッセージと共に証明が完了しました：

```
Proof completed.
```

私たちの最初の証明ができました。既に、このような単純な定理に対しても、証明のスクリプトは構造化されておらず、あまり読者に教育的ではありません。もしこのアプローチをもっと本格的な定理に拡張しようとすれば、証明のスクリプトは可読性が低く、タクティク・ベースの証明に反対する人々には都合のいい批判の的となるでしょう。幸いなことに、Coq はスクリプトによる高機能な自動化をサポートしており、この補題に対して短い証明を

与えることができます (自動化のタクティクは別の場所で定義しています)。これまで書いてきた証明の試みを中止し、新しく初めましょう。

```
Abort.

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

必要なのは帰納法による証明の決まり文句を書いて、残りの長々しい推論を自動化するタクティクを呼ぶことだけです。今回の証明ではタクティクの終わりでピリオドの変わりにセミコロンが使われています。セミコロンは二つのタクティクの間に使い、証明を構造化し合成します。タクティク *t1*; *t2* は *t1* を適用し、その後残される各サブゴールに *t2* を適用します。セミコロンは効果的な証明の自動化のための基本的な構成要素の一つです。ピリオドは証明途中の確認すべき状態がどこにあるのかを予め調べるには便利です。しかし複雑な証明は最終的には、セミコロンなどを使って一つのタクティクに合成し、ピリオドが一つだけになるようにすべきです。

crush タクティクは本書に付随したライブラリにあり、Coq の標準ライブラリ内のものではありません。本書のライブラリは証明の高度な自動化にとても役立つタクティクを他にもいくつか含んでいます。

Qed コマンドは証明が実際に完了していることを確かめ、そうであればその証明を保存します。これまで書いてきたタクティクたちは証明スクリプト、別の言葉で言えば Ltac プログラムの列で、これは正しく型付けされた Gallina の項です。定理が正しいことは、証明スクリプト自体ではなく、証明項が正しいことの (比較的単純な) 検査器のみで信用できます。本書の第 1 部では証明を Gallina の項として表現することの原理について紹介します。

主定理は今、容易に証明できます。うまくセミコロンを使い、ピリオド四つで証明をします。この証明はより簡単に進みます。

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

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

ここで、左辺を compile_correct' の主張に合うように書き換えましょう。標準ライブラリの以下の定理が有効です：

```
Check app_nil_end.
```

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

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

結論にはリストが複数個現れているので、定理内の変数 *l* の値を明示しました。どれを

書き換えたいかを明示しなければ、rewrite タクティクは別の場所を選んで書き換えてしまうことがあります。

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

これで補題が適用できます。

```
rewrite compile_correct'.
```

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

ほとんどが終わりました。左辺と右辺はシンプルな記号的評価によって一致するように見えます。Coq は記号的評価によって同じ結果に正規化されるものはいつでも同じ項として見なします。progDenote の定義よりここでのケースも同様です。詳細は気にせずとも、reflexivity タクティクはこの正規化をし左辺と右辺が構文的に等しいことを確かめます。\index{tactics!reflexivity}

```
  reflexivity.
Qed.
```

この証明はより短くでき自動化されますが、これは読者への演習問題としましょう。

2.2 型付き式

この節では、安全のため項の静的片付けを持つような式の構造を追加した最初の例を作ります。

2.2.1 ソース言語

式を区別するための型の自明な言語を定義します：

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

ほとんどのプログラミング言語と同様に、Coq は変数名の大文字と小文字を区別します。よって今定義された型 **type** は先ほど多相的な定理の主張の中で見た Type キーワード (詳細は後で述べます) とは異なります。 また、コンストラクタの Nat, Bool も標準ライブラリ内の型 **nat**, **bool** とは異なります。

拡張された二項演算子のセットを定義しましょう。

```
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.

tbinop の定義は **binop** と重要な意味で異なります。**binop** は `Set` 型を持つと宣言されましたが、**tbinop** は **type** → **type** → **type** → `Set` 型と宣言しました。**tbinop** は *indexed type family* として定義します。Indexed inductive types は Coq の表現力の核で、私たちの興味のあるほとんどのものはこれで定義されます。

tbinop の直感的な説明は、**tbinop** $t1\ \ t2\ \ t$ は型 $t1$, $t2$ のオペランドを取り、型 t の結果を返す二項演算子です。たとえば、コンストラクタ TLt (自然数の順序 ≦) は型 **tbinop** Nat Nat Bool を持ち、引数が自然数、結果がブール値であることを意味します。TEq の型は多相性によって少し複雑になっています。TEq は同じ型を持つ値を任意に取れるようにしているのです。

ML や Haskell は添字付けされた代数的データ型を持ちます。たとえば、ML や Haskell のリスト型はリストの要素の型によって添字付けられています。しかしながら、ML や Haskelll 98 は Coq に比べるとデータ型の定義に関して二つの大きな制限があります。

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) [45] are a popular feature in GHC Haskell, OCaml 4, and other languages that removes this first restriction.

二つ目の制限は GADTs でも制限されたままです。ML や Haskell では、型の添字は必ず型であって、式であってはいけません。Coq では、型は任意の Gallina 項により添字付けできます。型添字はプログラムと同じ領域に住むことができ、それらは通常のプログラムと同様に計算できます。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.

同様にして、片付き式に対して型族を定義できます。型 **texp** t を持つ項は対象言語の型 t を割り当てられます。(対話的定理証明の世界では慣習的に、証明支援器の言語をメタ言語と呼び、形式化されている言語を対象言語と呼びます。)

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

依存型のおかげで、構成から、すべての well-typed な **texp** は well-typed なソース言語の式を表します。これは私たちがこれから式についてしたい様々なことに対してとても便利であることが分かります。たとえば、今までのようなセマンティクスを定義するインタプリタのアプローチに適合させるのが簡単です。まず、オブジェクト言語の型を Coq の型に移す写像を定義します：

`Definition` typeDenote (t : **type**) : `Set` :=
  `match` t `with`
    | Nat ⇒ **nat**

```
    | Bool ⇒ bool
  end.
```

ここで、いくつかの事実について触れておきましょう。「プログラムの型」の型である Set はそれ自身がファーストクラスの型で、私たちは Set を返す関数を書くことができます。typeDenote の定義は明白で、Coq 標準ライブラリの型 **nat**, **bool** を使っています。私たちの二項演算子は、標準ライブラリ内の比較関数 eqb, beq_nat や leb を使って定義できます。それぞれ、ブール値間、自然数値間のイコール、自然数の≦を表します。

```
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 ⇒ beq_nat
    | TEq Bool ⇒ eqb
    | TLt ⇒ leb
  end.
```

この関数は先ほど定義した表示関数と比べていくつか違いがあります。まず、**tbinop** は添字付けされた型なので、その添字は tbinopDenote の追加の引数になります。次に、 This function has just a few differences from the denotation functions we saw earlier. First, **tbinop** is an indexed type, so its indices become additional arguments to tbinopDenote. Second, we need to perform a genuine *dependent pattern match*, where the necessary *type* of each case body depends on the *value* that has been matched. At this early stage, we will not go into detail on the many subtle aspects of Gallina that support dependent pattern-matching, but the subject is central to Part II of the book.

同じ手法により式の表示関数も自然に定義できます。TBinop コンストラクタの **type** 型の引数はパターンマッチの中で明示的に含めなければいけませんが、ここではそれらの引数を直接参照する必要はないのでアンダースコアを書きます。

```
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.
```

このセマンティクスが正しいことを確かめるためにいくつかのプログラムの例を評価します。

```
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
```

今、コンパイルのための適切なスタックマシンを定義する準備ができました。 Now we are ready to define a suitable stack machine target for compilation.

2.2.2 ターゲット言語

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* as an instruction to Coq's extensible parser. In particular, this directive applies to the whole `match` expression, which we ask to be parsed as though it were a type, so that the operator × is interpreted as Cartesian product instead of, say, multiplication. (Note that this use of *type* has no connection to the inductive type **type** that we have defined.)

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. We use a special form of `let` to destructure a multi-level tuple.

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

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

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

```
end.
```

The Coq type checker complains that:

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

This and other mysteries of Coq dependent typing we postpone until Part II of the book. The upshot of our later discussion is that it is often useful to push inside of `match` branches those function parameters whose types depend on the type of the value being matched. Our later, more complete treatment of Gallina's typing rules will explain why this helps.

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

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

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

2.2.3 Translation

To define 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
     : forall t : type, texp t -> forall 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)
```

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

2.2.4 Translation Correctness

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

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

Again, we need to strengthen the theorem statement so that the induction will go through. This time, to provide an excuse to demonstrate different tactics, I 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" [35] 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_assoc_reverse theorem that we used to rewrite the goal in the last section. We can abort this proof and prove such a lemma about tconcat.

```
Abort.
```

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

This one goes through completely automatically.

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

```
Hint Rewrite tconcat_correct.
```

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

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

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

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

```
Hint Rewrite tcompile_correct'.
```

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

It is probably worth emphasizing that we are doing more than building mathematical models. Our compilers are functional programs that can be executed efficiently. One strategy for doing so is based on *program extraction*, which generates OCaml code from Coq developments. For instance, we run a command to output the OCaml version of tcompile:

```
Extraction tcompile.
```

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

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

This chapter has been a whirlwind tour through two examples of the style of Coq development that I advocate. Parts II and III of the book focus on the key elements of that style, namely dependent types and scripted proof automation, respectively. Before we get there, we will spend some time in Part I on more standard foundational material. Part I may still be of interest to seasoned Coq hackers, since I follow the highly automated proof style even at that early stage.

Part I

Basic Programming and Proving

Chapter 3

Introducing Inductive Types

The logical foundation of Coq is the Calculus of Inductive Constructions, or CIC. In a sense, CIC is built from just two relatively straightforward features: function types and inductive types. From this modest foundation, we can prove essentially 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. Most of our examples reproduce functionality from the Coq standard library, and I have tried to copy the standard library's choices of identifiers, where possible, so many of the definitions here are already available in the default Coq environment.

The last chapter took a deep dive into some of the more advanced Coq features, to highlight the unusual approach that I advocate in this book. However, from this point on, we will rewind and go back to basics, presenting the relevant features of Coq in a more bottom-up manner. A useful first step is a discussion of the differences and relationships between proofs and programs in Coq.

3.1 Proof Terms

Mainstream presentations of mathematics treat proofs as objects that exist outside of the universe of mathematical objects. However, for a variety of reasoning tasks, it is convenient to encode proofs, traditional mathematical objects, and programs within a single formal language. Validity checks on mathematical objects are useful in any setting, to catch typos and other uninteresting errors. The benefits of static typing for programs are widely recognized, and Coq brings those benefits to both mathematical objects and programs via a uniform mechanism. In fact, from this point on, we will not bother to distinguish between programs and mathematical objects. Many mathematical formalisms are most easily encoded in terms of programs.

Proofs are fundamentally different from programs, because any two proofs of a theorem are considered equivalent, from a formal standpoint if not from an engineering standpoint. However, we can use the same type-checking technology to check proofs as we use to validate our programs. This is the *Curry-Howard correspondence* [9, 13], an approach for relating proofs and programs. We represent mathematical theorems as types, such that a theorem's proofs are exactly those programs that type-check at the corresponding type.

The last chapter's example already snuck in an instance of Curry-Howard. We used the token $\rightarrow$ to stand for both function types and logical implications. One reasonable conclusion upon seeing this might be that some fancy overloading of notations is at work. In fact, functions and implications are precisely identical according to Curry-Howard! That is, they are just two ways of describing the same computational phenomenon.

A short demonstration should explain how this can be. The identity function over the natural numbers is certainly not a controversial program.

```
Check (fun x : nat => x).
  : nat -> nat
```

Consider this alternate program, which is almost identical to the last one.

```
Check (fun x : True => x).
  : True -> True
```

The identity program is interpreted as a proof that **True**, the always-true proposition, implies itself! What we see is that Curry-Howard interprets implications as functions, where an input is a proposition being assumed and an output is a proposition being deduced. This intuition is not too far from a common one for informal theorem proving, where we might already think of an implication proof as a process for transforming a hypothesis into a conclusion.

There are also more primitive proof forms available. For instance, the term I is the single proof of **True**, applicable in any context.

```
Check I.
  : True
```

With I, we can prove another simple propositional theorem.

```
Check (fun _ : False => I).
  : False -> True
```

No proofs of **False** exist in the top-level context, but the implication-as-function analogy gives us an easy way to, for example, show that **False** implies itself.

```
Check (fun x : False => x).
  : False -> False
```

Every one of these example programs whose type looks like a logical formula is a *proof term*. We use that name for any Gallina term of a logical type, and we will elaborate shortly on what makes a type logical.

In the rest of this chapter, we will introduce different ways of defining types. Every example type can be interpreted alternatively as a type of programs or proofs.

One of the first types we introduce will be **bool**, with constructors true and false. Newcomers to Coq often wonder about the distinction between **True** and true and the distinction between **False** and false. One glib answer is that **True** and **False** are types, but true and false are not. A more useful answer is that Coq's metatheory guarantees that any term of type **bool** *evaluates* to either true or false. This means that we have an *algorithm* for answering any question phrased as an expression of type **bool**. Conversely, most propositions do not evaluate to **True** or **False**; the language of inductively defined propositions is much richer than that. We ought to be glad that we have no algorithm for deciding our formalized version of mathematical truth, since otherwise it would be clear that we could not formalize undecidable properties, like almost any interesting property of general-purpose programs.

3.2 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. We can verify the types of the two identifiers we introduce:

```
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*. Recall from the last section that 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 used in programming, 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).

The definition of **unit** places the type in Set. By replacing Set with Prop, **unit** with **True**, and tt with I, we arrive at precisely the definition of **True** that the Coq standard library employs! The program type **unit** is the Curry-Howard equivalent of the proposition **True**. We might make the tongue-in-cheek claim that, while philosophers have expended much ink on the nature of truth, we have now determined that truth is the **unit** type of functional programming.

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. At least within Coq's logical foundation of constructive logic, which we elaborate on more in the next chapter, 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. It turns out that **Empty_set** is the Curry-Howard equivalent of **False**. As for why **Empty_set** starts with a capital letter and not a lowercase letter like **unit** does, we must refer the reader to the authors of the Coq standard library, to which we try to be faithful.

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 negb (b : bool) : bool :=
  match b with
    | true ⇒ false
    | false ⇒ true
  end.
```

An alternative definition desugars to the above, thanks to an if notation overloaded to work with any inductive type that has exactly two constructors:

```
Definition negb' (b : bool) : bool :=
  if b then false else true.
```

We might want to prove that negb is its own inverse operation.

```
Theorem negb_inverse : ∀ b : bool, negb (negb b) = b.
  destruct b.
```

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

```
  2 subgoals

  ============================
   negb (negb true) = true

subgoal 2 is

 negb (negb 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 negb_ineq : ∀ b : bool, negb b ≠ b.
  destruct b; discriminate.
Qed.
```

The `discriminate` tactic 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
```

That is, to prove that a property describes all **bool**s, prove that it describes both true and false.

There is no interesting Curry-Howard analogue of **bool**. Of course, we can define such a type by replacing `Set` by `Prop` above, but the proposition we arrive at is not very useful. It is logically equivalent to **True**, but it provides two indistinguishable primitive proofs, true and false. In the rest of the chapter, we will skip commenting on Curry-Howard versions of inductive definitions where such versions are not interesting.

3.3 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.
```

The constructor 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 with `destruct` as for simpler inductive types, but 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 requires 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 $\mathsf{S}\ n = \mathsf{S}\ n$.

```
reflexivity.
```

Not much really went on in this proof, so the *crush* tactic from the CpdtTactics 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.
```

The `injection` tactic 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. This tactic attempts a variety of single proof steps, drawn from a user-specified database that we will later see how to extend.

There is also a very useful tactic called `congruence` that can prove this theorem immediately. The `congruence` tactic 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 over **nat_list** 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" type as an inductive type. For example, here are binary trees of naturals.

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

Here are two functions whose intuitive explanations are not so important. The first one computes the size of a tree, and the second performs some sort of splicing of one tree into the leftmost available leaf node of another.

```
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.

Hint Rewrite n_plus_O plus_assoc.

Theorem nsize_nsplice : ∀ tr1 tr2 : nat_btree, nsize (nsplice tr1 tr2)
  = plus (nsize tr2) (nsize tr1).
  induction tr1; crush.
Qed.
```

It is convenient that these proofs go through so easily, but it is still useful to look into the details of what happened, by checking the statement of the tree induction principle.

```
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
```

We have the usual two cases, one for each constructor of **nat_btree**.

3.4 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.

Implicit Arguments Nil [T].
```

After we end the section, the `Variable`s we used are added as extra function parameters for each defined identifier, as needed. With an `Implicit` *Arguments* command, we ask that T be inferred when we use Nil; Coq's heuristics already decided to apply a similar policy to Cons, because of the `Set Implicit` *Arguments* command elided at the beginning of this chapter. We verify that our definitions have been saved properly 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 T
```

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, despite a very real sense in which the type T is an argument to the constructor Cons, the inductive case in the type of list_ind (i.e., the third line of the type) includes no quantifier for T, even though all of the other arguments are quantified explicitly. Parameters in other inductive definitions are treated similarly in stating induction principles.

3.5 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.
```

This invocation of `Scheme` asks for the creation of induction principles even_list_mut for the type **even_list** and odd_list_mut for the type **odd_list**. The `Induction` keyword says we want standard induction schemes, since `Scheme` supports more exotic choices. Finally, `Sort Prop` establishes that we really want induction schemes, not recursion schemes, which are the same according to Curry-Howard, save for the `Prop`/`Set` distinction.

```
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)) →
```

$\forall$ e : **even_list**, P e

This is the principle we wanted in the first place.

The Scheme command is for asking Coq to generate particular induction schemes that are mutual among a set of inductive types (possibly only one such type, in which case we get a normal induction principle). In a sense, it generalizes the induction scheme generation that goes on automatically for each inductive definition. Future Coq versions might make that automatic generation smarter, so that Scheme is needed in fewer places. In a few sections, we will see how induction principles are derived theorems in Coq, so that there is not actually any need to build in *any* automatic scheme generation.

There is one more wrinkle left in using the even_list_mut induction principle: 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.
```

Here we use apply, which is one of the most essential basic tactics. When we are trying to prove fact P, and when *thm* is a theorem whose conclusion can be made to match P by proper choice of quantified variable values, the invocation apply *thm* will replace the current goal with one new goal for each premise of *thm*.

This use of apply may seem a bit *too* magical. To better see what is going on, we use a variant where we partially apply the theorem nat_ind to give an explicit value for the predicate that gives our induction hypothesis.

```
  Undo.
  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.

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 is not a good idea to assume that a proof assistant can infer extra predicates, so this way of applying mutual induction is about as straightforward as we may hope for.

3.6 Reflexive Types

A kind of inductive type called a *reflexive type* includes at least one constructor that takes as an argument *a function returning the same type we are defining*. One very useful class of examples is in modeling variable binders. Our example will be an encoding of the syntax of first-order logic. Since the idea of syntactic encodings of logic may require a bit of acclimation, let us first consider a simpler formula type for a subset of propositional logic. We are not yet using a reflexive type, but later we will extend the example reflexively.

```
Inductive pformula : Set :=
| Truth : pformula
| Falsehood : pformula
| Conjunction : pformula → pformula → pformula.
```

A key distinction here is between, for instance, the *syntax* Truth and its *semantics* **True**. We can make the semantics explicit with a recursive function. This function uses the infix operator ∧, which desugars to instances of the type family **and** from the standard library. The family **and** implements conjunction, the `Prop` Curry-Howard analogue of the usual pair type from functional programming (which is the type family **prod** in Coq's standard library).

```
Fixpoint pformulaDenote (f : pformula) : Prop :=
  match f with
    | Truth ⇒ True
    | Falsehood ⇒ False
    | Conjunction f1 f2 ⇒ pformulaDenote f1 ∧ pformulaDenote f2
  end.
```

This is just a warm-up that does not use reflexive types, the new feature we mean to introduce. When we set our sights on first-order logic instead, it becomes very handy to give constructors recursive arguments that are functions.

```
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 the syntax of 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; only some of them are legal.

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) [32], 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. Our candidate definition above violates the positivity requirement because it involves an argument of type **term** → **term**, where the type **term** that we are defining appears to the left of an arrow. The candidate type of App is fine, however, since every occurrence of **term** is either a constructor argument or the final result type.

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 final chapter, on programming language syntax and semantics.

3.7 An Interlude on Induction Principles

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. A close look at the details here will help us construct induction principles manually, which we will see is necessary for some more advanced inductive definitions.

```
Print nat_ind.
```

```
nat_ind =
fun P : nat → Prop ⇒ nat_rect P
     : ∀ P : nat → Prop,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

We see that this induction principle is defined in terms of a more general principle, nat_rect. The rec stands for "recursion principle," and the t at the end stands for Type.

```
Check nat_rect.
```

```
nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

The principle nat_rect gives P type **nat** → Type instead of **nat** → Prop. This 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 **nat** automatically:

```
Print nat_rec.
```

```
nat_rec =
fun P : nat → Set ⇒ nat_rect P
     : ∀ P : nat → Set,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

This is identical to the definition for nat_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:

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

Definition plus_rec : nat → nat → nat :=
  nat_rec (fun _ : nat ⇒ nat → nat) (fun m ⇒ m) (fun _ r m ⇒ S (r m)).

Theorem plus_equivalent : plus_recursive = plus_rec.
  reflexivity.
Qed.
```

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

```
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
```

The only new wrinkles here are, first, an anonymous recursive function definition, using the `fix` keyword of Gallina (which is like `fun` with recursion supported); and, second, the annotations on the `match` expression. This is a *dependently typed* pattern match, because the *type* of the expression depends on the *value* being matched on. We will meet more involved examples later, especially in Part II of the book.

Type inference for dependent pattern matching is undecidable, which can be proved by reduction from higher-order unification [14]. Thus, we often find ourselves needing to annotate our programs in a way that explains dependencies to the type checker. In the example of `nat_rect`, we have an `as` clause, which binds a name for the discriminee; and a `return` clause, which gives a way to compute the `match` result type as a function of the discriminee.

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

```
Fixpoint nat_rect' (P : nat → Type)
  (HO : P O)
  (HS : ∀ n, P n → P (S n)) (n : nat) :=
  match n return P n with
    | O ⇒ HO
    | S n' ⇒ HS n' (nat_rect' P HO HS n')
  end.
```

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, which we declare with the command `Hypothesis`, which is a synonym for `Variable` that, by convention, is used for variables whose types are propositions.

```
  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'.
```

It is apparent that induction principle implementations involve some tedium but not terribly much creativity.

3.8 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 :=
| 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 parameterized 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.

As we encountered with 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,
      (∀ (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 *good* ways to incorporate nested uses of 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.

Many induction principles for types with nested uses of **list** could benefit from a unified predicate capturing the idea that some property holds of every element in a list. By defining this generic predicate once, we facilitate reuse of library theorems about it. (Here, we are actually duplicating the standard library's Forall predicate, with a different implementation, for didactic purposes.)

```
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 review the definitions of **True** and ∧, 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 ∧ takes a little more work. Coq supports user registration of arbitrary parsing rules, and it is such a rule that is letting us write ∧ instead of an application of some inductive type family. We can find the underlying inductive type with the *Locate* command, whose argument may be a parsing token.

```
Locate "/\".
```

"A /\ B" := **and** A B : *type_scope* (*default interpretation*)

```
Print and.
```

Inductive **and** $(A$: Prop$)$ $(B$: Prop$)$: Prop := conj : $A \to B \to A \wedge B$

```
For conj: Arguments A, B are implicit
```

In addition to the definition of **and** itself, we get information on implicit arguments (and some other information that we omit here). 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 previous section of this chapter.

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

  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
      | 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.
```

There is no deep theoretical reason why this program should be rejected; Coq applies incomplete termination-checking heuristics, and it is necessary to learn a few of the most important rules. The term "nested inductive type" hints at the solution to this particular problem. Just as 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
      | 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** 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
    | 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
    | 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.

```
Hint Rewrite plus_S.
```

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

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, and the hint can continue working even if the rest of the proof structure changes significantly.

3.9 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 toProp (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 toProp such that our conclusion of **False** is computationally equivalent to toProp false. Thus, the `change` tactic will let us change the conclusion to toProp false. The general form `change` e replaces the conclusion with e, whenever Coq's built-in computation rules suffice to establish the equivalence of e with the original conclusion.

```
  change (toProp false).
```

```
  H : true = false
  ============================
   toProp false
```

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

```
  rewrite ← H.
```

```
  H : true = false
  ============================
   toProp 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 walk-through 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.
```

The key piece of creativity in this theorem comes in the use of the natural number predecessor function pred. Embodied in the implementation of `injection` is a generic recipe for writing such type-specific functions.

The examples in this section illustrate an important aspect of the design philosophy behind Coq. We could certainly design a Gallina replacement that built in rules for constructor discrimination and injectivity, but a simpler alternative is to include a few carefully chosen rules that enable the desired reasoning patterns and many others. A key benefit of this philosophy is that the complexity of proof checking is minimized, which bolsters our confidence that proved theorems are really true.

Chapter 4

帰納的な述語

いわゆる「Curry-Howard 同型対応」というものがあり，それは，関数型プログラムと数学的証明の形式的な対応のことです．前の章では，このテーマの最初の導入を行いました．標準ライブラリの **unit** と **True** について非常に類似していることを見てみましょう．

```
Print unit.
```

Inductive **unit** : Set := tt : **unit**

```
Print True.
```

Inductive **True** : Prop := **I** : **True**

unit はただ 1 つの値をとる型で，**True** は常に成り立つ命題であったことを思い出してください．この 2 つの概念には表面的な違いがありますが，両者ともに Inductive を使った定義になっている点は同じです．**unit** と **True** の類似点はまだあります．**unit** を **True** に，tt を **I** に，Set を Prop に置き換えると **True** の定義になるということがわかります．最初の 2 つの違いは名前の変更なので重要ではありませんが，3 つ目の違いは重要なもので，プログラムと証明をわけるものです．Set 型の T という項があればそれはプログラムの集合を表す型で，T 型を持つ項はプログラムです．Prop 型の T という項があればそれは論理的な命題で，その証明は T 型を持ちます．12 章ではもっと詳細に Prop と Set の理論的な違いを説明します．今の所は，証明が何かということについて，一般的な感覚に基づいておくことにします．

unit という型は tt という 1 つの値を取ります．**True** という型には **I** という 1 つの証明があります．なぜこれらの型を区別するのでしょうか？抽象的な文脈でカリーハワードのことを読んだことがあっても証明工学で使ったことのない人は，実はこれら 2 つの型は区別する**べきではない**と答えるでしょう．確かにこの見方には美しさを訴えかけてくるものがあります．しかし，実用的な証明においてカリーハワードはとてもゆるく扱われるべきだということを私は言いたいのです．このような区別をした方が良い Coq 特有の理由があります．コンパイルを効率的に行うことと古典主義の数学（論理学）に存在するパラドクスを避けるためというのもありますが，私はむしろ，証明とプログラミングを一緒にしなくなるようなより一般的な法則を主張します．

議論の重要なところはだいたいこのようなものです：$A \to B$ の全ての関数が同じではありませんが，$P \to Q$ という命題の全ての証明は等しいのです．この考えは *proof irrelevance* として知られています．そしてそれを論理学において形式化すると同じ命題の異なる証明を

区別するのが難しくなります．Proof irrelevance は Gallina では互換性はあるけど推論はできません（？）．このような理論的な懸念とは別に，私は Coq で開発をするのがもっとも効果的だと考えています．プログラムと証明に別々のテクニックを使えるからです．この本のほとんどの部分はこの区別のもとに書かれています．プログラムの際は依存型の存在を前提とした標準的な関数型プログタミングのテクニックを使います．そして証明の際はカスタムの Ltac 決定手続きを書きます．

上記の見方に注意すると，この章は前章の鏡像のようなものです．帰納的な定義を使って命題を定義する方法を紹介します．所々で類似性を指摘するでしょうが，効果的な Coq ユーザーの知恵袋は命題とデータ型で交わりません．依存型をもつ証明項を構築するタクティク使っていく一方で，この章はまた依存型のひそかな紹介でもあります．依存型は興味深い帰納的命題を作る土台になっています．さらに先の章ではより手動による依存型の応用も扱って行きます．

4.1 命題論理

命題論理に出てくる結合記号の定義を簡単に見ていくことから始めましょう．これからしばらくの間は，命題変数の集合を定めた Coq セクションに入っておくことにします．Coq 用語では，命題とは単に `Prop` 型の変数です．

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

Coq では，命題の結合記号のうち最も基本的なのは含意であり，→ で表します．これは今までのほぼ全ての証明でも使ってきました．含意は，帰納的に（Inductive を使って）定義されているわけではなく，むしろ Coq に関数型のコンストラクタとして組み込まれています．既に **True** の定義は紹介してあります．次の自明な定理を通して，帰納的な述語の証明における低レベル部分の様子を説明していきます．

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

構成しようとしている帰納的な述語のコンストラクタをつかって証明を進めるために `apply` タクティクを使うことができます．時には，適用しうるコンストラクタがただ 1 つしかないことがあります．そのような場合は以下のようにショートカットが使えます．

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

また，**False** という述語もあります．これは **Empty_set** のカリーハワード鏡像です．

```
  Print False.
```

```
  Inductive False : Prop :=
```

例えば自然数に対してと同じケース分析を **False** に対しても行うことで，**False** からはあらゆる結論を導くことができます．考えるべきケースがないため，そのようなケース分析は

即時に成功しゴールを証明してしまいます．

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

無矛盾のコンテキストからは **False** の証明は作れません．証明の過程の途中に出てくる矛盾したコンテキストでは，明確な **False** の証明により矛盾を証明して進むのが通常は最も簡単です．

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

この時点で，矛盾した仮定 $2+2=5$ があるので，特定の結論には意味がないのです．このような時は `elimtype` タクティクを使います．それの完全な説明は Coq ドキュメントをあたってください．ここで目的のためには，その変種の `elimtype` **False** のみを使って結論を **False** にします．なぜなら，矛盾したコンテキストからは任意の事実が従うからです．

```
  elimtype False.
```

$H : 2+2=5$
==============================
False

ここで，算術の残っていますが，*crush* を使います．

```
  crush.
Qed.
```

論理否定は **False** に関係のある概念です．

```
Print not.
```

not $=$ `fun` A : `Prop` $\Rightarrow$ $A \to$ **False**
: `Prop` $\to$ `Prop`

上記で not は **False** への含意のことだとわかります．この事実を照明中に明示的に使用することもできます．$\neg$ P という文法（ASCII 文字のチルダを使う）で not P と展開されます．

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

==============================
$2+2=5 \to$ **False**

```
  crush.
Qed.
```

先の章で導入した連言（「かつ」のこと）もまたあります．

```
Print and.
```

Inductive **and** $(A : \texttt{Prop})$ $(B : \texttt{Prop})$: Prop := conj : $A \to B \to A \wedge B$

興味を持った読者は **and** は **prod** （ペアの型）というカリーハワード同値を持つことを確かめられるでしょう．しかし，連言を証明するにはタクティクを使ってするのが最も便利です．**and** の可換性を明示的に証明することでそれが確認できます．$\wedge$ 演算子は **and** の中置略記です．

Theorem and_comm : $P \wedge Q \to Q \wedge P$.

$P \wedge Q$ のケース分析からはじめます.

```
destruct 1.
```

$H : P$
$H0 : Q$
============================
$Q \wedge P$

連言の証明はそれぞれの連言肢の証明を含んでいます（ので仮定が P と Q になりました）．それを反映したサブゴールができました．このサブゴール $Q \wedge P$ のそれぞれの連言肢のケースに分割することで証明を進めることができます.

```
split.
```

2 subgoals

$H : P$
$H0 : Q$
============================
Q

subgoal 2 *is*

P

それぞのケースで，結論は過程と同じですので，assumption タクティクで証明を完了します.

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

Coq の選言は **or** という名前で，中置演算子 $\vee$ で使えます.

Print **or**.

Inductive **or** $(A : \texttt{Prop})$ $(B : \texttt{Prop})$: Prop :=
  or_introl : $A \rightarrow A \vee B$ | or_intror : $B \rightarrow A \vee B$

選言を証明する方法が2つあることがわかりました：1つめの連言肢を証明するか，2つめの連言肢を証明するかです．カリーハワード対応による類似物はCoqの **sum** 型です．可換性をもう一度証明することで，主要なタクティクを実演しましょう．

Theorem or_comm : $P \vee Q \rightarrow Q \vee P$.

and の証明のときと同様にケース分析から始めますが，今回はケースが2つあります．

```
  destruct 1.
```

2 subgoals

$H : P$
============================
$Q \vee P$

subgoal 2 *is*

$Q \vee P$

最初のサブゴールにおいては，2つめの選言肢を証明することで（ゴールの）選言を証明したいのだとわかります．right タクティクによってその意図を伝えることができます．

```
  right; assumption.
```

1 subgoal

$H : Q$
============================
$Q \vee P$

```
  left; assumption.
Qed.
```

命題論理の全ての証明を手動でこつこつ進めないといけないというのは残念なことです．幸運なことに，その必要はありません．最も基本的なCoqの自動化タクティクの1つであるtautoは，構成的な命題論理の完全な決定手続きです（「構成的」の意味については次の節でより詳しく説明します）．ここまで証明してきた純粋な命題論理の定理はtautoを使うことで片付けることができます．

```
Theorem or_comm' : P \/ Q -> Q \/ P.
  tauto.
Qed.
```

命題論理的な理由づけは定理の証明の重要な部品になることもたまにはありますが，他の知恵が必要になることもあります．例えば算術です．intuition タクティクは tauto の一般化で，命題論理的な推論で証明できる全てを証明できます．証明を完了させるのにさらに他の事実が必要な場合でも，intuition は命題の法則を使ってなるべく簡単にしてくれます．次の例を考えて下さい．ここでは標準ライブラリのリスト連結演算++を使っています．

```
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.
```

多くの証明構造が intuition によって生成されますが，最終的な証明はリストに関する事実のみによっています．残りのサブゴールは人間が他にどんな知恵を入れなければならないかを示しています．

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

連結されたリストの長さについての定理が必要だとわかります．その定理については前の章でも証明したし，標準ライブラリにも含まれています．

```
  rewrite app_length.
```

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

ここまで来たら，純粋な命題論理の推論でサブゴールは証明できます．すなわち，length $ls1$ + length $ls2 = 6$ を P と考え，length $ls1$ = length $ls2$ を Q と考えて命題論理のトートロジに到達します．

```
  tauto.
Qed.
```

intuition タクティクは *crush* の実装の主要なつなぎの1つです．そのため，少し助けることで定理に対する短くて自動化された証明を得ることができます．

```
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.
```

```
    crush.
  Qed.
End Propositional.
```

セクションの終わりの意味はいつもと同じです．ここで定義した命題の定理は普遍的に量化されます．

4.2 構成的とはどういうことか?

ここまで提示してきたことのなかで混乱しそうなことは，**bool** と Prop の区別です．**bool** というデータ型は true と false という 2 つの値から構成されていますが，Prop はより原始的な型で，**True** や **False** といった要素は Prop に含まれます．これら 2 つ（bool と Prop）の概念を 1 つにまとめてしまってはどうでしょうか？また，**True** と **False** という 2 つの真偽状態以外が存在するとはどのようなことでしょうか？

Coq は**構成的**あるいは**直観主義的**な論理学に基づいている，というとこから答えが出てきます．古典論理だったら，もっと馴染みがあったでしょうが．構成的な論理では，古典的な恒真命題すなわち $\neg \neg P \to P$ や $P \lor \neg P$ は常には成り立ちません．一般的には，これらの恒真命題を証明できるのは P が計算可能性の理論でいう**決定可能**の場合のみです．Coq が使っている **or** のカリーハワード埋め込みによると，$P \lor \neg P$ の証明から P の証明または $\neg P$ の証明が抽出できるということになります．我々の証明は実行可能なただの関数型プログラムなので，一般的な排中律を許してしまうと停止問題への決定手続きを与えることになってしまうのです．停止問題として例えば「このチューリングマシンは停止する」というようなものも選べます．

同様の矛盾がある状況は全ての命題は **True** または **False** に評価されるとした倍にも発生します．Coq における評価は決定可能ですので，命題も決定可能なものに限ることにしています．

ということで **bool** と Prop の区別があるのです．**bool** 型のプログラムは作った時から計算的です … 結果を決めるために実行することは常に可能です．多くの Prop は決定不能なので，**bool** に比べて Prop を使えばより表現力豊かに式を書くことができます．しかし，避けられない結末として,「真偽を確かめるために Prop を実行する」ことはできなくなってしまいます．

構成的な論理によって全ての論理結合記号を審美的な魅力のある方法で定義できるようになります．直交した帰納的な定義です．すなわち，それぞれの結合記号は単純な共通の仕組みを使って独立に定義されます．構成的であることで，プログラム抽出ということもできるようになります．そこでは，プログラムを定理として記述して証明します．我々の証明はただの関数型プログラムなので，実行できるプログラムを最終的な証明から抽出できます．これは，古典的な証明からは自然に行うことはできませんでした．

後の章で Coq のプログラム抽出機能についてもっと見ていきます．しかしここでひとつ注意をしておきましょう．先に注意したカリーハワード対応を文面通りに受け取りすぎることです．プログラムを Coq の定理証明を使って書くことはできますが，そんなことをする人はほぼいません．証明とプログラムの区別を常にしておくことは最も有用です．もしも定理を証明することでプログラムを書いてしまったら，おそらく，証明を簡単にするためにアル

ゴリズムが非効率的になってしまうでしょう．微妙な定理を証明してる時にそのような状況を心配しないといけないというのは恥ずかしいことです．ですが，おそらくそのような必要はありません．というのは，証明からプログラムを抽出するという理想は理論的な研究に限定されているからです．

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
```

(Note that here, as always, each $\forall$ quantifier has the largest possible scope, so that the type of ex_intro could also be written $\forall\ x :\ A,\ (P\ x \to$ **ex** $P)$.)

The family **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.

Here is an example of a theorem statement with existential quantification. 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 version of this document that you are reading, 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 : $\forall$ n m : **nat**, ($\exists$ x : **nat**, n + x = m) $\rightarrow$ $n \leq m$.

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`, which 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** $\rightarrow$ 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. (Readers not familiar with formal semantics should not worry about not following this paragraph!)

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 choose different arguments to the type for different constructors.

For instance, our definition **isZero** makes the predicate provable only when the argument is 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 (but worry not if not familiar with 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`$)$ $(x : A)$: $A \to$ `Prop` := eq_refl : $x = x$

Behind the scenes, uses of infix = are expanded to instances of **eq**. 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 eq_refl, 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. Another way of stating that definition is: equality is defined as the least reflexive relation.

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

`Theorem` isZero_plus : ∀ n m : **nat**, **isZero** $m \to n$ + 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. (Again, worry not if the semantics-oriented terminology from this last sentence is unfamiliar.)

```
  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. Then, within the O case of the proof, we replace the fresh variable with O. This has the net effect of decrementing each of these numbers.

```
Abort.
```

To see more clearly what is happening, we can consider the type of **isZero**'s induction principle.

```
Check isZero_ind.
```

isZero_ind
  : $\forall$ P : **nat** $\rightarrow$ Prop, P 0 $\rightarrow$ $\forall$ n : **nat**, **isZero** n $\rightarrow$ P n

In our last proof script, `destruct` chose to instantiate P as `fun` $n \Rightarrow$ S n + S n = S (S (S (S n))). You can verify for yourself that this specialization of the principle applies to the goal and that the hypothesis P 0 then matches the subgoal we saw generated. 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`.

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. The rule EvenO has 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, with a new `Hint` variant that asks to consider all constructors of an inductive type during proof search. The tactic `auto` performs exhaustive proof search up to a fixed depth, considering only the proof steps we have registered as hints.

```
Hint Constructors even.
```

```
Theorem even_4' : even 4.
```

```
  auto.
Qed.
```

We may also use inversion with **even**.

```
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
```

The inversion tactic 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. More complex inductive definitions and theorems can cause inversion to generate equalities where neither side is a variable.

```
  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 : $\forall$ m : **nat**, **even** $n \to$ **even** $m \to$ **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 : $\forall$ m : **nat**, **even** $n \to$ **even** $m \to$ **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.

Recall that tactics like `induction` and `destruct` may be passed numbers to refer to unnamed lefthand sides of implications in the conclusion, where the argument n refers to the nth such hypothesis.

```
Restart.
  induction 1.
```

m : **nat**

============================

even $m \to$ **even** $(0 + m)$

subgoal 2 is:

even $m \to$ **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** $m \to$ **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 : $\forall$ n, **even** (S $(n + n)$) $\to$ **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 easier 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. We can search for a theorem that allows us to rewrite terms of the form $x + \mathsf{S}\ y$.

```
SearchRewrite (_ + S _).

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, if we use a variant of `apply` that has a `with` clause to give instantiations of quantified variables.

```
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.

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

The original theorem now follows trivially from our lemma, using a new tactic `eauto`, a fancier version of `auto` whose explanation we postpone to Chapter 13.

```
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`. An invocation of `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. In this case, `eauto` is able to determine the right values for those unification variables, using (unsurprisingly) a variant of the classic algorithm for *unification* [37].

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.
```

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.

```
Abort.
```

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.

Chapter 5

Infinite Data and Proofs

In lazy functional programming languages like Haskell, infinite data structures are everywhere [15]. 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; several such approaches are surveyed in Chapter 7. 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 : Type.

  CoInductive stream : Type :=
  | 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_falses : stream bool := Cons true falses_trues
with falses_trues : stream bool := Cons false trues_falses.
```

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_falses 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*. 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.

Some familiar functions are easy to write in co-recursive fashion.

```
Section map.
  Variables A B : Type.
  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 the guardedness condition.

The implications of the condition can be subtle. To illustrate how, 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 : Type.

  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 : Type.
  Variable f : A → B.

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

We get another error message about an unguarded recursive call.

```
End map'.
```

What is going wrong here? Imagine that, instead of interleave, we had called some other, less well-behaved function on streams. Here is one simpler example demonstrating the essential pitfall. We start by defining a standard function for taking the tail of a stream. Since streams are infinite, this operation is total.

```
Definition tl A (s : stream A) : stream A :=
  match s with
    | Cons _ s' ⇒ s'
  end.
```

Coq rejects the following definition that uses tl.

```
CoFixpoint bad : stream nat := tl (Cons 0 bad).
```

Imagine that Coq had accepted our definition, and consider how we might evaluate approx bad 1. We would be trying to calculate the first element in the stream bad. However, it is not hard to see that the definition of bad "begs the question": unfolding the definition of tl, we see that we essentially say "define bad to equal itself"! Of course such an equation admits no single well-defined solution, which does not fit well with the determinism of Gallina reduction.

Coq's complete rule for co-recursive definitions includes not just the basic guardedness condition, but also a requirement about where co-recursive calls may occur. In particular, a co-recursive call must be a direct argument to a constructor, *nested only inside of other constructor calls or* **fun** *or* **match** *expressions*. In the definition of bad, we erroneously nested the co-recursive call inside a call to tl, and we nested inside a call to interleave in the definition of map'.

Coq helps the user out a little by performing the guardedness check after using computation to simplify terms. For instance, any co-recursive function definition can be expanded by inserting extra calls to an identity function, and this change preserves guardedness. However, in other cases computational simplification can reveal why definitions are dangerous. Consider what happens when we inline the definition of tl in bad:

```
CoFixpoint bad : stream nat := bad.
```

This is the same looping definition we rejected earlier. A similar inlining process reveals an alternate view on our failed definition of map':

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

Clearly in this case the map' calls are not immediate arguments to constructors, so we violate the guardedness condition.

A more interesting question is why that condition is the right one. We can make an intuitive argument that the original map' definition is perfectly reasonable and denotes a well-understood transformation on streams, such that every output would behave properly with approx. The guardedness condition is an example of a syntactic check for *productivity* of co-recursive definitions. A productive definition can be thought of as one whose outputs can be forced in finite time to any finite approximation level, as with approx. If we replaced the guardedness condition with more involved checks, we might be able to detect and allow a broader range of productive definitions. However, mistakes in these checks could cause inconsistency, and programmers would need to understand the new, more complex checks. Coq's design strikes a balance between consistency and simplicity with its choice of guard condition, though we can imagine other worthwhile balances being struck, too.

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 : Type.

  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 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 **stream_eq**'s constructor. 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 at 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))
```

Note the cofix notation for anonymous co-recursion, which is analogous to the fix notation we have already seen for recursion. 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. A correct proof strategy for a theorem like this usually starts 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.

Must we always be cautious with automation in proofs by co-induction? Induction seems to have dual versions of the same pitfalls inherent in it, and yet we avoid those pitfalls by encapsulating safe Curry-Howard recursion schemes inside named induction principles. It turns out that we can usually do the same with *co-induction principles*. Let us take that tack here, so that we can arrive at an `induction` *x*; *crush*-style proof for ones_eq'.

An induction principle is parameterized over a predicate characterizing what we mean to prove, *as a function of the inductive fact that we already know*. Dually, a co-induction principle ought to be parameterized over a predicate characterizing what we mean to prove, *as a function of the arguments to the co-inductive predicate that we are trying to prove*.

To state a useful principle for **stream_eq**, it will be useful first to define the stream head function.

```
Definition hd A (s : stream A) : A :=
  match s with
    | Cons x _ ⇒ x
  end.
```

Now we enter a section for the co-induction principle, based on Park's principle as introduced in a tutorial by Giménez [11].

```
Section stream_eq_coind.
  Variable A : Type.
  Variable R : stream A → stream A → Prop.
```

This relation generalizes the theorem we want to prove, defining a set of pairs of streams that we must eventually prove contains the particular pair we care about.

```
  Hypothesis Cons_case_hd : ∀ s1 s2, R s1 s2 → hd s1 = hd s2.
  Hypothesis Cons_case_tl : ∀ s1 s2, R s1 s2 → R (tl s1) (tl s2).
```

Two hypotheses characterize what makes a good choice of R: it enforces equality of stream heads, and it is "hereditary" in the sense that an R stream pair passes on "R-ness" to its tails. An established technical term for such a relation is *bisimulation*.

Now it is straightforward to prove the principle, which says that any stream pair in R is equal. The reader may wish to step through the proof script to see what is going on.

```
  Theorem stream_eq_coind : ∀ s1 s2, R s1 s2 → stream_eq s1 s2.
    cofix; destruct s1; destruct s2; intro.
    generalize (Cons_case_hd H); intro Heq; simpl in Heq; rewrite Heq.
    constructor.
    apply stream_eq_coind.
    apply (Cons_case_tl H).
  Qed.
End stream_eq_coind.
```

To see why this proof is guarded, we can print it and verify that the one co-recursive call is an immediate argument to a constructor.

```
Print stream_eq_coind.
```

We omit the output and proceed to proving ones_eq'' again. The only bit of ingenuity is in choosing R, and in this case the most restrictive predicate works.

```
Theorem ones_eq'' : stream_eq ones ones'.
  apply (stream_eq_coind (fun s1 s2 ⇒ s1 = ones ∧ s2 = ones')); crush.
Qed.
```

Note that this proof achieves the proper reduction behavior via hd and tl, rather than frob as in the last proof. All three functions pattern match on their arguments, catalyzing computation steps.

Compared to the inductive proofs that we are used to, it still seems unsatisfactory that we had to write out a choice of R in the last proof. An alternate is to capture a common pattern of co-recursion in a more specialized co-induction principle. For the current example, that pattern is: prove **stream_eq** *s1 s2* where *s1* and *s2* are defined as their own tails.

```
Section stream_eq_loop.
  Variable A : Type.
  Variables s1 s2 : stream A.
```

```
  Hypothesis Cons_case_hd : hd s1 = hd s2.
  Hypothesis loop1 : tl s1 = s1.
  Hypothesis loop2 : tl s2 = s2.
```

The proof of the principle includes a choice of R, so that we no longer need to make such choices thereafter.

```
  Theorem stream_eq_loop : stream_eq s1 s2.
    apply (stream_eq_coind (fun s1' s2' ⇒ s1' = s1 ∧ s2' = s2)); crush.
  Qed.
End stream_eq_loop.
```

```
Theorem ones_eq''' : stream_eq ones ones'.
  apply stream_eq_loop; crush.
Qed.
```

Let us put stream_eq_coind through its paces a bit more, considering two different ways to compute infinite streams of all factorial values. First, we import the fact factorial function from the standard library.

```
Require Import Arith.
Print fact.
```

```
fact =
fix fact (n : nat) : nat :=
  match n with
  | 0 ⇒ 1
  | S n0 ⇒ S n0 × fact n0
  end
     : nat → nat
```

The simplest way to compute the factorial stream involves calling fact afresh at each position.

```
CoFixpoint fact_slow' (n : nat) := Cons (fact n) (fact_slow' (S n)).
Definition fact_slow := fact_slow' 1.
```

A more clever, optimized method maintains an accumulator of the previous factorial, so that each new entry can be computed with a single multiplication.

```
CoFixpoint fact_iter' (cur acc : nat) := Cons acc (fact_iter' (S cur) (acc × cur)).
Definition fact_iter := fact_iter' 2 1.
```

We can verify that the streams are equal up to particular finite bounds.

```
Eval simpl in approx fact_iter 5.
```

```
     = 1 :: 2 :: 6 :: 24 :: 120 :: nil
     : list nat
```

```
Eval simpl in approx fact_slow 5.
```

```
     = 1 :: 2 :: 6 :: 24 :: 120 :: nil
```

: **list nat**

Now, to prove that the two versions are equivalent, it is helpful to prove (and add as a proof hint) a quick lemma about the computational behavior of fact. (I intentionally skip explaining its proof at this point.)

```
Lemma fact_def : ∀ x n,
  fact_iter' x (fact n × S n) = fact_iter' x (fact (S n)).
  simpl; intros; f_equal; ring.
Qed.

Hint Resolve fact_def.
```

With the hint added, it is easy to prove an auxiliary lemma relating fact_iter' and fact_slow'. The key bit of ingenuity is introduction of an existential quantifier for the shared parameter n.

```
Lemma fact_eq' : ∀ n, stream_eq (fact_iter' (S n) (fact n)) (fact_slow' n).
  intro; apply (stream_eq_coind (fun s1 s2 ⇒ ∃ n, s1 = fact_iter' (S n) (fact n)
    ∧ s2 = fact_slow' n)); crush; eauto.
Qed.
```

The final theorem is a direct corollary of fact_eq'.

```
Theorem fact_eq : stream_eq fact_iter fact_slow.
  apply fact_eq'.
Qed.
```

As in the case of ones_eq', we may be unsatisfied that we needed to write down a choice of R that seems to duplicate information already present in a lemma statement. We can facilitate a simpler proof by defining a co-induction principle specialized to goals that begin with single universal quantifiers, and the strategy can be extended in a straightforward way to principles for other counts of quantifiers. (Our stream_eq_loop principle is effectively the instantiation of this technique to zero quantifiers.)

```
Section stream_eq_onequant.
  Variables A B : Type.
```

We have the types A, the domain of the one quantifier; and B, the type of data found in the streams.

```
  Variables f g : A → stream B.
```

The two streams we compare must be of the forms $f\ x$ and $g\ x$, for some shared x. Note that this falls out naturally when x is a shared universally quantified variable in a lemma statement.

```
  Hypothesis Cons_case_hd : ∀ x, hd (f x) = hd (g x).
  Hypothesis Cons_case_tl : ∀ x, ∃ y, tl (f x) = f y ∧ tl (g x) = g y.
```

These conditions are inspired by the bisimulation requirements, with a more general version of the R choice we made for fact_eq' inlined into the hypotheses of stream_eq_coind.

```
  Theorem stream_eq_onequant : ∀ x, stream_eq (f x) (g x).
```

```
    intro; apply (stream_eq_coind (fun s1 s2 ⇒ ∃ x, s1 = f x ∧ s2 = g x)); crush; eauto.
  Qed.
End stream_eq_onequant.
```

```
Lemma fact_eq'' : ∀ n, stream_eq (fact_iter' (S n)) (fact n)) (fact_slow' n).
  apply stream_eq_onequant; crush; eauto.
Qed.
```

We have arrived at one of our customary automated proofs, thanks to the new principle.

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 imperative programming language and use that semantics to prove the correctness of a trivial optimization that removes spurious additions by 0. We follow the technique of *co-inductive big-step operational semantics* [19].

We define a suggestive synonym for **nat**, as we will consider programs over infinitely many variables, represented as **nat**s.

```
Definition var := nat.
```

We define a type vars of maps from variables to values. To define a function `set` for setting a variable's value in a map, we use the standard library function `beq_nat` for comparing natural numbers.

```
Definition vars := var → nat.
Definition set (vs : vars) (v : var) (n : nat) : vars :=
  fun v' ⇒ if beq_nat v v' then n else vs v'.
```

We define a simple arithmetic expression language with variables, and we give it a semantics via an interpreter.

```
Inductive exp : Set :=
| Const : nat → exp
| Var : var → exp
| Plus : exp → exp → exp.
```

```
Fixpoint evalExp (vs : vars) (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Var v ⇒ vs v
    | Plus e1 e2 ⇒ evalExp vs e1 + evalExp vs e2
  end.
```

Finally, we define a language of commands. It includes variable assignment, sequencing, and a `while` form that repeats as long as its test expression evaluates to a nonzero value.

```
Inductive cmd : Set :=
| Assign : var → exp → cmd
```

```
| Seq : cmd → cmd → cmd
| While : exp → cmd → cmd.
```

We could define an inductive relation to characterize the results of command evaluation. However, such a relation would not capture *nonterminating* executions. With a co-inductive relation, we can capture both cases. The parameters of the relation are an initial state, a command, and a final state. A program that does not terminate in a particular initial state is related to *any* final state. For more realistic languages than this one, it is often possible for programs to *crash*, in which case a semantics would generally relate their executions to no final states; so relating safely non-terminating programs to all final states provides a crucial distinction.

```
CoInductive evalCmd : vars → cmd → vars → Prop :=
| EvalAssign : ∀ vs v e, evalCmd vs (Assign v e) (set vs v (evalExp vs e))
| EvalSeq : ∀ vs1 vs2 vs3 c1 c2, evalCmd vs1 c1 vs2
  → evalCmd vs2 c2 vs3
  → evalCmd vs1 (Seq c1 c2) vs3
| EvalWhileFalse : ∀ vs e c, evalExp vs e = 0
  → evalCmd vs (While e c) vs
| EvalWhileTrue : ∀ vs1 vs2 vs3 e c, evalExp vs1 e ≠ 0
  → evalCmd vs1 c vs2
  → evalCmd vs2 (While e c) vs3
  → evalCmd vs1 (While e c) vs3.
```

Having learned our lesson in the last section, before proceeding, we build a co-induction principle for **evalCmd**.

```
Section evalCmd_coind.
  Variable R : vars → cmd → vars → Prop.

  Hypothesis AssignCase : ∀ vs1 vs2 v e, R vs1 (Assign v e) vs2
    → vs2 = set vs1 v (evalExp vs1 e).

  Hypothesis SeqCase : ∀ vs1 vs3 c1 c2, R vs1 (Seq c1 c2) vs3
    → ∃ vs2, R vs1 c1 vs2 ∧ R vs2 c2 vs3.

  Hypothesis WhileCase : ∀ vs1 vs3 e c, R vs1 (While e c) vs3
    → (evalExp vs1 e = 0 ∧ vs3 = vs1)
    ∨ ∃ vs2, evalExp vs1 e ≠ 0 ∧ R vs1 c vs2 ∧ R vs2 (While e c) vs3.
```

The proof is routine. We make use of a form of `destruct` that takes an *intro pattern* in an `as` clause. These patterns control how deeply we break apart the components of an inductive value, and we refer the reader to the Coq manual for more details.

```
  Theorem evalCmd_coind : ∀ vs1 c vs2, R vs1 c vs2 → evalCmd vs1 c vs2.
    cofix; intros; destruct c.
    rewrite (AssignCase H); constructor.
    destruct (SeqCase H) as [? [? ?]]; econstructor; eauto.
    destruct (WhileCase H) as [[? ?] | [? [? [? ?]]]]; subst; econstructor; eauto.
```

```
  Qed.
End evalCmd_coind.
```

Now that we have a co-induction principle, we should use it to prove something! Our example is a trivial program optimizer that finds places to replace $0 + e$ with e.

```
Fixpoint optExp (e : exp) : exp :=
  match e with
    | Plus (Const 0) e ⇒ optExp e
    | Plus e1 e2 ⇒ Plus (optExp e1) (optExp e2)
    | _ ⇒ e
  end.

Fixpoint optCmd (c : cmd) : cmd :=
  match c with
    | Assign v e ⇒ Assign v (optExp e)
    | Seq c1 c2 ⇒ Seq (optCmd c1) (optCmd c2)
    | While e c ⇒ While (optExp e) (optCmd c)
  end.
```

Before proving correctness of optCmd, we prove a lemma about optExp. This is where we have to do the most work, choosing pattern match opportunities automatically.

```
Lemma optExp_correct : ∀ vs e, evalExp vs (optExp e) = evalExp vs e.
  induction e; crush;
    repeat (match goal with
              | [ ⊢ context[match ?E with Const _ ⇒ _ | _ ⇒ _ end] ] ⇒ destruct E
              | [ ⊢ context[match ?E with O ⇒ _ | S _ ⇒ _ end] ] ⇒ destruct E
            end; crush).
Qed.

Hint Rewrite optExp_correct.
```

The final theorem is easy to establish, using our co-induction principle and a bit of Ltac smarts that we leave unexplained for now. Curious readers can consult the Coq manual, or wait for the later chapters of this book about proof automation. At a high level, we show inclusions between behaviors, going in both directions between original and optimized programs.

```
Ltac finisher := match goal with
                   | [ H : evalCmd _ _ _ ⊢ _ ] ⇒ ((inversion H; [])
                     || (inversion H; [|])); subst
                 end; crush; eauto 10.

Lemma optCmd_correct1 : ∀ vs1 c vs2, evalCmd vs1 c vs2
  → evalCmd vs1 (optCmd c) vs2.
  intros; apply (evalCmd_coind (fun vs1 c' vs2 ⇒ ∃ c, evalCmd vs1 c vs2
    ∧ c' = optCmd c)); eauto; crush;
    match goal with
```

```
    | [ H : _ = optCmd ?E ⊢ _ ] ⇒ destruct E; simpl in *; discriminate
      || injection H; intros; subst
  end; finisher.
Qed.

Lemma optCmd_correct2 : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  → evalCmd vs1 c vs2.
  intros; apply (evalCmd_coind (fun vs1 c vs2 ⇒ evalCmd vs1 (optCmd c) vs2));
    crush; finisher.
Qed.

Theorem optCmd_correct : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  ↔ evalCmd vs1 c vs2.
  intuition; apply optCmd_correct1 || apply optCmd_correct2; assumption.
Qed.
```

In this form, the theorem tells us that the optimizer preserves observable behavior of both terminating and nonterminating programs, but we did not have to do more work than for the case of terminating programs alone. We merely took the natural inductive definition for terminating executions, made it co-inductive, and applied the appropriate co-induction principle. Curious readers might experiment with adding command constructs like `if`; the same proof script should continue working, after the co-induction principle is extended to the new evaluation rules.

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. The techniques we will learn make it possible to reduce the cost of program verification dramatically.

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.

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

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

```
Eval compute in pred_strong1 two_gt0.
```

```
     = 1
     : nat
```

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

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

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

```
"0 > 0"
```

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

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

We are lucky that Coq's heuristics infer the `return` clause (specifically, `return` $n > 0 \rightarrow$ **nat**) for us in the definition of pred_strong1, leading to the following elaborated code:

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

By making explicit the functional relationship between value n and the result type of the `match`, we guide Coq toward proper type checking. The clause for this example follows by simple copying of the original annotation on the definition. In general, however, the `match` annotation inference problem is undecidable. The known undecidable problem of *higher-order unification* [14] reduces to the `match` type inference problem. Over time, Coq is enhanced with more and more heuristics to get around this problem, but there must always exist `match`es whose types Coq cannot infer without annotations.

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

```
Extraction pred_strong1.
```

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

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

The proof argument has disappeared! We get exactly the OCaml code we would have written manually. This is our first demonstration of the main technically interesting feature of Coq program extraction: proofs are erased 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 : Type) (P : A → Prop) : Type :=
```

exist : ∀ x : A, P x → **sig** P

The family **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
  "{ x : A | P }" := sig (fun x : A ⇒ P)

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

To build a value of a subset type, we use the exist constructor, and the details of how to do that follow from the output of our earlier `Print` **sig** command, where we elided the extra information that parameter A is implicit. We need an extra _ here and not in the definition of pred_strong2 because *parameters* of inductive types (like the predicate P for **sig**) are not mentioned in pattern matching, but *are* mentioned in construction of terms (if they are not marked as implicit arguments).

```
Eval compute in pred_strong2 (exist _ 2 two_gt0).

     = 1
     : nat

Extraction pred_strong2.
```

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

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

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

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

```
Definition pred_strong3 (s : {n : nat | n > 0}) : {m : nat | proj1_sig s = S m} :=
  match s return {m : nat | proj1_sig s = S m} with
```

```
    | exist 0 pf ⇒ match zgtz pf with end
    | exist (S n') pf ⇒ exist _ n' (eq_refl _)
  end.
```

```
Eval compute in pred_strong3 (exist _ 2 two_gt0).
```

```
     = exist (fun m : nat ⇒ 2 = S m) 1 (eq_refl 2)
     : {m : nat | proj1_sig (exist (lt 0) 2 two_gt0) = S m}
```

A value in a subset type can be thought of as a *dependent pair* (or *sigma type*) of a base value and a proof about it. The function proj1_sig extracts the first component of the pair. It 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. (Recall that False_rec is the `Set`-level induction principle for **False**, which can be used to produce a value in any `Set` given a proof of **False**.)

```
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. It may seem strange to change perspective so implicitly between programming and proving, but recall that programs and proofs are two sides of the same coin in Coq, thanks to the Curry-Howard correspondence.

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. (More formally, `Defined` marks an identifier as *transparent*, allowing it to be unfolded; while `Qed` marks an identifier as *opaque*, preventing unfolding.) 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' (eq_refl (S n'))
end
     : ∀ n : nat, n > 0 → {m : nat | n = S m}
```

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

```
Eval compute in pred_strong4 two_gt0.
```

```
     = exist (fun m : nat ⇒ 2 = S m) 1 (eq_refl 2)
     : {m : nat | 2 = S m}
```

A tactic modifier called `abstract` can be helpful for producing shorter terms, by automatically abstracting subgoals into named lemmas.

```
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); abstract crush.
Defined.
```

```
Print pred_strong4'.
```

```
pred_strong4' =
fun n : nat ⇒
match n as n0 return (n0 > 0 → {m : nat | n0 = S m}) with
| 0 ⇒
    fun _H : 0 > 0 ⇒
    False_rec {m : nat | 0 = S m} (pred_strong4'_subproof n _H)
| S n' ⇒
    fun _H : S n' > 0 ⇒
    exist (fun m : nat ⇒ S n' = S m) n' (pred_strong4'_subproof0 n _H)
end
     : ∀ n : nat, n > 0 → {m : nat | n = S m}
```

We are almost done with the ideal implementation of dependent predecessor. We can use Coq's syntax extension facility to arrive at code with almost no complexity beyond a Haskell or ML program with a complete specification in a comment. In this book, we will not dwell on the details of syntax extensions; the Coq manual gives a straightforward introduction to them.

```
Notation "!" := (False_rec _ _).
```

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

Definition pred_strong5 : ∀ n : nat, n > 0 → {m : nat | n = S m}.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ !
      | S n' ⇒ fun _ ⇒ [n']
    end); crush.
Defined.
```

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

```
Eval compute in pred_strong5 two_gt0.
```

$= [1]$
$: \{m : \textbf{nat} \mid 2 = \mathsf{S}\ m\}$

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

```
Obligation Tactic := crush.

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

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

```
Eval compute in pred_strong6 two_gt0.
```

$= [1]$
$: \{m : \textbf{nat} \mid 2 = \mathsf{S}\ m\}$

In this case, we see that the new definition yields the same computational behavior as before.

6.2 Decidable Proposition Types

There is another type in the standard library that 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}
```

Here, the constructors of **sumbool** have types written in terms of a registered notation for **sumbool**, such that the result type of each constructor desugars to **sumbool** A B. We can define some notations of our own 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.
```

```
Eval compute in eq_nat_dec 2 2.
```

$= Yes$
$: \{2 = 2\} + \{2 \neq 2\}$

```
Eval compute in eq_nat_dec 2 3.
```

$= No$
$: \{2 = 3\} + \{2 \neq 3\}$

Note that the `Yes` and `No` notations are hiding proofs establishing the correctness of the outputs.

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} + {$n \neq m$}.
  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} + {$x \neq y$}.

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.
  Defined.
End In_dec.

Eval compute in In_dec eq_nat_dec 2 (1 :: 2 :: nil).
```

= *Yes*
: {In 2 (1 :: 2 :: nil)} + { ¬ In 2 (1 :: 2 :: nil)}

```
Eval compute in In_dec eq_nat_dec 3 (1 :: 2 :: nil).
```

= *No*
: {In 3 (1 :: 2 :: nil)} + { ¬ In 3 (1 :: 2 :: nil)}

The In_dec function 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')
```

This is more or the less code for the corresponding function from the OCaml standard library.

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 failure 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 return {{m | n = S m}} with
      | O ⇒ ??
      | S n' ⇒ [|n'|]
    end); trivial.
Defined.
```

```
Eval compute in pred_strong7 2.
```

$= [|1|]$
$: \{\{m \mid 2 = \mathsf{S}\ m\}\}$

```
Eval compute in pred_strong7 0.
```

$= ??$
$: \{\{m \mid 0 = \mathsf{S}\ m\}\}$

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}
```

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.
```

```
Eval compute in pred_strong8 2.
```

```
     = [||1||]
     : {m : nat | 2 = S m} + {2 = 0}
```

```
Eval compute in pred_strong8 0.
```

```
     = !!
     : {m : nat | 0 = S m} + {0 = 0}
```

As with our other maximally expressive pred function, we arrive at quite simple output values, thanks to notations.

6.4 Monadic Notations

We can treat **maybe** like a monad [41], 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. Note that the notation definition uses an ASCII `<-`, while later code uses (in this rendering) a nicer left arrow ←.

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

The meaning of x ← *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. Again, the notation definition exposes the ASCII syntax with an operator `<--`, while the later code uses a nicer long left arrow ⟵.

```
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.
```

This example demonstrates how judicious selection of notations can hide complexities in the rich types of programs.

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 return {{t | hasType e t}} 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.
```

The lemma 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 return {t : type | hasType e t} + {∀ t, ¬ hasType e t} 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. Such a step is usually warranted when defining a recursive function with `refine`. 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.

Our new function remains easy to test:

```
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)}
```

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.

Chapter 7

General Recursion

Termination of all programs is a crucial property of Gallina. Non-terminating programs introduce logical inconsistency, where any theorem can be proved with an infinite loop. Coq uses a small set of conservative, syntactic criteria to check termination of all recursive definitions. These criteria are insufficient to support the natural encodings of a variety of important programming idioms. Further, since Coq makes it so convenient to encode mathematics computationally, with functional programs, we may find ourselves wanting to employ more complicated recursion in mathematical definitions.

What exactly are the conservative criteria that we run up against? For *recursive* definitions, recursive calls are only allowed on *syntactic subterms* of the original primary argument, a restriction known as *primitive recursion*. In fact, Coq's handling of reflexive inductive types (those defined in terms of functions returning the same type) gives a bit more flexibility than in traditional primitive recursion, but the term is still applied commonly. In Chapter 5, we saw how *co-recursive* definitions are checked against a syntactic guardedness condition that guarantees productivity.

Many natural recursion patterns satisfy neither condition. For instance, there is our simple running example in this chapter, merge sort. We will study three different approaches to more flexible recursion, and the latter two of the approaches will even support definitions that may fail to terminate on certain inputs, without any up-front characterization of which inputs those may be.

Before proceeding, it is important to note that the problem here is not as fundamental as it may appear. The final example of Chapter 5 demonstrated what is called a *deep embedding* of the syntax and semantics of a programming language. That is, we gave a mathematical definition of a language of programs and their meanings. This language clearly admitted non-termination, and we could think of writing all our sophisticated recursive functions with such explicit syntax types. However, in doing so, we forfeit our chance to take advantage of Coq's very good built-in support for reasoning about Gallina programs. We would rather use a *shallow embedding*, where we model informal constructs by encoding them as normal Gallina programs. Each of the three techniques of this chapter follows that style.

7.1 Well-Founded Recursion

The essence of terminating recursion is that there are no infinite chains of nested recursive calls. This intuition is commonly mapped to the mathematical idea of a *well-founded relation*, and the associated standard technique in Coq is *well-founded recursion*. The syntactic-subterm relation that Coq applies by default is well-founded, but many cases demand alternate well-founded relations. To demonstrate, let us see where we get stuck on attempting a standard merge sort implementation.

```
Section mergeSort.
  Variable A : Type.
  Variable le : A → A → bool.
```

We have a set equipped with some "less-than-or-equal-to" test.

A standard function inserts an element into a sorted list, preserving sortedness.

```
  Fixpoint insert (x : A) (ls : list A) : list A :=
    match ls with
      | nil ⇒ x :: nil
      | h :: ls' ⇒
        if le x h
          then x :: ls
          else h :: insert x ls'
    end.
```

We will also need a function to merge two sorted lists. (We use a less efficient implementation than usual, because the more efficient implementation already forces us to think about well-founded recursion, while here we are only interested in setting up the example of merge sort.)

```
  Fixpoint merge (ls1 ls2 : list A) : list A :=
    match ls1 with
      | nil ⇒ ls2
      | h :: ls' ⇒ insert h (merge ls' ls2)
    end.
```

The last helper function for classic merge sort is the one that follows, to split a list arbitrarily into two pieces of approximately equal length.

```
  Fixpoint split (ls : list A) : list A × list A :=
    match ls with
      | nil ⇒ (nil, nil)
      | h :: nil ⇒ (h :: nil, nil)
      | h1 :: h2 :: ls' ⇒
        let (ls1, ls2) := split ls' in
          (h1 :: ls1, h2 :: ls2)
    end.
```

Now, let us try to write the final sorting function, using a natural number "≤" test leb from the standard library.

```
Fixpoint mergeSort (ls : list A) : list A :=
  if leb (length ls) 1
    then ls
    else let lss := split ls in
      merge (mergeSort (fst lss)) (mergeSort (snd lss)).
```

```
Recursive call to mergeSort has principal argument equal to
"fst (split ls)" instead of a subterm of "ls".
```

The definition is rejected for not following the simple primitive recursion criterion. In particular, it is not apparent that recursive calls to mergeSort are syntactic subterms of the original argument *ls*; indeed, they are not, yet we know this is a well-founded recursive definition.

To produce an acceptable definition, we need to choose a well-founded relation and prove that mergeSort respects it. A good starting point is an examination of how well-foundedness is formalized in the Coq standard library.

```
Print well_founded.
```

```
well_founded =
fun (A : Type) (R : A → A → Prop) ⇒ ∀ a : A, Acc R a
```

The bulk of the definitional work devolves to the *accessibility* relation **Acc**, whose definition we may also examine.

```
Print Acc.
```

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
    Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

In prose, an element x is accessible for a relation R if every element "less than" x according to R is also accessible. Since **Acc** is defined inductively, we know that any accessibility proof involves a finite chain of invocations, in a certain sense that we can make formal. Building on Chapter 5's examples, let us define a co-inductive relation that is closer to the usual informal notion of "absence of infinite decreasing chains."

```
CoInductive infiniteDecreasingChain A (R : A → A → Prop) : stream A → Prop :=
| ChainCons : ∀ x y s, infiniteDecreasingChain R (Cons y s)
  → R y x
  → infiniteDecreasingChain R (Cons x (Cons y s)).
```

We can now prove that any accessible element cannot be the beginning of any infinite decreasing chain.

```
Lemma noBadChains' : ∀ A (R : A → A → Prop) x, Acc R x
  → ∀ s, ¬infiniteDecreasingChain R (Cons x s).
```

```
  induction 1; crush;
    match goal with
      | [ H : infiniteDecreasingChain _ _ ⊢ _ ] ⇒ inversion H; eauto
    end.
Qed.
```

From here, the absence of infinite decreasing chains in well-founded sets is immediate.

```
Theorem noBadChains : ∀ A (R : A → A → Prop), well_founded R
  → ∀ s, ¬infiniteDecreasingChain R s.
  destruct s; apply noBadChains'; auto.
Qed.
```

Absence of infinite decreasing chains implies absence of infinitely nested recursive calls, for any recursive definition that respects the well-founded relation. The `Fix` combinator from the standard library formalizes that intuition:

```
Check Fix.
```

```
Fix
     : ∀ (A : Type) (R : A → A → Prop),
       well_founded R →
       ∀ P : A → Type,
       (∀ x : A, (∀ y : A, R y x → P y) → P x) →
       ∀ x : A, P x
```

A call to `Fix` must present a relation R and a proof of its well-foundedness. The next argument, P, is the possibly dependent range type of the function we build; the domain A of R is the function's domain. The following argument has this type:

```
       ∀ x : A, (∀ y : A, R y x → P y) → P x
```

This is an encoding of the function body. The input x stands for the function argument, and the next input stands for the function we are defining. Recursive calls are encoded as calls to the second argument, whose type tells us it expects a value y and a proof that y is "less than" x, according to R. In this way, we enforce the well-foundedness restriction on recursive calls.

The rest of `Fix`'s type tells us that it returns a function of exactly the type we expect, so we are now ready to use it to implement mergeSort. Careful readers may have noticed that `Fix` has a dependent type of the sort we met in the previous chapter.

Before writing mergeSort, we need to settle on a well-founded relation. The right one for this example is based on lengths of lists.

```
Definition lengthOrder (ls1 ls2 : list A) :=
  length ls1 < length ls2.
```

We must prove that the relation is truly well-founded. To save some space in the rest of this chapter, we skip right to nice, automated proof scripts, though we postpone introducing the principles behind such scripts to Part III of the book. Curious readers may still replace semicolons with periods and newlines to step through these scripts interactively.

```
Hint Constructors Acc.

Lemma lengthOrder_wf' : ∀ len, ∀ ls, length ls ≤ len → Acc lengthOrder ls.
  unfold lengthOrder; induction len; crush.
Defined.

Theorem lengthOrder_wf : well_founded lengthOrder.
  red; intro; eapply lengthOrder_wf'; eauto.
Defined.
```

Notice that we end these proofs with Defined, not Qed. Recall that Defined marks the theorems as *transparent*, so that the details of their proofs may be used during program execution. Why could such details possibly matter for computation? It turns out that Fix satisfies the primitive recursion restriction by declaring itself as *recursive in the structure of* ***Acc*** *proofs*. This is possible because **Acc** proofs follow a predictable inductive structure. We must do work, as in the last theorem's proof, to establish that all elements of a type belong to **Acc**, but the automatic unwinding of those proofs during recursion is straightforward. If we ended the proof with Qed, the proof details would be hidden from computation, in which case the unwinding process would get stuck.

To justify our two recursive mergeSort calls, we will also need to prove that split respects the lengthOrder relation. These proofs, too, must be kept transparent, to avoid stuckness of Fix evaluation. We use the syntax @*foo* to reference identifier *foo* with its implicit argument behavior turned off. (The proof details below use Ltac features not introduced yet, and they are safe to skip for now.)

```
Lemma split_wf : ∀ len ls, 2 ≤ length ls ≤ len
  → let (ls1, ls2) := split ls in
    lengthOrder ls1 ls ∧ lengthOrder ls2 ls.
  unfold lengthOrder; induction len; crush; do 2 (destruct ls; crush);
    destruct (le_lt_dec 2 (length ls));
      repeat (match goal with
                   | [ _ : length ?E < 2 ⊢ _ ] ⇒ destruct E
                   | [ _ : S (length ?E) < 2 ⊢ _ ] ⇒ destruct E
                   | [ IH : _ ⊢ context[split ?L] ] ⇒
                     specialize (IH L); destruct (split L); destruct IH
                 end; crush).
Defined.

Ltac split_wf := intros ls ?; intros; generalize (@split_wf (length ls) ls);
  destruct (split ls); destruct 1; crush.

Lemma split_wf1 : ∀ ls, 2 ≤ length ls
  → lengthOrder (fst (split ls)) ls.
  split_wf.
Defined.

Lemma split_wf2 : ∀ ls, 2 ≤ length ls
  → lengthOrder (snd (split ls)) ls.
```

```
    split_wf.
  Defined.

  Hint Resolve split_wf1 split_wf2.
```

To write the function definition itself, we use the `refine` tactic as a convenient way to write a program that needs to manipulate proofs, without writing out those proofs manually. We also use a replacement le_lt_dec for leb that has a more interesting dependent type. (Note that we would not be able to complete the definition without this change, since `refine` will generate subgoals for the `if` branches based only on the *type* of the test expression, not its *value*.)

```
  Definition mergeSort : list A → list A.
    refine (Fix lengthOrder_wf (fun _ ⇒ list A)
      (fun (ls : list A)
        (mergeSort : ∀ ls' : list A, lengthOrder ls' ls → list A) ⇒
        if le_lt_dec 2 (length ls)
          then let lss := split ls in
            merge (mergeSort (fst lss) _) (mergeSort (snd lss) _)
          else ls)); subst lss; eauto.
  Defined.
End mergeSort.
```

The important thing is that it is now easy to evaluate calls to mergeSort.

```
Eval compute in mergeSort leb (1 :: 2 :: 36 :: 8 :: 19 :: nil).
   = 1 :: 2 :: 8 :: 19 :: 36 :: nil
```

Since the subject of this chapter is merely how to define functions with unusual recursion structure, we will not prove any further correctness theorems about mergeSort. Instead, we stop at proving that mergeSort has the expected computational behavior, for all inputs, not merely the one we just tested.

```
Theorem mergeSort_eq : ∀ A (le : A → A → bool) ls,
  mergeSort le ls = if le_lt_dec 2 (length ls)
    then let lss := split ls in
      merge le (mergeSort le (fst lss)) (mergeSort le (snd lss))
    else ls.
  intros; apply (Fix_eq (@lengthOrder_wf A) (fun _ ⇒ list A)); intros.
```

The library theorem Fix_eq imposes one more strange subgoal upon us. We must prove that the function body is unable to distinguish between "self" arguments that map equal inputs to equal outputs. One might think this should be true of any Gallina code, but in fact this general *function extensionality* property is neither provable nor disprovable within Coq. The type of Fix_eq makes clear what we must show manually:

```
  Check Fix_eq.

Fix_eq
     : ∀ (A : Type) (R : A → A → Prop) (Rwf : well_founded R)
```

$(P : A \rightarrow \texttt{Type})$
$(F : \forall\ x : A, (\forall\ y : A, R\ y\ x \rightarrow P\ y) \rightarrow P\ x),$
$(\forall\ (x : A)\ (f\ g : \forall\ y : A, R\ y\ x \rightarrow P\ y),$
$(\forall\ (y : A)\ (p : R\ y\ x), f\ y\ p = g\ y\ p) \rightarrow F\ x\ f = F\ x\ g) \rightarrow$
$\forall\ x : A,$
$\texttt{Fix}\ \mathit{Rwf}\ P\ F\ x = F\ x\ (\texttt{fun}\ (y : A)\ (\_ : R\ y\ x) \Rightarrow \texttt{Fix}\ \mathit{Rwf}\ P\ F\ y)$

Most such obligations are dischargeable with straightforward proof automation, and this example is no exception.

```
  match goal with
    | [ ⊢ context[match ?E with left _ ⇒ _ | right _ ⇒ _ end] ] ⇒ destruct E
  end; simpl; f_equal; auto.
Qed.
```

As a final test of our definition's suitability, we can extract to OCaml.

```
Extraction mergeSort.
```

```
let rec mergeSort le x =
  match le_lt_dec (S (S O)) (length x) with
  | Left ->
    let lss = split x in
    merge le (mergeSort le (fst lss)) (mergeSort le (snd lss))
  | Right -> x
```

We see almost precisely the same definition we would have written manually in OCaml! It might be a good exercise for the reader to use the commands we saw in the previous chapter to clean up some remaining differences from idiomatic OCaml.

One more piece of the full picture is missing. To go on and prove correctness of mergeSort, we would need more than a way of unfolding its definition. We also need an appropriate induction principle matched to the well-founded relation. Such a principle is available in the standard library, though we will say no more about its details here.

```
Check well_founded_induction.
```

well_founded_induction
$: \forall\ (A : \texttt{Type})\ (R : A \rightarrow A \rightarrow \texttt{Prop}),$
$\text{well\_founded}\ R \rightarrow$
$\forall\ P : A \rightarrow \texttt{Set},$
$(\forall\ x : A, (\forall\ y : A, R\ y\ x \rightarrow P\ y) \rightarrow P\ x) \rightarrow$
$\forall\ a : A, P\ a$

Some more recent Coq features provide more convenient syntax for defining recursive functions. Interested readers can consult the Coq manual about the commands `Function` and *Program* `Fixpoint`.

7.2 A Non-Termination Monad Inspired by Domain Theory

The key insights of domain theory [44] inspire the next approach to modeling non-termination. Domain theory is based on *information orders* that relate values representing computation results, according to how much information these values convey. For instance, a simple domain might include values "the program does not terminate" and "the program terminates with the answer 5." The former is considered to be an *approximation* of the latter, while the latter is *not* an approximation of "the program terminates with the answer 6." The details of domain theory will not be important in what follows; we merely borrow the notion of an approximation ordering on computation results.

Consider this definition of a type of computations.

```
Section computation.
  Variable A : Type.
```

The type A describes the result a computation will yield, if it terminates.

We give a rich dependent type to computations themselves:

```
  Definition computation :=
    {f : nat → option A
      | ∀ (n : nat) (v : A),
        f n = Some v
        → ∀ (n' : nat), n' ≥ n
          → f n' = Some v}.
```

A computation is fundamentally a function f from an *approximation level* n to an optional result. Intuitively, higher n values enable termination in more cases than lower values. A call to f may return None to indicate that n was not high enough to run the computation to completion; higher n values may yield Some. Further, the proof obligation within the subset type asserts that f is *monotone* in an appropriate sense: when some n is sufficient to produce termination, so are all higher n values, and they all yield the same program result v.

It is easy to define a relation characterizing when a computation runs to a particular result at a particular approximation level.

```
  Definition runTo (m : computation) (n : nat) (v : A) :=
    proj1_sig m n = Some v.
```

On top of runTo, we also define run, which is the most abstract notion of when a computation runs to a value.

```
  Definition run (m : computation) (v : A) :=
    ∃ n, runTo m n v.
End computation.
```

The book source code contains at this point some tactics, lemma proofs, and hint commands, to be used in proving facts about computations. Since their details are orthogonal to the message of this chapter, I have omitted them in the rendered version.

Now, as a simple first example of a computation, we can define Bottom, which corresponds to an infinite loop. For any approximation level, it fails to terminate (returns None). Note the use of `abstract` to create a new opaque lemma for the proof found by the *run* tactic. In contrast to the previous section, opaque proofs are fine here, since the proof components of computations do not influence evaluation behavior. It is generally preferable to make proofs opaque when possible, as this enforces a kind of modularity in the code to follow, preventing it from depending on any details of the proof.

```
Section Bottom.
  Variable A : Type.

  Definition Bottom : computation A.
    exists (fun _ : nat ⇒ @None A); abstract run.
  Defined.

  Theorem run_Bottom : ∀ v, ¬run Bottom v.
    run.
  Qed.
End Bottom.
```

A slightly more complicated example is Return, which gives the same terminating answer at every approximation level.

```
Section Return.
  Variable A : Type.
  Variable v : A.

  Definition Return : computation A.
    intros; exists (fun _ : nat ⇒ Some v); abstract run.
  Defined.

  Theorem run_Return : run Return v.
    run.
  Qed.
End Return.
```

The name Return was meant to be suggestive of the standard operations of monads [41]. The other standard operation is `Bind`, which lets us run one computation and, if it terminates, pass its result off to another computation. We implement bind using the notation `let` (x, y) := *e1* `in` *e2*, for pulling apart the value *e1* which may be thought of as a pair. The second component of a computation is a proof, which we do not need to mention directly in the definition of `Bind`.

```
Section Bind.
  Variables A B : Type.
  Variable m1 : computation A.
  Variable m2 : A → computation B.

  Definition Bind : computation B.
```

```
    exists (fun n ⇒
      let (f1, _) := m1 in
      match f1 n with
        | None ⇒ None
        | Some v ⇒
          let (f2, _) := m2 v in
            f2 n
      end); abstract run.
  Defined.

  Theorem run_Bind : ∀ (v1 : A) (v2 : B),
    run m1 v1
    → run (m2 v1) v2
    → run Bind v2.
    run; match goal with
           | [ x : nat, y : nat ⊢ _ ] ⇒ exists (max x y)
         end; run.
  Qed.
End Bind.
```

A simple notation lets us write Bind calls the way they appear in Haskell.

```
Notation "x <- m1 ; m2" :=
  (Bind m1 (fun x ⇒ m2)) (right associativity, at level 70).
```

We can verify that we have indeed defined a monad, by proving the standard monad laws. Part of the exercise is choosing an appropriate notion of equality between computations. We use "equality at all approximation levels."

```
Definition meq A (m1 m2 : computation A) := ∀ n, proj1_sig m1 n = proj1_sig m2 n.

Theorem left_identity : ∀ A B (a : A) (f : A → computation B),
  meq (Bind (Return a) f) (f a).
  run.
Qed.

Theorem right_identity : ∀ A (m : computation A),
  meq (Bind m (@Return _)) m.
  run.
Qed.

Theorem associativity : ∀ A B C (m : computation A)
  (f : A → computation B) (g : B → computation C),
  meq (Bind (Bind m f) g) (Bind m (fun x ⇒ Bind (f x) g)).
  run.
Qed.
```

Now we come to the piece most directly inspired by domain theory. We want to support general recursive function definitions, but domain theory tells us that not all definitions are reasonable; some fail to be *continuous* and thus represent unrealizable computations. To formalize an analogous notion of continuity for our non-termination monad, we write down the approximation relation on computation results that we have had in mind all along.

```
Section lattice.
  Variable A : Type.

  Definition leq (x y : option A) :=
    ∀ v, x = Some v → y = Some v.
End lattice.
```

We now have the tools we need to define a new `Fix` combinator that, unlike the one we saw in the prior section, does not require a termination proof, and in fact admits recursive definition of functions that fail to terminate on some or all inputs.

```
Section Fix.
```

First, we have the function domain and range types.

```
  Variables A B : Type.
```

Next comes the function body, which is written as though it can be parameterized over itself, for recursive calls.

```
  Variable f : (A → computation B) → (A → computation B).
```

Finally, we impose an obligation to prove that the body f is continuous. That is, when f terminates according to one recursive version of itself, it also terminates with the same result at the same approximation level when passed a recursive version that refines the original, according to leq.

```
  Hypothesis f_continuous : ∀ n v v1 x,
    runTo (f v1 x) n v
    → ∀ (v2 : A → computation B),
      (∀ x, leq (proj1_sig (v1 x) n) (proj1_sig (v2 x) n))
      → runTo (f v2 x) n v.
```

The computational part of the `Fix` combinator is easy to define. At approximation level 0, we diverge; at higher levels, we run the body with a functional argument drawn from the next lower level.

```
  Fixpoint Fix' (n : nat) (x : A) : computation B :=
    match n with
      | O ⇒ Bottom _
      | S n' ⇒ f (Fix' n') x
    end.
```

Now it is straightforward to package Fix' as a computation combinator `Fix`.

```
  Hint Extern 1 (_ ≥ _) ⇒ omega.
  Hint Unfold leq.

  Lemma Fix'_ok : ∀ steps n x v, proj1_sig (Fix' n x) steps = Some v
```

```
    → ∀ n', n' ≥ n
      → proj1_sig (Fix' n' x) steps = Some v.
    unfold runTo in *; induction n; crush;
      match goal with
        | [ H : _ ≥ _ ⊢ _ ] ⇒ inversion H; crush; eauto
      end.
  Qed.

  Hint Resolve Fix'_ok.

  Hint Extern 1 (proj1_sig _ _ = _) ⇒ simpl;
    match goal with
      | [ ⊢ proj1_sig ?E _ = _ ] ⇒ eapply (proj2_sig E)
    end.

  Definition Fix : A → computation B.
    intro x; exists (fun n ⇒ proj1_sig (Fix' n x) n); abstract run.
  Defined.
```

Finally, we can prove that Fix obeys the expected computation rule.

```
  Theorem run_Fix : ∀ x v,
    run (f Fix x) v
    → run (Fix x) v.
    run; match goal with
           | [ n : nat ⊢ _ ] ⇒ exists (S n); eauto
         end.
  Qed.
End Fix.
```

After all that work, it is now fairly painless to define a version of mergeSort that requires no proof of termination. We appeal to a program-specific tactic whose definition is hidden here but present in the book source.

```
Definition mergeSort' : ∀ A, (A → A → bool) → list A → computation (list A).
  refine (fun A le ⇒ Fix
    (fun (mergeSort : list A → computation (list A))
      (ls : list A) ⇒
      if le_lt_dec 2 (length ls)
        then let lss := split ls in
          ls1 ← mergeSort (fst lss);
          ls2 ← mergeSort (snd lss);
          Return (merge le ls1 ls2)
        else Return ls) _); abstract mergeSort'.
Defined.
```

Furthermore, "running" mergeSort' on concrete inputs is as easy as choosing a sufficiently high approximation level and letting Coq's computation rules do the rest. Contrast this with the proof work that goes into deriving an evaluation fact for a deeply embedded language, with one explicit proof rule application per execution step.

```
Lemma test_mergeSort' : run (mergeSort' leb (1 :: 2 :: 36 :: 8 :: 19 :: nil))
  (1 :: 2 :: 8 :: 19 :: 36 :: nil).
  exists 4; reflexivity.
Qed.
```

There is another benefit of our new Fix compared with the one we used in the previous section: we can now write recursive functions that sometimes fail to terminate, without losing easy reasoning principles for the terminating cases. Consider this simple example, which appeals to another tactic whose definition we elide here.

```
Definition looper : bool → computation unit.
  refine (Fix (fun looper (b : bool) ⇒
    if b then Return tt else looper b) _); abstract looper.
Defined.

Lemma test_looper : run (looper true) tt.
  exists 1; reflexivity.
Qed.
```

As before, proving outputs for specific inputs is as easy as demonstrating a high enough approximation level.

There are other theorems that are important to prove about combinators like Return, Bind, and Fix. In general, for a computation c, we sometimes have a hypothesis proving run c v for some v, and we want to perform inversion to deduce what v must be. Each combinator should ideally have a theorem of that kind, for c built directly from that combinator. We have omitted such theorems here, but they are not hard to prove. In general, the domain theory-inspired approach avoids the type-theoretic "gotchas" that tend to show up in approaches that try to mix normal Coq computation with explicit syntax types. The next section of this chapter demonstrates two alternate approaches of that sort. In the final section of the chapter, we review the pros and cons of the different choices, coming to the conclusion that none of them is obviously better than any one of the others for all situations.

7.3 Co-Inductive Non-Termination Monads

There are two key downsides to both of the previous approaches: both require unusual syntax based on explicit calls to fixpoint combinators, and both generate immediate proof obligations about the bodies of recursive definitions. In Chapter 5, we have already seen how co-inductive types support recursive definitions that exhibit certain well-behaved varieties of non-termination. It turns out that we can leverage that co-induction support for encoding of general recursive definitions, by adding layers of co-inductive syntax. In effect, we mix elements of shallow and deep embeddings.

Our first example of this kind, proposed by Capretta [4], defines a silly-looking type of thunks; that is, computations that may be forced to yield results, if they terminate.

```
CoInductive thunk (A : Type) : Type :=
| Answer : A → thunk A
| Think : thunk A → thunk A.
```

A computation is either an immediate Answer or another computation wrapped inside Think. Since **thunk** is co-inductive, every **thunk** type is inhabited by an infinite nesting of Thinks, standing for non-termination. Terminating results are Answer wrapped inside some finite number of Thinks.

Why bother to write such a strange definition? The definition of **thunk** is motivated by the ability it gives us to define a "bind" operation, similar to the one we defined in the previous section.

```
CoFixpoint TBind A B (m1 : thunk A) (m2 : A → thunk B) : thunk B :=
  match m1 with
    | Answer x ⇒ m2 x
    | Think m1' ⇒ Think (TBind m1' m2)
  end.
```

Note that the definition would violate the co-recursion guardedness restriction if we left out the seemingly superfluous Think on the righthand side of the second `match` branch.

We can prove that Answer and TBind form a monad for **thunk**. The proof is omitted here but present in the book source. As usual for this sort of proof, a key element is choosing an appropriate notion of equality for **thunk**s.

In the proofs to follow, we will need a function similar to one we saw in Chapter 5, to pull apart and reassemble a **thunk** in a way that provokes reduction of co-recursive calls.

```
Definition frob A (m : thunk A) : thunk A :=
  match m with
    | Answer x ⇒ Answer x
    | Think m' ⇒ Think m'
  end.

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

As a simple example, here is how we might define a tail-recursive factorial function.

```
CoFixpoint fact (n acc : nat) : thunk nat :=
  match n with
    | O ⇒ Answer acc
    | S n' ⇒ Think (fact n' (S n' × acc))
  end.
```

To test our definition, we need an evaluation relation that characterizes results of evaluating **thunk**s.

```
Inductive eval A : thunk A → A → Prop :=
| EvalAnswer : ∀ x, eval (Answer x) x
| EvalThink : ∀ m x, eval m x → eval (Think m) x.

Hint Rewrite frob_eq.

Lemma eval_frob : ∀ A (c : thunk A) x,
  eval (frob c) x
  → eval c x.
  crush.
Qed.

Theorem eval_fact : eval (fact 5 1) 120.
  repeat (apply eval_frob; simpl; constructor).
Qed.
```

We need to apply constructors of `eval` explicitly, but the process is easy to automate completely for concrete input programs.

Now consider another very similar definition, this time of a Fibonacci number function.

```
Notation "x <- m1 ; m2" :=
  (TBind m1 (fun x ⇒ m2)) (right associativity, at level 70).

CoFixpoint fib (n : nat) : thunk nat :=
  match n with
    | 0 ⇒ Answer 1
    | 1 ⇒ Answer 1
    | _ ⇒ n1 ← fib (pred n);
      n2 ← fib (pred (pred n));
      Answer (n1 + n2)
  end.
```

Coq complains that the guardedness condition is violated. The two recursive calls are immediate arguments to TBind, but TBind is not a constructor of **thunk**. Rather, it is a defined function. This example shows a very serious limitation of **thunk** for traditional functional programming: it is not, in general, possible to make recursive calls and then make further recursive calls, depending on the first call's result. The fact example succeeded because it was already tail recursive, meaning no further computation is needed after a recursive call.

I know no easy fix for this problem of **thunk**, but we can define an alternate co-inductive monad that avoids the problem, based on a proposal by Megacz [23]. We ran into trouble because TBind was not a constructor of **thunk**, so let us define a new type family where "bind" is a constructor.

```
CoInductive comp (A : Type) : Type :=
| Ret : A → comp A
| Bnd : ∀ B, comp B → (B → comp A) → comp A.
```

This example shows off Coq's support for *recursively non-uniform parameters*, as in the case of the parameter A declared above, where each constructor's type ends in **comp** A, but there is a recursive use of **comp** with a different parameter B. Beside that technical wrinkle, we see the simplest possible definition of a monad, via a type whose two constructors are precisely the monad operators.

It is easy to define the semantics of terminating **comp** computations.

```
Inductive exec A : comp A → A → Prop :=
| ExecRet : ∀ x, exec (Ret x) x
| ExecBnd : ∀ B (c : comp B) (f : B → comp A) x1 x2, exec (A := B) c x1
  → exec (f x1) x2
  → exec (Bnd c f) x2.
```

We can also prove that Ret and Bnd form a monad according to a notion of **comp** equality based on **exec**, but we omit details here; they are in the book source at this point.

Not only can we define the Fibonacci function with the new monad, but even our running example of merge sort becomes definable. By shadowing our previous notation for "bind," we can write almost exactly the same code as in our previous mergeSort' definition, but with less syntactic clutter.

```
Notation "x <- m1 ; m2" := (Bnd m1 (fun x ⇒ m2)).
CoFixpoint mergeSort'' A (le : A → A → bool) (ls : list A) : comp (list A) :=
  if le_lt_dec 2 (length ls)
    then let lss := split ls in
      ls1 ← mergeSort'' le (fst lss);
      ls2 ← mergeSort'' le (snd lss);
      Ret (merge le ls1 ls2)
    else Ret ls.
```

To execute this function, we go through the usual exercise of writing a function to catalyze evaluation of co-recursive calls.

```
Definition frob' A (c : comp A) :=
  match c with
    | Ret x ⇒ Ret x
    | Bnd _ c' f ⇒ Bnd c' f
  end.
Lemma exec_frob : ∀ A (c : comp A) x,
  exec (frob' c) x
  → exec c x.
  destruct c; crush.
Qed.
```

Now the same sort of proof script that we applied for testing **thunk**s will get the job done.

```
Lemma test_mergeSort'' : exec (mergeSort'' leb (1 :: 2 :: 36 :: 8 :: 19 :: nil))
```

```
  (1 :: 2 :: 8 :: 19 :: 36 :: nil).
  repeat (apply exec_frob; simpl; econstructor).
Qed.
```

Have we finally reached the ideal solution for encoding general recursive definitions, with minimal hassle in syntax and proof obligations? Unfortunately, we have not, as **comp** has a serious expressivity weakness. Consider the following definition of a curried addition function:

```
Definition curriedAdd (n : nat) := Ret (fun m : nat ⇒ Ret (n + m)).
```

This definition works fine, but we run into trouble when we try to apply it in a trivial way.

```
Definition testCurriedAdd := Bnd (curriedAdd 2) (fun f ⇒ f 3).
```

```
Error: Universe inconsistency.
```

The problem has to do with rules for inductive definitions that we will study in more detail in Chapter 12. Briefly, recall that the type of the constructor Bnd quantifies over a type B. To make testCurriedAdd work, we would need to instantiate B as **nat** → **comp nat**. However, Coq enforces a *predicativity restriction* that (roughly) no quantifier in an inductive or co-inductive type's definition may ever be instantiated with a term that contains the type being defined. Chapter 12 presents the exact mechanism by which this restriction is enforced, but for now our conclusion is that **comp** is fatally flawed as a way of encoding interesting higher-order functional programs that use general recursion.

7.4 Comparing the Alternatives

We have seen four different approaches to encoding general recursive definitions in Coq. Among them there is no clear champion that dominates the others in every important way. Instead, we close the chapter by comparing the techniques along a number of dimensions. Every technique allows recursive definitions with termination arguments that go beyond Coq's built-in termination checking, so we must turn to subtler points to highlight differences.

One useful property is automatic integration with normal Coq programming. That is, we would like the type of a function to be the same, whether or not that function is defined using an interesting recursion pattern. Only the first of the four techniques, well-founded recursion, meets this criterion. It is also the only one of the four to meet the related criterion that evaluation of function calls can take place entirely inside Coq's built-in computation machinery. The monad inspired by domain theory occupies some middle ground in this dimension, since generally standard computation is enough to evaluate a term once a high enough approximation level is provided.

Another useful property is that a function and its termination argument may be developed separately. We may even want to define functions that fail to terminate on some or all inputs. The well-founded recursion technique does not have this property, but the other three do.

One minor plus is the ability to write recursive definitions in natural syntax, rather than with calls to higher-order combinators. This downside of the first two techniques is actually rather easy to get around using Coq's notation mechanism, though we leave the details as an exercise for the reader. (For this and other details of notations, see Chapter 12 of the Coq 8.4 manual.)

The first two techniques impose proof obligations that are more basic than termination arguments, where well-founded recursion requires a proof of extensionality and domain-theoretic recursion requires a proof of continuity. A function may not be defined, and thus may not be computed with, until these obligations are proved. The co-inductive techniques avoid this problem, as recursive definitions may be made without any proof obligations.

We can also consider support for common idioms in functional programming. For instance, the **thunk** monad effectively only supports recursion that is tail recursion, while the others allow arbitrary recursion schemes.

On the other hand, the **comp** monad does not support the effective mixing of higher-order functions and general recursion, while all the other techniques do. For instance, we can finish the failed curriedAdd example in the domain-theoretic monad.

```
Definition curriedAdd' (n : nat) := Return (fun m : nat ⇒ Return (n + m)).

Definition testCurriedAdd := Bind (curriedAdd' 2) (fun f ⇒ f 3).
```

The same techniques also apply to more interesting higher-order functions like list map, and, as in all four techniques, we can mix primitive and general recursion, preferring the former when possible to avoid proof obligations.

```
Fixpoint map A B (f : A → computation B) (ls : list A) : computation (list B) :=
  match ls with
    | nil ⇒ Return nil
    | x :: ls' ⇒ Bind (f x) (fun x' ⇒
      Bind (map f ls') (fun ls'' ⇒
        Return (x' :: ls'')))
  end.

Theorem test_map : run (map (fun x ⇒ Return (S x)) (1 :: 2 :: 3 :: nil))
  (2 :: 3 :: 4 :: nil).
  exists 1; reflexivity.
Qed.
```

One further disadvantage of **comp** is that we cannot prove an inversion lemma for executions of `Bind` without appealing to an *axiom*, a logical complication that we discuss at more length in Chapter 12. The other three techniques allow proof of all the important theorems within the normal logic of Coq.

Perhaps one theme of our comparison is that one must trade off between, on one hand, functional programming expressiveness and compatibility with normal Coq types and computation; and, on the other hand, the level of proof obligations one is willing to handle at function definition time.

Chapter 8

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 outside `Prop`, a possibility that sets Coq apart from all of the competition not based on type theory.

8.1 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 tail. 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.
```

Past Coq versions signalled an error for this definition. The code is still invalid within Coq's core language, but current Coq versions automatically add annotations to the original program, producing a valid core program. These are the annotations on `match` discriminees that we began to study in the previous chapter. We can rewrite `app` to give the annotations explicitly.

```
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. That is, we could consider the length indices to lists to live in a separate, compile-time-only universe from the lists themselves. Compile-time data may be *erased* such that we can still execute a program. As an example where erasure would not work, consider an injection function from regular lists to length-indexed lists. Here the run-time computation actually depends on details of the compile-time argument, if we decide that the list to inject can be considered compile-time. More commonly, we think of lists as run-time data. Neither case will work with naïve erasure. (It is not too important to grasp the details of this run-time/compile-time distinction, since Coq's expressive power comes from avoiding such restrictions.)

```
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. We write ??? as a placeholder for a term that we do not know how to write, not for any real Coq notation like those introduced two chapters ago.

```
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. In fact, recent versions of Coq *do* allow this, by implicit translation to a match that considers all constructors; the error message above was generated by an older Coq version. It is educational to discover for ourselves the encoding that the most recent Coq versions use. 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 [14], 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.

Check hd'.
```

```
hd'
     : ∀ n : nat, ilist n → match n with
                                        | 0 ⇒ unit
                                        | S _ ⇒ A
                                        end
```

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

```
End ilist.
```

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.

8.2 The One Rule of Dependent Pattern Matching in Coq

The rest of this chapter will demonstrate a few other elegant applications of dependent types in Coq. Readers encountering such ideas for the first time often feel overwhelmed, concluding that there is some magic at work whereby Coq sometimes solves the halting problem for the programmer and sometimes does not, applying automated program understanding in a way far beyond what is found in conventional languages. The point of this section is to cut off that sort of thinking right now! Dependent type-checking in Coq follows just a few algorithmic rules. Chapters 10 and 12 introduce many of those rules more formally, and the main additional rule is centered on *dependent pattern matching* of the kind we met in the previous section.

A dependent pattern match is a `match` expression where the type of the overall `match` is a function of the value and/or the type of the *discriminee*, the value being matched on. In other words, the `match` type *depends* on the discriminee.

When exactly will Coq accept a dependent pattern match as well-typed? Some other dependently typed languages employ fancy decision procedures to determine when programs satisfy their very expressive types. The situation in Coq is just the opposite. Only very straightforward symbolic rules are applied. Such a design choice has its drawbacks, as it forces programmers to do more work to convince the type checker of program validity. However, the great advantage of a simple type checking algorithm is that its action on *invalid* programs is easier to understand!

We come now to the one rule of dependent pattern matching in Coq. A general dependent pattern match assumes this form (with unnecessary parentheses included to make the syntax easier to parse):

```
match E as y in (T x1 ... xn) return U with
  | C z1 ... zm ⇒ B
  | ...
end
```

The discriminee is a term *E*, a value in some inductive type family *T*, which takes *n* arguments. An `as` clause binds the name *y* to refer to the discriminee *E*. An `in` clause binds an explicit name *xi* for the *i*th argument passed to *T* in the type of *E*.

We bind these new variables *y* and *xi* so that they may be referred to in *U*, a type given in the `return` clause. The overall type of the `match` will be *U*, with *E* substituted for *y*, and with each *xi* substituted by the actual argument appearing in that position within *E*'s type.

In general, each case of a `match` may have a pattern built up in several layers from the constructors of various inductive type families. To keep this exposition simple, we will focus on patterns that are just single applications of inductive type constructors to lists of variables. Coq actually compiles the more general kind of pattern matching into this more restricted kind automatically, so understanding the typing of `match` requires understanding the typing of `match`es lowered to match one constructor at a time.

The last piece of the typing rule tells how to type-check a `match` case. A generic constructor application *C z1 ... zm* has some type *T x1' ... xn'*, an application of the type family used in *E*'s type, probably with occurrences of the *zi* variables. From here, a simple recipe determines what type we will require for the case body *B*. The type of *B* should be *U* with the following two substitutions applied: we replace *y* (the `as` clause variable) with *C z1 ... zm*, and we replace each *xi* (the `in` clause variables) with *xi'*. In other words, we specialize the result type based on what we learn based on which pattern has matched the discriminee.

This is an exhaustive description of the ways to specify how to take advantage of which pattern has matched! No other mechanisms come into play. For instance, there is no way to specify that the types of certain free variables should be refined based on which pattern has matched. In the rest of the book, we will learn design patterns for achieving similar effects, where each technique leads to an encoding only in terms of `in`, `as`, and `return` clauses.

A few details have been omitted above. In Chapter 3, we learned that inductive type families may have both *parameters* and regular arguments. Within an `in` clause, a parameter position must have the wildcard _ written, instead of a variable. (In general, Coq uses wildcard _'s either to indicate pattern variables that will not be mentioned again or to indicate positions where we would like type inference to infer the appropriate terms.) Furthermore, recent Coq versions are adding more and more heuristics to infer dependent `match` annotations in certain conditions. The general annotation inference problem is undecidable, so there will always be serious limitations on how much work these heuristics can do. When in doubt about why a particular dependent `match` is failing to type-check, add an explicit `return` annotation! At that point, the mechanical rule sketched in this section will provide a complete account of "what the type checker is thinking." Be sure to avoid the common pitfall of writing a `return` annotation that does not mention any variables bound by `in` or `as`; such a `match` will never refine typing requirements based on which pattern has matched. (One simple exception to this rule is that, when the discriminee is a variable, that same variable may be treated as if it were repeated as an `as` clause.)

8.3 A Tagless Interpreter

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

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

Inductive exp : type → Set :=
```

```
| NConst : nat → exp Nat
| Plus : exp Nat → exp Nat → exp Nat
| Eq : exp Nat → exp Nat → exp Bool

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

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

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

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

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Nat ⇒ nat
    | Bool ⇒ bool
    | Prod t1 t2 ⇒ typeDenote t1 × typeDenote t2
  end%type.
```

The typeDenote function 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. The token *type* is one example of an identifier bound to a *notation scope delimiter*. In this book, we will not go into more detail on notation scopes, but the Coq manual can be consulted for more information.

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) := option (match t with
                                               | Prod t1 t2 ⇒ 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 the harmless **option unit**, since we do not care about extracting components of non-pairs. Now pairOut is easy to write.

```
Definition pairOut t (e : exp t) :=
  match e in (exp t) return (pairOutType t) with
    | Pair _ _ e1 e2 ⇒ Some (e1 , e2)
    | _ ⇒ None
  end.
```

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 with explicit `return` clauses.

```
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 exp Nat 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 exp Bool 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 exp Bool 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 to do so 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's CpdtTactics module. General elimination/inversion of dependently typed hypotheses is undecidable, as witnessed by a simple reduction from the known-undecidable problem of higher-order unification, which has come up a few times already. The tactic *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 `dependent destruction`'s implementation, but for now, we treat it as a useful black box. (In Chapter 12, we will also see how `dependent destruction` forces us to make a larger philosophical commitment about our logic than we might like, and we will see some workarounds.)

```
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 (which again is safe to skip and uses Ltac features not introduced yet).

```
Restart.

induction e; crush;
  repeat (match goal with
           | [ ⊢ context[match cfold ?E with NConst _ ⇒ _ | _ ⇒ _ 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.
```

With this example, we get a first taste of how to build automated proofs that adapt automatically to changes in function definitions.

8.4 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 guarantee 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 whose root has color c and that has black depth d. The latter property means that there are exactly 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.

```
Check min_dec.

min_dec
     : ∀ n m : nat, {min n m = n} + {min n m = m}

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 $P \rightarrow Q$. This transformation is useful 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**.

A quick word of encouragement: 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" [27] and transcribed the code to use dependent types. Luckily, the details are not so important here; types alone will tell us that insertion preserves balancedness, and we will prove that insertion produces trees containing the right keys.

```
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' _ c0 _ t1 y t2 ⇒
      match t1 in rbtree c n return rbtree c0 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 Black 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.

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' _ c0 _ t1 z t2 ⇒
      match t1 in rbtree c n return rbtree c0 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 Black n → rbtree c2 n
            → { c : color & rbtree c (S n) } with
            | RedNode _ c z' d ⇒ fun b a ⇒
              {<RedNode (BlackNode a data b) z (BlackNode c z' d)>}
            | b ⇒ fun a t ⇒ {<BlackNode t data (RedNode a z b)>}
          end t1'
      end t2
  end.
```

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

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

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

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

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

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

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

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

Finally, we are in the home stretch of our effort to define insert. We just need a few more definitions of non-recursive functions. First, we need to give the final characterization of insert's return type. Inserting into a red-rooted tree gives a black-rooted tree where black depth has increased, and inserting into a black-rooted tree gives a tree where black depth has stayed the same and where the root is an arbitrary color.

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

A simple clean-up procedure translates insResults into insertResults.

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

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

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

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

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

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

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

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

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

```
  Ltac present_balance :=
    crush;
    repeat (match goal with
              | [ _ : context[match ?T with Leaf ⇒ _ | _ ⇒ _ end] ⊢ _ ] ⇒
                dep_destruct T
              | [ ⊢ context[match ?T with Leaf ⇒ _ | _ ⇒ _ end] ] ⇒ dep_destruct T
            end; crush).
```

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

```
  Lemma present_balance1 : ∀ n (a : rtree n) (y : nat) c2 (b : rbtree c2 n),
    present z (projT2 (balance1 a y b))
```

```
  ↔ rpresent z a ∨ z = y ∨ present z b.
  destruct a; present_balance.
Qed.

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

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

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

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

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

We can generate executable OCaml code with the command `Recursive Extraction insert`, which also automatically outputs the OCaml versions of all of insert's dependencies. In our previous extractions, we wound up with clean OCaml code. Here, we find uses of `Obj.magic`, OCaml's unsafe cast operator for tweaking the apparent type of an expression in an arbitrary way. Casts appear for this example because the return type of insert depends on the *value* of the function's argument, a pattern that OCaml cannot handle. Since Coq's type system is much more expressive than OCaml's, such casts are unavoidable in general.

Since the OCaml type-checker is no longer checking full safety of programs, we must rely on Coq's extractor to use casts only in provably safe ways.

8.5 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. That is, a string *s* matches regular expression **star** *e* if and only if *s* can be decomposed into a sequence of substrings that all match *e*. We use Coq's string support, which comes through a combination of the String 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, such that the index of a **regexp** tells us which language (string predicate) it recognizes. Here is a reasonable-looking definition that is restricted to constant characters and concatenation. We use the constructor String, which is the analogue of list cons for the type **string**, where "" is like list nil.

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

The function 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. The execution of 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 ⊢ _ ] ⇒ destruct (eq_nat_dec (length S') (S N))
    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' _
```

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

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

Implicit Arguments split [P1 P2].
```

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

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

Some new lemmas and hints about the **star** type family are useful. 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.
```

Variable 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, defined by straightforward reduction from dec_star'.

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

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

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

It is interesting to pause briefly to consider alternate implementations of matches. Dependent types give us much latitude in how specific correctness properties may be encoded with types. For instance, we could have made **regexp** a non-indexed inductive type, along the lines of what is possible in traditional ML and Haskell. We could then have implemented a recursive function to map **regexp**s to their intended meanings, much as we have done with types and programs in other examples. That style is compatible with the `refine`-based approach that we have used here, and it might be an interesting exercise to redo the code from this subsection in that alternate style or some further encoding of the reader's choice. The main advantage of indexed inductive types is that they generally lead to the smallest amount of code.

Many regular expression matching problems are easy to test. The reader may run each of the following queries to verify that it gives the correct answer. We use evaluation strategy `hnf` to reduce each term to *head-normal form*, where the datatype constructor used to build its value is known. (Further reduction would involve wasteful simplification of proof terms justifying the answers of our procedures.)

```
Example a_star := Star (Char "a"%char).
Eval hnf in matches a_star "".
Eval hnf in matches a_star "a".
Eval hnf in matches a_star "b".
Eval hnf in matches a_star "aa".
```

Evaluation inside Coq does not scale very well, so it is easy to build other tests that run for hours or more. Such cases are better suited to execution with the extracted OCaml code.

Chapter 9

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.

9.1 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 $\{m : \textbf{nat} \mid m < n\}$. The type family name stands for "finite."

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

An instance of **fin** is essentially a more richly typed copy of a prefix of the natural numbers. Every element is a First iterated through applying Next a number of times that indicates which number is being selected. For instance, the three values of type **fin** 3 are First 2, Next (First 1), and Next (Next (First 0)).

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 `Fixpoints` 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`, with the `return` clause chosen carefully so that it returns the proper type A in case the **fin** index is O, which we know is true here; and so that it returns an easy-to-inhabit type **unit** in the remaining, impossible cases, which nonetheless appear explicitly in the body of the `match`.

```
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'*. In fact, the error message Coq gives for this exact code can be confusing, thanks to an overenthusiastic type inference heuristic. We are told that the Nil case body has type `match` X `with` | O ⇒ A | S _ ⇒ **unit** `end` for a unification variable X, while it is expected to have type A. We can see that setting X to O resolves the conflict, but Coq is not yet smart enough to do this unification automatically. Repeating the function's type in a `return` annotation, used with an `in` annotation, leads us to a more informative error message, saying that *idx'* has type **fin** *n1* while it is expected to have type **fin** *n0*, where *n0* is bound by the Cons pattern and *n1* by the `Next` pattern. As the code is written above, nothing forces these two natural numbers to be equal, though we know intuitively that they must be.

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.

```
  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.
```

The only tricky bit is remembering to use our *dep_destruct* tactic in place of plain `destruct` when faced with a baffling tactic error message.

9.2 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 :=
  | HNil : hlist nil
  | HCons : ∀ (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 (in type family **member**) by the types of data that they point to.

```
  Variable elm : A.

  Inductive member : list A → Type :=
  | HFirst : ∀ ls, member (elm :: ls)
  | HNext : ∀ 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 **hlist**s. The same basic `match` tricks apply. In the HCons 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
```

```
      | HNil ⇒ fun mem ⇒
        match mem in member ls' return (match ls' with
                                          | nil ⇒ B elm
                                          | _ :: _ ⇒ unit
                                        end) with
          | HFirst _ ⇒ tt
          | HNext _ _ _ ⇒ tt
        end
      | HCons _ _ 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
          | HFirst _ ⇒ fun x _ ⇒ x
          | HNext _ _ mem' ⇒ fun _ get_mls' ⇒ get_mls' mem'
        end x (hget mls')
    end.
End hlist.

Implicit Arguments HNil [A B].
Implicit Arguments HCons [A B x ls].

Implicit Arguments HFirst [A elm ls].
Implicit Arguments HNext [A elm x ls].
```

By putting the parameters A and B in `Type`, we enable fancier kinds of polymorphism than in mainstream functional languages. 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 :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

    = 5
    : (fun T : Set ⇒ T) nat

Eval simpl in hget someValues (HNext HFirst).

    = 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 :=
```

```
HCons (1, 2) (HCons (true, false) HNil).
```

There are many other useful applications of heterogeneous lists, based on different choices of the first argument to **hlist**.

9.2.1 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 (but worry not if you are not familiar with such terminology from 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 index variable representation [10]. 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 an **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 HCons 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' (HCons x s)
end.
```

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

```
Eval simpl in expDenote Const HNil.

    = tt
     : typeDenote Unit

Eval simpl in expDenote (Abs (dom := Unit) (Var HFirst)) HNil.

     = fun x : unit ⇒ x
     : typeDenote (Arrow Unit Unit)

Eval simpl in expDenote (Abs (dom := Unit)
  (Abs (dom := Unit) (Var (HNext HFirst)))) HNil.

     = fun x _ : unit ⇒ x
     : typeDenote (Arrow Unit (Arrow Unit Unit))

Eval simpl in expDenote (Abs (dom := Unit) (Abs (dom := Unit) (Var HFirst))) HNil.

     = fun _ x0 : unit ⇒ x0
     : typeDenote (Arrow Unit (Arrow Unit Unit))

Eval simpl in expDenote (App (Abs (Var HFirst)) Const) HNil.

     = 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. Other, more common approaches to language formalization often state and prove explicit theorems about type safety of languages. In the above example, we got type safety, termination, and other meta-theorems for free, by reduction to CIC, which we know has those properties.

9.3 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. Here we will preface type names with the letter f to indicate that they are based on explicit recursive *function* 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 = 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). For instance, the three values of type ffin 3 are None, Some None, and Some (Some None).

```
  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 ffin 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 idea 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
                       | eq_refl ⇒ 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 := eq_refl : 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 eq_refl shows that y can only ever be instantiated to x. Thus, within a pattern-match with eq_refl, occurrences of y can be replaced with occurrences of x for typing purposes.

```
End fhlist.
```

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

How does one choose between the two data structure encoding strategies we have presented so far? Before answering that question in this chapter's final section, we introduce one further approach.

9.4 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 in nested inductive definitions.

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 domain 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 O f1 ≥ rifoldr plus O 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.

9.4.1 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 $n \to$ **exp'** Bool, and the bodies are represented with a function of type ffin $n \to$ **exp'** t, where t is the overall type. The final **exp'** t argument is the default case. For example, here is an expression that successively checks whether $2 + 2 = 5$ (returning 0 if so) or if $1 + 1 = 2$ (returning 1 if so), returning 2 otherwise.

```
Example ex1 := Cond 2
  (fun f ⇒ match f with
              | None ⇒ Eq (Plus (NConst 2) (NConst 2)) (NConst 5)
              | Some None ⇒ Eq (Plus (NConst 1) (NConst 1)) (NConst 2)
              | Some (Some v) ⇒ match v with end
            end)
  (fun f ⇒ match f with
              | None ⇒ NConst 0
              | Some None ⇒ NConst 1
              | Some (Some v) ⇒ match v with end
            end)
  (NConst 2).
```

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 exp' Nat 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 exp' Bool 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. The following lemma formalizes that property. The proof is a standard mostly automated one, with the only wrinkle being a guided instantiation of the quantifiers in the induction hypothesis.

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

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

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

We add our two lemmas as hints and perform standard automation with pattern-matching of subterms to destruct.

9.5 Choosing Between Representations

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

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

Recursive types usually involve much less initial effort, but they can be less convenient to use with proof automation. For instance, the `simpl` tactic (which is among the ingredients in *crush*) will sometimes be overzealous in simplifying uses of functions over recursive types. Consider a call get l f, where variable l has type filist A (S n). The type of l would be simplified to an explicit pair type. In a proof involving many recursive types, this kind of unhelpful "simplification" can lead to rapid bloat in the sizes of subgoals. Even worse, it can prevent syntactic pattern-matching, like in cases where filist is expected but a pair type is found in the "simplified" version. The same problem applies to applications of recursive functions to values in recursive types: the recursive function call may "simplify" when the top-level structure of the type index but not the recursive value is known, because such functions are generally defined by recursion on the index, not the value.

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.

Finally, Coq type inference can be more helpful in constructing values in inductive types. Application of a particular constructor of that type tells Coq what to expect from the arguments, while, for instance, forming a generic pair does not make clear an intention to interpret the value as belonging to a particular recursive type. This downside can be mitigated to an extent by writing "constructor" functions for a recursive type, mirroring the definition of the corresponding inductive type.

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.

Chapter 10

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

10.1 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 [10] that encodes terms canonically.

The delta rule is for unfolding global definitions. We can use it here to unfold the definition of pred'. We do this with the `cbv` tactic, which takes a list of reduction rules and makes as many call-by-value reduction steps as possible, using only those rules. There is an analogous tactic `lazy` for call-by-need reduction.

```
cbv delta.
```

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

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

```
cbv beta.
```

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

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

```
cbv iota.
```

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

Now we need another beta reduction.

```
cbv beta.
```

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

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

```
cbv zeta.
```

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

  reflexivity.
Qed.
```

The `beta` reduction rule applies to recursive functions as well, and its behavior may be surprising in some instances. For instance, we can run some simple tests using the reduction strategy compute, which applies all applicable rules of the definitional equality.

```
Definition id (n : nat) := n.

Eval compute in fun x ⇒ id x.

     = fun x : nat ⇒ x

Fixpoint id' (n : nat) := n.

Eval compute in fun x ⇒ id' x.

     = fun x : nat ⇒ (fix id' (n : nat) : nat := n) x
```

By running compute, we ask Coq to run reduction steps until no more apply, so why do we see an application of a known function, where clearly no beta reduction has been performed? The answer has to do with ensuring termination of all Gallina programs. One candidate rule would say that we apply recursive definitions wherever possible. However, this would clearly lead to nonterminating reduction sequences, since the function may appear fully applied within its own definition, and we would naïvely "simplify" such applications immediately. Instead, Coq only applies the beta rule for a recursive function when *the top-level structure of the recursive argument is known*. For id' above, we have only one argument n, so clearly it is the recursive argument, and the top-level structure of n is known when the function is applied to O or to some S e term. The variable x is neither, so reduction is blocked.

What are recursive arguments in general? Every recursive function is compiled by Coq to a fix expression, for anonymous definition of recursive functions. Further, every fix with multiple arguments has one designated as the recursive argument via a struct annotation. The recursive argument is the one that must decrease across recursive calls, to appease Coq's termination checker. Coq will generally infer which argument is recursive, though we may also specify it manually, if we want to tweak reduction behavior. For instance, consider this definition of a function to add two lists of **nat**s elementwise:

```
Fixpoint addLists (ls1 ls2 : list nat) : list nat :=
  match ls1, ls2 with
    | n1 :: ls1' , n2 :: ls2' ⇒ n1 + n2 :: addLists ls1' ls2'
    | _, _ ⇒ nil
  end.
```

By default, Coq chooses *ls1* as the recursive argument. We can see that *ls2* would have been another valid choice. The choice has a critical effect on reduction behavior, as these two examples illustrate:

```
Eval compute in fun ls ⇒ addLists nil ls.

     = fun _ : list nat ⇒ nil

Eval compute in fun ls ⇒ addLists ls nil.

     = fun ls : list nat ⇒
```

```
(fix addLists (ls1 ls2 : list nat) : list nat :=
   match ls1 with
   | nil ⇒ nil
   | n1 :: ls1' ⇒
       match ls2 with
       | nil ⇒ nil
       | n2 :: ls2' ⇒
           (fix plus (n m : nat) : nat :=
              match n with
              | 0 ⇒ m
              | S p ⇒ S (plus p m)
              end) n1 n2 :: addLists ls1' ls2'
       end
   end) ls nil
```

The outer application of the fix expression for addLists was only simplified in the first case, because in the second case the recursive argument is *ls*, whose top-level structure is not known.

The opposite behavior pertains to a version of addLists with *ls2* marked as recursive.

```
Fixpoint addLists' (ls1 ls2 : list nat) {struct ls2} : list nat :=
  match ls1, ls2 with
    | n1 :: ls1' , n2 :: ls2' ⇒ n1 + n2 :: addLists' ls1' ls2'
    | _, _ ⇒ nil
  end.

Eval compute in fun ls ⇒ addLists' ls nil.

     = fun ls : list nat ⇒ match ls with
                              | nil ⇒ nil
                              | _ :: _ ⇒ nil
                              end
```

We see that all use of recursive functions has been eliminated, though the term has not quite simplified to nil. We could get it to do so by switching the order of the match discriminees in the definition of addLists'.

Recall that co-recursive definitions have a dual rule: a co-recursive call only simplifies when it is the discriminee of a match. This condition is built into the beta rule for cofix, the anonymous form of CoFixpoint.

The standard **eq** relation is critically dependent on the definitional equality. The relation **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, I will introduce proof methods for goals that use proofs of the standard propositional equality "as data."

10.2 Heterogeneous Lists Revisited

One of our example dependent data structures from the last chapter (code repeated below) was the heterogeneous list and its associated "cursor" type. The recursive version poses some special challenges related to equality proofs, since it uses such proofs in its definition of fmember 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
                       | eq_refl ⇒ 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, which sets us on a collision course with the problems that are the subject of this chapter.

```
  Variable elm : A.

  Theorem fhget_fhmap : ∀ ls (mem : fmember elm ls) (hls : fhlist B ls),
    fhget (fhmap hls) mem = f (fhget hls mem).

    induction ls; crush.
```

In Coq 8.2, one subgoal remains at this point. Coq 8.3 has added some tactic improvements that enable *crush* to complete all of both inductive cases. To introduce the basics of reasoning about equality, it will be useful to review what was necessary in Coq 8.2.

Part of our single remaining subgoal is:

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

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

```
    destruct a0.
```

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

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

```
  assert (a0 = eq_refl _).
```

```
The term "eq_refl ?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. The `destruct` tactic 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 eq_refl ⇒ O end.
  simple destruct pf; reflexivity.
Qed.
```

The tactic `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
                                     | eq_refl ⇒ 0
                                     end) with
| eq_refl ⇒ eq_refl 0
end
     : ∀ (x : A) (pf : x = elm), 0 = match pf with
                                     | eq_refl ⇒ 0
```

```
end
```

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

```
Definition lemma1' (x : A) (pf : x = elm) :=
  match pf return (0 = match pf with
                           | eq_refl => 0
                         end) with
    | eq_refl => eq_refl 0
  end.
```

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

```
Lemma lemma2 : ∀ (x : A) (pf : x = x), O = match pf with eq_refl => 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
                             | eq_refl => 0
                           end) with
      | eq_refl => eq_refl 0
    end.
```

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

```
Lemma lemma3 : ∀ (x : A) (pf : x = x), pf = eq_refl 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' = eq_refl x') with
      | eq_refl => eq_refl _
    end.
```

```
The term "eq_refl x'" has type "x' = x'" while it is expected to have type
 "x = x'"
```

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

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

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

Check UIP_refl.
```

UIP_refl
  $: \forall\ (U : \texttt{Type})\ (x :\ U)\ (p :\ x = x),\ p = \mathsf{eq\_refl}\ x$

The theorem UIP_refl comes from the Eqdep module of the standard library. (Its name uses the acronym "UIP" for "unicity of identity proofs.") 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*, the term *eq_rect_eq* below.

```
Print eq_rect_eq.
```

*** [*eq_rect_eq* :
$\forall\ (U : \texttt{Type})\ (p :\ U)\ (Q : U \to \texttt{Type})\ (x :\ Q\ p)\ (h :\ p = p),$
$x = \mathsf{eq\_rect}\ p\ Q\ x\ p\ h$]

The axiom *eq_rect_eq* states a "fact" that seems like common sense, once the notation is deciphered. The term 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`. The statement of *eq_rect_eq* just says that `match`es on proofs of $p = p$, for any p, are superfluous and may be removed. We can see this intuition better in code by asking Coq to simplify the theorem statement with the `compute` reduction strategy.

```
Eval compute in (∀ (U : Type) (p : U) (Q : U → Type) (x : Q p) (h : p = p),
  x = eq_rect p Q x p h).

   = ∀ (U : Type) (p : U) (Q : U → Type) (x : Q p) (h : p = p),
     x = match h in (_ = y) return (Q y) with
         | eq_refl ⇒ x
         end
```

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 [39], in a study that also established unprovability of the axiom in CIC.

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

```
Check Streicher_K.
```

```
Streicher_K
     : ∀ (U : Type) (x : U) (P : x = x → Prop),
       P eq_refl → ∀ p : x = x, P p
```

This is the opaquely 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.
```

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. To simplify presentation, we will stick with the axiom version in the rest of this chapter.

10.3 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_assoc : ∀ 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_assoc : ∀ 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
      | eq_refl ⇒ 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
 | eq_refl ⇒ 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
| eq_refl ⇒
    (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
| eq_refl ⇒
    (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
| eq_refl ⇒
    fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
      (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3
```

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

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

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

```
∀ f : fhlist B ((ls1 ++ ls2) ++ ls3),
(a0,
match pf' in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒ f
end) =
match pf in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒ (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 such a phenomenon applies to the case here and to many other proofs that use dependent types heavily. Speaking informally, the reason why this kind of activity helps is that match annotations contain some positions where only variables are allowed. 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
| eq_refl ⇒ f
end) =
match pf0 in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒ (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 so before because other parts of the goal require the proofs to retain their original types. In particular, the call to fhapp that we generalized must have type (*ls1* ++ *ls2*) ++ *ls3*, for some values of the list variables. If we rewrite the type of the proof used to type-cast this value to something like *ls1* ++ *ls2* ++ *ls3* = *ls1* ++ *ls2* ++ *ls3*, then the lefthand side of the equality would no longer match the type of the term we are trying to cast.

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

```
rewrite app_assoc.
```

```
============================
∀ (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
| eq_refl ⇒ f
end) =
match pf0 in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒ (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].
```

This proof strategy was cumbersome and unorthodox, from the perspective of mainstream mathematics. The next section explores an alternative that leads to simpler developments in some cases.

10.4 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
```

The identifier JMeq stands for "John Major equality," a name coined by Conor McBride [22] as an inside joke about British politics. The definition JMeq starts out looking a lot like the definition of **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 == eq_refl x :=
  match pf return (pf == eq_refl _) with
    | eq_refl ⇒ JMeq_refl _
  end.
```

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

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

```
Lemma lemma4 : ∀ (A : Type) (x : A) (pf : x = x),
  O = match pf with eq_refl ⇒ 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. The *JMeq_eq* axiom has been proved on paper to be consistent, but asserting it may still be considered to complicate the logic we work in, so there is some motivation for avoiding it.

We can redo our fhapp associativity proof based around JMeq.

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

This time, the naïve theorem statement type-checks.

```
Theorem fhapp_assoc' : ∀ 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 (fhapp hls2 hls3)) == (a0, fhapp (fhapp b hls2) hls3)
```

It looks like one rewrite with the inductive hypothesis should be enough to make the goal trivial. Here is what happens when we try that in Coq 8.2:

```
  rewrite IHls1.
```

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

Coq 8.4 currently gives an error message about an uncaught exception. Perhaps that will be fixed soon. In any case, it is educational to consider a more explicit approach.

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_assoc.
```

```
============================
 ∀ 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'.
```

This example illustrates a general pattern: heterogeneous equality often simplifies theorem statements, but we still need to do some work to line up some dependent pattern matches that tactics will generate for us.

The proof we have found relies on the *JMeq_eq* axiom, which we can verify with a command that we will discuss more in two chapters.

```
Print Assumptions fhapp_assoc'.

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

It was the `rewrite` H tactic that implicitly appealed to the axiom. By restructuring the proof, we can avoid axiom dependence. A general lemma about pairs provides the key element. (Our use of `generalize` above can be thought of as reducing the proof to another, more complex and specialized lemma.)

```
Lemma pair_cong : ∀ A1 A2 B1 B2 (x1 : A1) (x2 : A2) (y1 : B1) (y2 : B2),
  x1 == x2
  → y1 == y2
  → (x1, y1) == (x2, y2).
  intros until y2; intros Hx Hy; rewrite Hx; rewrite Hy; reflexivity.
Qed.

Hint Resolve pair_cong.

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

  Theorem fhapp_assoc'' : ∀ 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.
  Qed.
End fhapp''.

Print Assumptions fhapp_assoc''.

Closed under the global context
```

One might wonder exactly which elements of a proof involving JMeq imply that *JMeq_eq* must be used. For instance, above we noticed that `rewrite` had brought *JMeq_eq* into the proof of fhapp_assoc', yet here we have also used `rewrite` with JMeq hypotheses while avoiding axioms! One illuminating exercise is comparing the types of the lemmas that `rewrite` uses under the hood to implement the rewrites. Here is the normal lemma for **eq** rewriting:

```
Check eq_ind_r.
```

```
eq_ind_r
     : ∀ (A : Type) (x : A) (P : A → Prop),
       P x → ∀ y : A, y = x → P y
```

The corresponding lemma used for JMeq in the proof of pair_cong is defined internally by rewrite as needed, but its type happens to be the following.

```
internal_JMeq_rew_r
     : ∀ (A : Type) (x : A) (B : Type) (b : B)
         (P : ∀ B0 : Type, B0 → Type), P B b → x == b → P A x
```

The key difference is that, where the **eq** lemma is parameterized on a predicate of type $A \to$ Prop, the JMeq lemma is parameterized on a predicate of type more like $\forall$ A : Type, $A \to$ Prop. To apply eq_ind_r with a proof of $x = y$, it is only necessary to rearrange the goal into an application of a fun abstraction to y. In contrast, to apply the alternative principle, it is necessary to rearrange the goal to an application of a fun abstraction to both y and *its type*. In other words, the predicate must be *polymorphic* in y's type; any type must make sense, from a type-checking standpoint. There may be cases where the former rearrangement is easy to do in a type-correct way, but the second rearrangement done naïvely leads to a type error.

When rewrite cannot figure out how to apply the alternative principle for $x == y$ where x and y have the same type, the tactic can instead use a different theorem, which is easy to prove as a composition of eq_ind_r and *JMeq_eq*.

```
Check JMeq_ind_r.
```

```
JMeq_ind_r
     : ∀ (A : Type) (x : A) (P : A → Prop),
       P x → ∀ y : A, y == x → P y
```

Ironically, where in the proof of fhapp_assoc' we used rewrite app_assoc to make it clear that a use of JMeq was actually homogeneously typed, we created a situation where rewrite applied the axiom-based JMeq_ind_r instead of the axiom-free principle!

For another simple example, consider this theorem that applies a heterogeneous equality to prove a congruence fact.

```
Theorem out_of_luck : ∀ n m : nat,
  n == m
  → S n == S m.
  intros n m H.
```

Applying JMeq_ind_r is easy, as the pattern tactic will transform the goal into an application of an appropriate fun to a term that we want to abstract. (In general, pattern abstracts over a term by introducing a new anonymous function taking that term as argument.)

```
  pattern n.
```

n : **nat**
m : **nat**
H : $n == m$
============================
(fun $n0$: **nat** ⇒ S $n0$ == S m) n

apply JMeq_ind_r with ($x := m$); auto.

However, we run into trouble trying to get the goal into a form compatible with the alternative principle.

Undo 2.

pattern **nat**, n.

```
Error: The abstracted term "fun (P : Set) (n0 : P) => S n0 == S m"
is not well typed.
Illegal application (Type Error):
The term "S" of type "nat -> nat"
cannot be applied to the term
 "n0" : "P"
This term has type "P" which should be coercible to
"nat".
```

In other words, the successor function S is insufficiently polymorphic. If we try to generalize over the type of n, we find that S is no longer legal to apply to n.

Abort.

Why did we not run into this problem in our proof of fhapp_assoc''? The reason is that the pair constructor is polymorphic in the types of the pair components, while functions like S are not polymorphic at all. Use of such non-polymorphic functions with JMeq tends to push toward use of axioms. The example with **nat** here is a bit unrealistic; more likely cases would involve functions that have *some* polymorphism, but not enough to allow abstractions of the sort we attempted above with pattern. For instance, we might have an equality between two lists, where the goal only type-checks when the terms involved really are lists, though everything is polymorphic in the types of list data elements. The Heq[1] library builds up a slightly different foundation to help avoid such problems.

10.5 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

[1] http://www.mpi-sws.org/~gil/Heq/

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 by showing how each of 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 = eq_refl 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 eq_refl ⇒ 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 (eq_refl 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
            | eq_refl ⇒ y
            end
  ============================
   x = y

  destruct H.

  x0 : A = A
```

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

```
rewrite H.
```

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

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

We see that, in a very formal sense, we are free to switch back and forth between the two styles of proofs about equality proofs. One style may be more convenient than the other for some proofs, but we can always interconvert between our results. The style that does not use heterogeneous equality may be preferable in cases where many results do not require the tricks of this 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.

10.6 Equality of Functions

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

```
Theorem two_funs : (fun n ⇒ n) = (fun n ⇒ n + 0).
```

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, and indeed the standard library already contains that axiom.

```
Require Import FunctionalExtensionality.
About functional_extensionality.
```

```
functional_extensionality :
∀ (A B : Type) (f g : A → B), (∀ x : A, 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 two_funs is trivial.

```
Theorem two_funs : (fun n ⇒ n) = (fun n ⇒ n + 0).
```

```
  apply functional_extensionality; crush.
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 functional_extensionality, but we can use change to fix that problem.

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

Unlike in the case of *eq_rect_eq*, we have no way of deriving this axiom of *functional extensionality* for types with decidable equality. To allow equality reasoning without axioms, it may be worth rewriting a development to replace functions with alternate representations, such as finite map types for which extensionality is derivable in CIC.

Chapter 11

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 [21] and Haskell type classes [40] are more flexible cases. These language features are often not as powerful as 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 [30], 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.

11.1 Reifying 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, which we will study in more detail in the next chapter. 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 reified 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. We use the `Record` command as a shorthand for defining an inductive type with a single constructor, plus projection functions for pulling out any of the named arguments to that constructor.

```
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 $A \times B$. We can generalize to any number of arguments via tupling.

With this definition, it is easy to define a datatype representation in terms of lists of constructors. The intended meaning is that the datatype came from an inductive definition including exactly the constructors in the list.

```
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.
```

The type **Empty_set** has no constructors, so its representation is the empty list. The type **unit** has one constructor with no arguments, so its one reified 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 → ilist T (recursive c) → 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 8.

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

Finally, the evidence for type T is a heterogeneous 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 a helpful notation for constructor denotations. The ASCII `~>` from the notation will be rendered later as ⇝.

```
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.
```

Recall that the hd and tl calls above operate on richly typed lists, where type indices tell us the lengths of lists, guaranteeing the safety of operations like hd. The type annotation attached to each definition provides enough information for Coq to infer list lengths at appropriate points.

11.2 Recursive Definitions

We built these encodings of datatypes to help us write datatype-generic recursive functions. To do so, we will want a reified 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) :=
  ∀ (R : Type), datatypeDenote R dt → (T → 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 heterogeneous 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 reified 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 ls
```

The function 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 that 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 **ilist**-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
```

As our examples show, even recursive datatypes are mapped to normal-looking size functions.

11.2.1 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 → string

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
```

Some of these simplified terms seem overly complex because we have turned off simplification of calls to append, which is what uses of the ++ operator desugar to. Selective ++ simplification would combine adjacent string literals, yielding more or less the code we would write manually to implement this printing scheme.

11.2.2 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
```

These map functions are just as easy to use as those we write by hand. Can you figure out the input-output pattern that map_nat S displays in these examples?

```
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
```

We get map_nat S $n = 2 \times n + 1$, because the mapping process adds an extra S at every level of the inductive tree that defines a natural, including at the last level, the O constructor.

11.3 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.
  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.

  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 IHr 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.
```

The proof involves complex subgoals, but, still, few steps are required, and then we may reuse our work across a variety of datatypes.

Part III

証明工学

Chapter 12

論理プログラミングによる証明探索

カリー・ハワード対応は、証明をすることが「まさしく」プログラミングであることを示しますが、このふたつの活動の実際はとても異なります。私たちは、一般に、プログラムの型のみならず属性について注意を払いますが、証明についてはそうではありません。定理のどんな証明でも同様です。結果として、自動化された証明探索は自動化されたプログラミングよりも概念的に簡単です。

Prolog [38] などの言語に組み入れられた論理プログラミング logic programming [20] のパラダイムは高階論理の証明探索にとてもよく適合します。この章では、過去の論理プログラミングの経験がなくてもよいように、詳細を紹介します。

12.1 論理プログラミング入門

標準ライブラリから加算の定義を思いだしましょう。

```
Print plus.
```

```
plus =
fix plus (n m : nat) : nat := match n with
                              | 0 ⇒ m
                              | S p ⇒ S (plus p m)
                              end
```

これは関数プログラミングのスタイルにおける再帰的な定義です。また、前の章で定義した帰納的関係に対応する論理プログラミングのスタイルにも従います。

```
Inductive plusR : nat → nat → nat → Prop :=
| PlusO : ∀ m, plusR O m m
| PlusS : ∀ n m r, plusR n m r
  → plusR (S n) m (S r).
```

直観的にいうと、事実 **plusR** $n\ m\ r$ は、plus $n\ m = r$ であるときだけ成り立ちます。この対応を形式的に証明することは難しくありません。

```
Hint Constructors plusR.
```

```
Theorem plus_plusR : ∀ n m,
  plusR n m (n + m).
  induction n; crush.
Qed.

Theorem plusR_plus : ∀ n m r,
  plusR n m r
  → r = n + m.
  induction 1; crush.
Qed.
```

plus の関数定義では、算術についての簡単な等式は計算に従います。

```
Example four_plus_three : 4 + 3 = 7.
  reflexivity.
Qed.

Print four_plus_three.
```

four_plus_three = eq_refl

関係の定義では、同じ等式は証明するのにより多くのステップをとりますが、そのプロセスは完全に機械的です。例えば、simple-minded な手動の証明探索の戦略を考えよう。後でエラーメッセージの表示された手順は、最終的なスクリプトから省略されます。

```
Example four_plus_three' : plusR 4 3 7.
  apply PlusO.
Error: Impossible to unify "plusR 0 ?24 ?24" with "plusR 4 3 7".

  apply PlusS.
  apply PlusO.
Error: Impossible to unify "plusR 0 ?25 ?25" with "plusR 3 3 6".

  apply PlusS.
  apply PlusO.
Error: Impossible to unify "plusR 0 ?26 ?26" with "plusR 2 3 5".

  apply PlusS.
  apply PlusO.
Error: Impossible to unify "plusR 0 ?27 ?27" with "plusR 1 3 4".

  apply PlusS.
  apply PlusO.
```

この時点で証明が完了します。単純な手続きが、この種類の全ての証明を見つけられることは、明かに間違いありません。ふたつの候補となるステップ apply PlusO と apply PlusS

からなる可能な証明木のすべてを探検しただけです。先に `Hint Constructors` をふたつの候補となる 証明のステップ のヒントとして登録するのに使ったので、組み込みタクティク `auto` は、この戦略に従います。

```
Restart.
  auto.
Qed.
```

```
Print four_plus_three'.
```

four_plus_three' = PlusS (PlusS (PlusS (PlusS (PlusO 3))))

すこし複雑なゴールについて同じアプローチを試してみましょう。

```
Example five_plus_three : plusR 5 3 8.
  auto.
```

この場合は、`auto` は進捗をするのに十分ではありません。

単一の候補のステップは、可能な証明木の無限のスペースに導くかもしれないので、`auto` は考慮するべき最大の木の深さをパラメータとして与えられるようになっています。ディフォルトの深さは 5 で、ゴールを証明するために深さ 6 が必要ならそのように設定します。

```
  auto 6.
```

しばしば、`info` タクティカルは、`auto` が見つける証明木の記述を見るのに便利です。(このタクティカルは、これを書いている Coq 8.4 では使用できませんが、すぐに再登場することを期待しています。特別な場合の *info_auto* タクティクは、`auto` の饒舌な置き換えとして提供されます。)

```
Restart.
  info auto 6.
```

```
 == apply PlusS; apply PlusS; apply PlusS; apply PlusS;
     apply PlusS; apply PlusO.
```

```
Qed.
```

論理プログラミングのふたつのキーとなるコンポーネントは、*backtracking* バックトラッキングと *unification* ユニフィケーションです。

これらの技法を実際に見るために、さらに単純 (silly) な例を考えます。ここでは、候補となる証明のステップが反射的で、量化子を具体化します。(quantifier instantiation)

```
Example seven_minus_three : ∃ x, x + 3 = 7.
```

説明のために、計算について最小限の理解をもったユーザーを考えましょう。量化子を具体化することを選ぶことによって始めます。ex_intro は、存在量化された式のためのコンストラクタであることを思い出しましょう。

```
  apply ex_intro with 0.
```

```
  reflexivity.
```

```
  Error: Impossible to unify "7" with "0 + 3".
```

これは、デッドエンドのように見えます。apply を実行した箇所に *backtrack* バックトラックして、よりよい別の選択をします。

```
Restart.
  apply ex_intro with 4.
  reflexivity.
Qed.
```

これは、かなり退屈なバックトラッキングの例です。一般に、auto に渡された深さの上限までなら、証明木を導くために、異なった証明のステップの候補が見つからない場合は、証明の途中の (under-construction) の証明木のどんなノードでも、任意の回数のバックトラッキングの行き先になるでしょう。

次に、加算についての関係式 (formulation) を切り替えるとき、それを簡単にするユニフィケーションの実演をします。

Example seven_minus_three' : $\exists$ x, **plusR** x 3 7.

以前とおなじように、手作業で量化子の具体化を推測しようとしますが、ここではそれは必要ありません。apply の代わりに、推測を延期したいそれらのパラメータの代わりにプレースホルダー *unification variables* ユニフィケーション変数をつかう (proceeds with) eapply を使用します。

```
  eapply ex_intro.
```

```
1 subgoal

  ============================
   plusR ?70 3 7
```

いまでは、正しく **plusR** のコンストラクタを適用することで証明を終了することができます。なお、新しいユニフィケーション変数は新しい unknown として生成されます。

```
  apply PlusS.
```

```
  ============================
   plusR ?71 3 6
```

```
  apply PlusS. apply PlusS. apply PlusS.
```

```
  ============================
   plusR ?74 3 3
```

```
  apply PlusO.
```

auto タクティクはユニフィケーション変数を導入するステップの順番を実行しませんが、eauto タクティクは実行します。eauto スタイルの証明探索は、より多くの可能な証明木を網羅できず、それゆえ、実行するのにより長く時間が掛かるので、このふたつのタクティクを使い分けると便利です。

```
Restart.
```

```
  info eauto 6.

 == eapply ex_intro; apply PlusS; apply PlusS;
    apply PlusS; apply PlusS; apply PlusO.
Qed.
```

この証明は、論理プログラミングが、関数プログラミングと比べて、証明探索を単純化する最初の例を与えます。一般に、関数型プログラムは、単に一方方向に実行されることを意味し、関数は入力と出力の重なりのない集合 (disjoint sets) です。最後の例では、特定の出力を生じさせる入力を推定するために、論理型プログラムを後ろ向きに効率的に実行しました。他の入力の未知の値を推測するために、同じことが働きます。

```
Example seven_minus_four' : ∃ x, plusR 4 x 7.
  eauto 6.
Qed.
```

正しい補助的な事実を証明するために、具体的な関数プログラムについて、上で論理プログラムにしたのと同じ方法で、理由づけすることができます。

plusR のコンストラクタが、plus についての補題を自然に翻訳することを証明しましょう。

最初に、左辺または右辺がワイルドカードとマッチする等式を証明するライブラリ関数を見つけるためのコマンドである SearchRewrite コマンドを使って、それらの補題が、既に標準ライブラリで証明されていることを見つけることができます。

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

plus_O_n: ∀ n : **nat**, $0 + n = n$

コマンド Hint Immediate は、auto と eauto が、この補題を証明木の任意の葉の候補として考慮するように依頼します。

```
Hint Immediate plus_O_n.
```

PlusS に対応するものを証明します。

```
Lemma plusS : ∀ n m r,
  n + m = r
  → S n + m = S r.
  crush.
Qed.
```

コマンド Hint Resolve は、証明木の葉のみならず、任意のレベルにおいて試みられる新しい証明のステップの候補に加えます。

```
Hint Resolve plusS.
```

適切なヒントを登録したので、これまでの例を通常の機能追加 plus で再現できます。

```
Example seven_minus_three'' : ∃ x, x + 3 = 7.
  eauto 6.
Qed.

Example seven_minus_four : ∃ x, 4 + x = 7.
```

```
  eauto 6.
Qed.
```

さらなる例では eauto が終了しないことがわかるので、この新しいヒントデータベースは、完全な決定可能な手続き (complete decision procedure) からかけ離れています。

```
Example seven_minus_four_zero : ∃ x, 4 + x + 0 = 7.
  eauto 6.
Abort.
```

さらなる補題が便利でしょう。

```
Lemma plusO : ∀ n m,
  n = m
  → n + 0 = m.
  crush.
Qed.

Hint Resolve plusO.
```

もし、 plus への入力を論理プログラムに関連する入力として考えるなら、新しいルール *plus0* は曖昧さを導きいれます。例えば、和 $0+0$ は、我々の注目するオペランドに依存して、*plus_0_n* と *plus0* のどちらにもマッチします。

この曖昧さは潜在的な探索木の数を増やし、証明探索を遅くし、意味的には問題を供さないが、実際に自動化した証明につながります。

```
Example seven_minus_four_zero : ∃ x, 4 + x + 0 = 7.
  eauto 7.
Qed.
```

可能な証明木のスペースを広げるヒントを追加することが、どれだけ損害を被ることになるでしょうか？古典的なものは、このライブラリの定理で具体的にしたように、等式についての推移性 (transitivity) の無制限な使用から来ています。

```
Check eq_trans.

eq_trans
     : ∀ (A : Type) (x y z : A), x = y → y = z → x = z
```

ヒントはセクションのスコープ上にあるので、不幸なヒントの選択の効果を含むためにセクションに入ります。

```
Section slow.
  Hint Resolve eq_trans.
```

以下の事実は間違いですが、eauto のステップは、その証明の検索に非常に長い時間を費やすことを止めません。実行にいかに長くの時間を要したかを測るのために便利なコマンド Time を使います。以下のステップは何も進ませません。

```
  Example zero_minus_one : ∃ x, 1 + x = 0.
    Time eauto 1.
Finished transaction in 0. secs (0.u,0.s)
```

```
Time eauto 2.
Finished transaction in 0. secs (0.u,0.s)

Time eauto 3.
Finished transaction in 0. secs (0.008u,0.s)

Time eauto 4.
Finished transaction in 0. secs (0.068005u,0.004s)

Time eauto 5.
Finished transaction in 2. secs (1.92012u,0.044003s)
```

実行時間が指数的に増加することを心配なら、 debug タクティカルは、eauto がどこで時間を無駄にしているのかを見て、試行されるすべての証明ステップのトレースを出力するのに役立ちます。ルール eq_trans は証明木の各ノードで apply し、eauto はがそれらの全ての位置で試します。

debug eauto 3.

1 *depth*=3
1.1 *depth*=2 eapply ex_intro
1.1.1 *depth*=1 apply plusO
1.1.1.1 *depth*=0 eapply eq_trans
1.1.2 *depth*=1 eapply eq_trans
1.1.2.1 *depth*=1 apply plus_n_O
1.1.2.1.1 *depth*=0 apply plusO
1.1.2.1.2 *depth*=0 eapply eq_trans
1.1.2.2 *depth*=1 apply @eq_refl
1.1.2.2.1 *depth*=0 apply plusO
1.1.2.2.2 *depth*=0 eapply eq_trans
1.1.2.3 *depth*=1 apply eq_add_S ; trivial
1.1.2.3.1 *depth*=0 apply plusO
1.1.2.3.2 *depth*=0 eapply eq_trans
1.1.2.4 *depth*=1 apply eq_sym ; trivial
1.1.2.4.1 *depth*=0 eapply eq_trans
1.1.2.5 *depth*=0 apply plusO
1.1.2.6 *depth*=0 apply plusS
1.1.2.7 *depth*=0 apply f_equal (*A*:=**nat**)
1.1.2.8 *depth*=0 apply f_equal2 (*A1*:=**nat**) (*A2*:=**nat**)
1.1.2.9 *depth*=0 eapply eq_trans

Abort.
End slow.

ときには、しかし、推移性 (transitivity) は、eauto で自動的に通過する証明を得るのに必要なものです。この場合は、必要によって呼び出すことのできる、異なった別のグループ分離するために、名前を付けた *hint databases* ヒントデータベースを使うことができます。ここで、eq_trans をデータベース slow に置きます。

```
Hint Resolve eq_trans : slow.
```

```
Example from_one_to_zero : ∃ x, 1 + x = 0.
  Time eauto.
```

```
Finished transaction in 0. secs (0.004u,0.s)
```

この eauto は、ゴールを証明するのに失敗しますが、しかし、少なくともそれは上記の 2 秒よりも実質的に短い時間で済みます！ Abort.

上のひとつの単純な例は、推移性 (transitivity) によって汚染されることなしに、同じ合計時間で実行されます。

```
Example seven_minus_three_again : ∃ x, x + 3 = 7.
  Time eauto 6.
```

```
Finished transaction in 0. secs (0.004001u,0.s)
```

```
Qed.
```

推移性を必要とするとき、それを明示的に求めます。

```
Example needs_trans : ∀ x y, 1 + x = y
  → y = 2
  → ∃ z, z + x = 3.
  info eauto with slow.
```

```
 == intro x; intro y; intro H; intro H0; simple eapply ex_intro;
    apply plusS; simple eapply eq_trans.
    exact H.

    exact H0.
```

```
Qed.
```

info のトレースは、eq_trans がちょうど証明を完成するのに必要な位置で使われていることを示します。auto と eauto は、intro を、ステップを証明木の深さの上限をに向かって数えるとなしに、いつも実行します。

12.2 拘束されていない値を探す

リストの length 関数の定義を思いだしましょう。

```
Print length.
```

```
length =
fun A : Type ⇒
```

```
fix length (l : list A) : nat :=
  match l with
  | nil ⇒ 0
  | _ :: l' ⇒ S (length l')
  end
```

この関数は、前向きの方向に、入力から出力に計算するので簡単です。

```
Example length_1_2 : length (1 :: 2 :: nil) = 2.
  auto.
Qed.
```

```
Print length_1_2.
```

```
length_1_2 = eq_refl
```

最後の節で、出力からの入力を計算するのを助けるために、論理プログラミングのスタイルで length を再構築するためのいくつかの補題を証明します。

```
Theorem length_O : ∀ A, length (nil (A := A)) = O.
  crush.
Qed.
```

```
Theorem length_S : ∀ A (h : A) t n,
  length t = n
  → length (h :: t) = S n.
  crush.
Qed.
```

```
Hint Resolve length_O length_S.
```

長さ 2 の **list nat** が存在することを証明するためのヒントとして、これらを適用しましょう。(ここで、単に length_S と同じコマンドを使うのが便利なので、length_O を `Hint Immediate` ではなく `Hint Resolve` で登録しています。`Resolve` と `Immediate` は、前提のない定理をヒントにするなら同じ意味を持ちます。

```
Example length_is_2 : ∃ ls : list nat, length ls = 2.
  eauto.
```

```
No more subgoals but non-instantiated existential variables:
Existential 1 = ?20249 : [ |- nat]
Existential 2 = ?20252 : [ |- nat]
```

Coq は、証明探索の途中で導入された幾つかのユニフィケーション変数を決定することなしに、証明を終えたことに文句をいいます。型 **nat** の **任意** の値(たとば 0)をどちらの変数にも差し込むことができますから、このエラーメッセージは少し馬鹿げて (silly) います!しかし、より複雑な型では、それらの値 (inhabitants) を見つけることは、一般的に定理証明と同じくらい複雑かもしれません。

`Show Proof` コマンドは、それらが使われたところで明示的に出現する未定のユニフィケーション変数と一緒に、`eauto` 見つけたものを正確に示します。

```
Show Proof.

Proof: ex_intro (fun ls : list nat => length ls = 2)
         (?20249 :: ?20252 :: nil)
         (length_S ?20249 (?20252 :: nil)
            (length_S ?20252 nil (length_O nat)))
Abort.
```

そのリストのふたつの要素を表わす、ふたつのユニフィケーション変数があります。しかしながら、リストの長さはデータの値と独立です。証明探索はユニフィケーションによって適切なデータの要素を自然に見つけ出すので、逆説的に、リストの制約によって証明探索のプロセスはより簡単になります。ライブラリ述語 **Forall** は便利でしょう。

```
Print Forall.

Inductive Forall (A : Type) (P : A → Prop) : list A → Prop :=
    Forall_nil : Forall P nil
  | Forall_cons : ∀ (x : A) (l : list A),
                  P x → Forall P l → Forall P (x :: l)

Example length_is_2 : ∃ ls : list nat, length ls = 2
  ∧ Forall (fun n ⇒ n ≥ 1) ls.
  eauto 9.
Qed.
```

証明の項を印刷することによって、eauto がどのリストを見つけたかを見ることができます。

```
Print length_is_2.

length_is_2 =
ex_intro
  (fun ls : list nat ⇒ length ls = 2 ∧ Forall (fun n : nat ⇒ n ≥ 1) ls)
  (1 :: 1 :: nil)
  (conj (length_S 1 (1 :: nil) (length_S 1 nil (length_O nat)))
     (Forall_cons 1 (le_n 1)
        (Forall_cons 1 (le_n 1) (Forall_nil (fun n : nat ⇒ n ≥ 1)))))
```

もうひとつ、奇妙な (fancier) 例を見てみましょう。最初に、ひとつのリストのすべてのデータ要素を合計するための関数を定義するために標準の高階関数を使います。

```
Definition sum := fold_right plus O.
```

他の基本的な補題は、証明探索を導くのに便利です。

```
Lemma plusO' : ∀ n m,
  n = m
  → 0 + n = m.
  crush.
```

```
Qed.

Hint Resolve plusO'.
```

最後に、カスタムヒントを登録するためのコマンド Hint Extern を見ます。これは、証明探索の間にゴールに対してマッチするパターンを提供します。ひとたび、パターンがマッチしたら、タクティク（矢印 ⇒ の右辺で与えられた) が試みられます。次に、数字 1 は、このステップの優先度を与えます。低い優先度は高い優先度の前に試みられ、それは、証明探索の時間に大きな影響をもたらすことができます。

```
Hint Extern 1 (sum _ = _) => simpl.
```

ここで、合計が 0 である長さ 2 のリストを見つけることができます。

```
Example length_and_sum : ∃ ls : list nat, length ls = 2
  ∧ sum ls = O.
  eauto 7.
Qed.
```

証明の項を印刷することは（で)、見つかったなんの不思議もない (unsurprising) リストを示します。ここに、どのリストが使用されるのかが明白でない例があります。eauto が選んだリストがどれか判りますか？

```
Example length_and_sum' : ∃ ls : list nat, length ls = 5
  ∧ sum ls = 42.
  eauto 15.
Qed.
```

上記のリストはあまりにも多くのゼロが含まれているので、私たちは答えの一部を放棄して、望むよりも面白くないと言うでしょう。さらなる制約は、問題のより小さい具体例 (インスタンス) に対して、異なる答えを強制します。

```
Example length_and_sum'' : ∃ ls : list nat, length ls = 2
  ∧ sum ls = 3
  ∧ Forall (fun n => n ≠ 0) ls.
  eauto 11.
Qed.
```

この種類の練習を通して続けることができましたが、リストを自動的に検索するよりもさらに興味深いのは **プログラム** を自動的に見つけることでしょう。

12.3 プログラムの合成

ここに算術式の単純な構文の型があります。これは、本書で以前に数回使用したものと似ています。この場合、ちょうどひとつだけの識別された変数を式のなかに言及することができます。

```
Inductive exp : Set :=
| Const : nat → exp
```

```
| Var : exp
| Plus : exp → exp → exp.
```

帰納的な関係は式の意味を指定し、変数値と式を式の値に関連付けます。

```
Inductive eval (var : nat) : exp → nat → Prop :=
| EvalConst : ∀ n, eval var (Const n) n
| EvalVar : eval var Var var
| EvalPlus : ∀ e1 e2 n1 n2, eval var e1 n1
  → eval var e2 n2
  → eval var (Plus e1 e2) (n1 + n2).

Hint Constructors eval.
```

auto を 特定の式の意味を実行するために使えます。

```
Example eval1 : ∀ var, eval var (Plus Var (Plus (Const 8) Var)) (var + (8 + var)).
  auto.
Qed.
```

不幸にも、eval のコンストラクタは、次のように、算術的な同一性 に依存するような定理を証明するのに十分ではありません。

```
Example eval1' : ∀ var, eval var (Plus Var (Plus (Const 8) Var)) (2 × var + 8).
  eauto.
Abort.
```

eval1' を証明するのを助けるために、別の等式の前提を挿入する、別なバージョンの EvalPlus を証明します。

この種類の演出は、eauto のユニフィケーションの制限を回避するのに役に立ちます。直接的なヒントとしての EvalPlus は、結果が既に定数ではなく加算として表現されているゴールにのみ一致します。

下の代替バージョンでは、加算の木がどのように分解されたとしても、最初の 2 つの前提を証明するために、eauto は *n1* と *n2* の値を決める際に自由な範囲 (regin) を与え、3 番目の前提は reflexivity によって証明されます。

```
Theorem EvalPlus' : ∀ var e1 e2 n1 n2 n, eval var e1 n1
  → eval var e2 n2
  → n1 + n2 = n
  → eval var (Plus e1 e2) n.
  crush.
Qed.

Hint Resolve EvalPlus'.
```

さらに、eauto を量化子のない線形算術 (quantifier-free linear arithmetic) のための完全な決定手順を提供する標準タクティクである omega に適用させせます。Hint Extern を通して、どんな等式のゴールに対してでも omega を使うように要求します。

abstract タクティカルは、成功した証明のそれぞれに対して新しい補題を生成し、最後の証明項において、その補題は、算術的に等しいこと (arithmetic equality) の全体の証明を置き換える箇所で参照されます。

```
Hint Extern 1 (_ = _) ⇒ abstract omega.
```

eval1' に戻って、それを自動的に証明します。

```
Example eval1' : ∀ var, eval var (Plus Var (Plus (Const 8) Var)) (2 × var + 8).
  eauto.
Qed.

Print eval1'.
```

```
eval1' =
fun var : nat ⇒
EvalPlus' (EvalVar var) (EvalPlus (EvalConst var 8) (EvalVar var))
  (eval1'_subproof var)
     : ∀ var : nat,
        eval var (Plus Var (Plus (Const 8) Var)) (2 × var + 8)
```

補題 *eval1'_subproof* は abstract omega によって生成されました。

これで、常に特定のシンボリック値に評価されるプログラム（算術式）を検索することで、論理プログラミングの柔軟性を利用する準備が整いました。

```
Example synthesize1 : ∃ e, ∀ var, eval var e (var + 7).
  eauto.
Qed.

Print synthesize1.
```

```
synthesize1 =
ex_intro (fun e : exp ⇒ ∀ var : nat, eval var e (var + 7))
  (Plus Var (Const 7))
  (fun var : nat ⇒ EvalPlus (EvalVar var) (EvalConst var 7))
```

プログラム合成の能力を見せてくれるもう 2 つの例があります。

```
Example synthesize2 : ∃ e, ∀ var, eval var e (2 × var + 8).
  eauto.
Qed.

Example synthesize3 : ∃ e, ∀ var, eval var e (3 × var + 42).
  eauto.
Qed.
```

これらの例は、変数 var の線形な式を示します。これらのどんな式でも、適当な k と n なら、$k \times var + n$ に等しいです。任意の式の意味がこのような線形の式に等しいということを証明できることは、多分それほど驚きではありませんが、そのような事実を手動で証明するとは退屈です。このセクションを終了するにあたり、eauto を使って証明を完成させ、k と n の値を自動的に見つけてみましょう。

一連の補題を証明してヒントとして追加します。別の eval コンストラクタと補題と算術に関するいくつかの事実があります。

```
Theorem EvalConst' : ∀ var n m, n = m
```

```
  → eval var (Const n) m.
  crush.
Qed.

Hint Resolve EvalConst'.

Theorem zero_times : ∀ n m r,
  r = m
  → r = 0 × n + m.
  crush.
Qed.

Hint Resolve zero_times.

Theorem EvalVar' : ∀ var n,
  var = n
  → eval var Var n.
  crush.
Qed.

Hint Resolve EvalVar'.

Theorem plus_0 : ∀ n r,
  r = n
  → r = n + 0.
  crush.
Qed.

Theorem times_1 : ∀ n, n = 1 × n.
  crush.
Qed.

Hint Resolve plus_0 times_1.
```

もうひとつ、定理に特に特化した算術的な補題で終わりましょう。この事実は、半環代数構造の公理に従うので、半環なら自然に (since the naturals form a semiring)、組み込みタクティク ring を使うことができます。

```
Require Import Arith Ring.

Theorem combine : ∀ x k1 k2 n1 n2,
  (k1 × x + n1) + (k2 × x + n2) = (k1 + k2) × x + (n1 + n2).
  intros; ring.
Qed.

Hint Resolve combine.
```

私たちのヒントの選択は、k と n の値を選ぶ手続きを電報で伝える (telegraphing) ようで、ちょっといんちきです (cheating, to an extent)。それにもかかわらず、これらの補題が定位置にある場合、補題の構成を明示的に編成せずに自動証明を成しとげることができます。

```
Theorem linear : ∀ e, ∃ k, ∃ n,
```

$\forall$ *var*, **eval** *var* *e* ($k \times var + n$).
  induction *e*; *crush*; eauto.
Qed.

証明項を印刷することで、入力項ごとに定数を選ぶための手続きを見ることができます。

12.4 さらに auto のヒントについて

ここで止めて、auto と eauto のヒントの可能性を検討しましょう。ヒントは、これまでの多くの例で拡張されるのを見た、 **ヒントデータベース** に格納されます。ヒントデータベースが指定されていない場合は、デフォルトのデータベースが使用されます。auto または eauto によって、デフォルトデータベースのヒントが常に使用されます。命令的に (imperatively) ヒントデータベースを拡張する機会 (chance) は重要で、なぜなら、Ltac プログラミング言語において、異なるソースファイル内の異なるモジュールによってシームレスにその値を拡張することのできる「グローバル変数」を作ることができないためです。*crush* はすべてについて一度定義できたところで、開発の間にヒントを自動的に適用しながら、これまでのところヒントの利点を見てきました。実際、*crush* は auto の観点から定義されており、この拡張性をどのように達成するかが説明されています。他のユーザー定義のタクティクでも、auto や eauto と同様の利点が得られます。

auto と eauto についての基本的なヒント Hint Immediate *lemma* は、補題を適用し、仮説を打ち消して (discharging)、ただちに目標をひとつの証明のステップで解くように依頼します。

Resolve *lemma* は、同じことをしますが、ネストされた証明検索の対象となる新しい前提を追加するかもしれません。

Constructors *type* は、Resolve のように動作し、帰納的なすべてのコンストラクタに適用されます。

Unfold *ident* は、証明のゴールの先頭に現れるときに *ident* を展開 (unfolding) することを試みます。

これらの Hint コマンドのそれぞれに接尾辞を付けて、指定されたデータベースのみにヒントを追加することができます。たとえば、Hint Resolve *lemma* : *my_db* で、ヒントを特定のデータベースに追加したなら、例えば auto with *my_db* のときだけに使われるでしょう。

auto の追加引数は、auto 8 や auto 8 with *my_db* のように、深さ優先で検索する証明木の最大の深さを指定します。デフォルトの深さは 5 です。

これらの Hint コマンドはすべて、より原始的なヒントの種類 Extern で表現できます。Hint Extern の例をいくつか挙げると、より多くの可能性が示されます。

Theorem bool_neq : true $\neq$ false.
  auto.

A call to *crush* would have discharged this goal, but the default hint database for auto contains no hint that applies.

Abort.

私たちが証明しようとしている定理の再記述だけではない **bool** 特有のヒントを思いつく

のは難しいです。幸運なことに、組み込みタクティク congruence と、等式の定理について完全な手続き、解釈されない関数、データ型のコンストラクタを適用することにより、十分簡単になります。

```
Hint Extern 1 (_ ≠ _) ⇒ congruence.

Theorem bool_neq : true ≠ false.
  auto.
Qed.
```

Hint *Exter* は、全部 Ltac 言語で実装されているかもしれません。この例は、ヒントがどこで match を使うか示しています。

```
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 の呼び出しは、*intor* が行うことよりも進捗しない。なぜなら、*both* の結論はゴールの結論とユニファイしないため、auto の呼び出しは、証明するために前提 *both* を適用しない。しかしながら、auto にこの種類のゴールの扱いを教えることができる。

```
    Hint Extern 1 (P ?X) ⇒
      match goal with
        | [ H : ∀ x, P x ∧ _ ⊢ _ ] ⇒ apply (proj1 (H X))
      end.

    auto.
  Qed.
```

Extern パターンは、関連するタクティクで使った、ユニフィケーション変数を束縛するかもしれないことがわかります。proj1 関数 は、U の証明を $U \wedge V$ の証明から取り出す (extracting) 標準ライブラリから取り入れました。

```
End forall_and.
```

この例が成功したあと、より野心的になって、すべての可能な述語 P へのヒントを一般化することを探し求めるかもしれません。

```
Hint Extern 1 (?P ?X) ⇒
  match goal with
    | [ H : ∀ x, P x ∧ _ ⊢ _ ] ⇒ apply (proj1 (H X))
  end.

User error: Bound head variable
```

Coq の auto のヒントデータベースは、タクティクが試みるためのリストとして、*head symbols* **頭部シンボル** をマッピングするように働きます。これはなぜなら、Extern パター

ンの定数の頭部は静的に決定されるべきだからです。最初の Extern ヒントにおいて、$x \neq y$ の構文糖衣を取り除く (desugars) と not (**eq** x y) なので、頭部シンボルは not でした。次の例では、頭部シンボルは P でした。

幸いにも、より基本的な形式である Hint Extern にも適用されます。⇒ の左側にパターンを残して、対応するロジックを Ltac スクリプトに組み込むことができます。

```
Hint Extern 1 ⇒
  match goal with
    | [ H : ∀ x, ?P x ∧ _ ⊢ ?P ?X ] ⇒ apply (proj1 (H X))
  end.
```

この種の Hint Extern は、証明木の **すべての** ノードに適用されることに注意してください。高価な Ltac スクリプトは、証明検索を大幅に遅くする可能性があります。

12.5 Rewrite のヒント

ヒントの他の側面は量化された等式を書き換えることです。関連するコマンド Hint Rewrite をこれまでの多くの例で使ったことがありました。*crush* タクティクは、これらのヒントを組み込みタクティク autorewrite を呼び出すことで使います。rewrite ヒントは、デフォルトではディフォルトデータベース *core* に追加される Hint Rewrite *lemma* のかたちをとっています。しかし、別のヒントデータベースも Hint Resolve のように指定することもできます。

次の例は autorewrite を直接使うことを示します。Hint Rewrite はディフォルトのデータベースを使うので、autorewrite はその名前のデータベースが必要です。

```
Section autorewrite.
  Variable A : Set.
  Variable f : A → A.

  Hypothesis f_f : ∀ x, f (f x) = f x.

  Hint Rewrite f_f.

  Lemma f_f_f : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with core; reflexivity.
  Qed.
```

ヒントの選択に十分な注意を払わないときに、autorewrite が問題を引き起こす幾つかの方法があります。最初に、ヒントの集合が終わらない書き換えのシステムを定義しているとき、その場合、autorewrite の呼び出しが終わらないことがあります。次に、誤った道筋 (path) に autorewrite を導くヒントを加えることがあります。例えば：

```
  Section garden_path.
    Variable g : A → A.
    Hypothesis f_g : ∀ x, f x = g x.
    Hint Rewrite f_g.

    Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
```

```
    intros; autorewrite with core.
```

```
============================
 g (g (g x)) = g x
```

```
  Abort.
```

新しいヒントは、古いヒントがもう適用できないかたち (form) となった、ゴールに適用されます。この、新しいヒントが証明探索を遅くするかもしれない、auto の状況と対比される rewrite ヒントの “非単調性 (non-monotonicity)” は、古い証明を “壊す (break)” ことは決してありません。

キーとなる違いは、autorewrite は、ゴールを解くことなしに変形するかもしれないのに対して、auto は、ゴールと解くか、または、それを変えないかのどちらかです。ヒントデータベースを変更すると特定のゴールについて見つかった証明が変更される可能性があり、その証明は、証明状態の他の場所に表示されるユニフィケーション変数の設定に影響を与える可能性があるので、eauto の状況はやや複雑です。

```
Reset garden_path.
```

autorewrite タクティクは、追加の前提を含む量化された等式でも機能しますが、同様の誤った書き換えを避けるように注意する必要があります。

```
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.

  Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with core.

============================
 g (g (g x)) = g x

subgoal 2 is:
 P x
subgoal 3 is:
 P (f x)
subgoal 4 is:
 P (f x)

  Abort.
```

前提を証明することができないことを知っているにもかかわらず、不適切なルールは、前と同様に 3 回実行 (fire) しました。

```
Reset garden_path.
```

最終的な成功は、生成された前提に適用するタクティクを指定する Hint Rewrite に対する追加の引数を使用します。そのようなヒントは、タクティクがすべての前提で成功した場合にのみ使用され、前提によってはいくつかの前提のためにさらにサブゴールを残します。

```
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.

  Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with core; reflexivity.
  Qed.
```

生成された前提が仮定の中にあるとき、さらに、autorewrite を f_g に適用します。

```
  Lemma f_f_f_g : ∀ x, P x → f (f x) = g x.
    intros; autorewrite with core; reflexivity.
  Qed.
End garden_path.
```

結論に対するのと同様に、仮定に対して書き換えをするとき、autorewrite with db in * のかたちを適用するのは便利です。

```
Lemma in_star : ∀ x y, f (f (f (f x))) = f (f y)
  → f x = f (f (f y)).
  intros; autorewrite with core in *; assumption.
Qed.

End autorewrite.
```

多くの証明は、auto と autorewrite の巧妙な組み合わせで、快適にモジュール化された方法で自動化することができます。

Chapter 13

Ltac による証明探索

これまで多くの証明の自動化の例を見てきましたが、Coq の証明探索の手続きのためのドメイン固有言語 (domain-specific language) である Ltac によって、一部のコードの断片は魅力的なものでした (some with tantalizing code snippets)。この章は、Ltac の機能をボトムアップに示すことを目標に、特に、バックトラッキング探索のための独創的なアプローチをサポートする、Ltac の `match` 構成要素 (construct) に焦点をあてます。最初に、Coq に組み込まれている便利な自動化タクティクをいくつか実行します。それらの詳細はマニュアルに記載されているので、出来ることを概説するだけにします。

13.1 組み込み 自動化タクティク

多くのタクティクが *crush* によって繰り返し呼び出されます。

`intuition` タクティクは、ゴールの命題論理的構造を簡略化 (simplifies) します。

`congruence` タクティクは、等式と合同閉包 (congruence closure) のルールに加え、帰納型のコンストラクタの属性を適用します。

`omega` タクティクは、あなたの求めに応じて (depending on whom you ask)、quantifier-free linear arithmetic 量化子のない線形算術、または、Presburger arithmetic プレスバーガー算術と呼ばれる理論に対する完全な決定手続きを提供します。

すなわち `omega` は、その原始式 (atomic formulas) が、自然数または整数を基本的な比較演算の対象とし、そのオペランドが定数、変数、加算および減算から構成された、（加算または減算の省略形として利用可能な定数による乗算を含む）命題論理の式として解釈されることのできる、ゴールの部分のみを見ることに続く (that follows from looking only at parts of that goal) 任意のゴールを証明します。

`ring` は、関与する型に依存して、(代数のように) 環まはた半環の公理を適用することによって、ゴールを解きます。Coq の開発では、関連する公理を証明することによって、新しい型を環と半環の一部として宣言することができます。環における分数に変換することで、体の値を簡略化するための同様のタクティク `field` があります。`ring` と `field` の両方は、等式のゴールだけを解くことができます。`fourier` タクティクは、Coq 標準ライブラリで公理化された実数の不等式を証明するフーリエの方法を使用します。

The *setoid* の手法 (facility) は、`rewrite` のようなタクティクによって理解される新しい

等価関係 (equivalence relations) を登録することを可能にします。

たとえば Prop は、“if and only if” の等価関係を setoid として登録されています。新しい setoid を登録する能力は、すべての推論 (reasoning) が、「関係によって改変された (modding out by a relation)」後に実行される箇所において、数学で一般的な種類の証明において非常に有用です。

Coq マニュアルを熟読することで学ぶことのできる、組み込みの「ブラックボックス」な自動化タクティクがあります。

Coq の本当の約束事は、Ltac を使って、問題に特化したタクティクのコーディングのなかにあります。

13.2 Ltac プログラミングの基礎

すでに Ltac プロラムの多くの例を見てきました。本章の残りでは、重要な機能とデザインパターンを徹底的に紹介しようとします。

match タクティクのひとつの共通の使い方は、このタクティクの定義にあるように、条件分析 (case analysis) のための内容の識別です。

```
Ltac find_if :=
  match goal with
    | [ ⊢ if ?X then _ else _ ] ⇒ destruct X
  end.
```

このタクティクは結論が if かどうかチェックし、もしそうなら、条件式 (test expression) を destruct します。ある特定の種類の定理は、このようなタクティクによって、自動的に証明することが簡単 (trivial) です。

```
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.
```

ここで使った repeat は、*tactical* タクティカル または タクティク・コンビネータ (tactic combinator) と呼ばれます。repeat t の振る舞いは、t を実行し、すべての生成されたサブゴールに t を実行し、**それら** が生成したサブゴールに t を実行し、というぐあいに、繰り返し続けることです。この探索木の任意の点で t が失敗したとき、その特定のサブゴールは、後のタクティクのによって扱われるために残されます。なので、いつも成功するタクティクと一緒に repeat を使用しないことが重要です。

他のとても便利な Ltac の構成要素 (building block) は、*context patterns* コンテキスト・パターンです。

```
Ltac find_if_inside :=
  match goal with
    | [ ⊢ context[if ?X then _ else _] ] ⇒ destruct X
  end.
```

このタクティクの振る舞いは、結論の if であり、その後に条件式を destruct する任意の部分項 (subterm) を見つけることです。このバージョンは *find_if* を含みます。

```
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.
```

find_if-inside を *find_if* が十分に簡約できなかったゴールを証明するために使うことができます。

```
Theorem hmm2 : ∀ (a b : bool),
  (if a then 42 else 42) = (if b then 42 else 42).
  intros; repeat find_if_inside; reflexivity.
Qed.
```

多くの決定性の手続きは「repeat match ループ」によって Ltac で記述されます。たとえば、tauto のサブセットの機能を実装することができます。

```
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 ⊢ _ ] ⇒ specialize (H1 H2)
         end.
```

match のパターンはユニフィケーション変数を仮定と結論のパターンの間で共有することができ、結論が仮説といつマッチするかを把握することは容易です。exact タクティクは、適切な型の証明項が与えられているとき、ゴールを完全に解きます。

（\index{natural deduction}natural deduction~\cite{TAPLNatDed} 自然演繹の意味において）いくつかの結合子 (* suhara: ∧と∨ *) についての導入のルールを実装することは自明です。

最後のルールは、仮説を与えられた引数の集合（量化された変数、または、含意による局所的仮説のため）に特化したバージョンに置き換える\index{tactics!specialize}[specialize] タクティクを使って三段論法 (modus ponens) を実装します。慣例によって、[specialize] への引数が一連の引数に対する仮説 [H] の適用である場合、その特殊化の結果が [H] に置き換えられます。他の項については、結果は [generalize] と同じです。

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

  Theorem propositional : (P ∨ Q ∨ False) ∧ (P → Q) → True ∧ Q.
    my_tauto.
  Qed.
End propositional.
```

すべての含意 (implication) を消すこと (clearing) によって情報を失なわないので、三段論法の実装をするのは比較的簡単です。もし、量化された含意についての同様の完全な手続きを実装するなら、特定の命題がまだ仮説に含まれていないことを保証する方法が必要です。これを効果的に行うには、最初に match の意味についてもう少し学ぶ必要があります。

マッチは ML のように動作すると想定するのは魅力的です。実際、その動作にはいくつかの重要な違いがあります。ひとつは、変数やコンストラクタに制限されることなく、パターンに任意の式を含めることです。もうひとつは、同じ変数が複数回現れることがあり、暗黙に等値の制約を含むことです。

他のふたつの違いは、他のものよりはるかに重要です。match 構文は、**失敗によるバックトラッキングの意味** を持っています。ML では、パターンマッチングは、一致させる最初のパターンを見つけてから、その本体を実行することによって動作します。本体が例外を発生させた場合、全体の一致は同じ例外を発生させます。Coq では、ケース条件本体の失敗は、代わりにケースのリストを通じて検索を続行します。

例えば、この（不必要で冗長な）証明のスクリプトはこのように動きます。

```
Theorem m1 : True.
  match goal with
    | [ ⊢ _ ] ⇒ intro
    | [ ⊢ True ] ⇒ constructor
  end.
Qed.
```

最初のケースは簡単に一致しますが、その結論は限定子や含意から始まるわけではないので、その本体のタクティクは失敗します。同様の ML のマッチでは、パターンマッチ全体が失敗します。Coq では、次のパターンを取り戻して試してみます。これも一致します。その本体のタクティクは成功するので、全体のタクティクも成功します。

この例では、失敗が match 内の別のパターンにどのように移動するかを示しています。失敗はまた、単一のパターンにマッチする異なる方法を見つける試みを引き起こします。

別の例を考えてみましょう。

```
Theorem m2 : ∀ P Q R : Prop, P → Q → R → Q.
  intros; match goal with
            | [ H : _ ⊢ _ ] ⇒ idtac H
          end.
```

Coq は *H1* を印刷します。ひとつの引数を取って idtac を適用することは、match 情報を取り出すための便利なデバッキングツールです。この match は最初に H を *H1* に束縛しようとしますが、*H1* は Q を証明するために使用できません。それにもかかわらず、タクティクの次のバリエーションはゴールを証明するのに成功します：

```
  match goal with
    | [ H : _ ⊢ _ ] ⇒ exact H
  end.
Qed.
```

タクティクは最初に H と *H1* をユニファイします、その場合は exact H が失敗するため、戦術エンジンは H の可能な値をさらに検索します。最終的には、正しい値 に到達するので、exact H と全体のタクティクは成功します。

では、命題が私たちの仮説の中にないことを確認するためのタクティクを実装する用意ができました：

```
Ltac notHyp P :=
  match goal with
    | [ _ : P ⊢ _ ] ⇒ fail 1
    | _ ⇒
      match P with
        | ?P1 ∧ ?P2 ⇒ first [ notHyp P1 | notHyp P2 | fail 2 ]
        | _ ⇒ idtac
      end
  end.
```

パターン・マッチングにに組み込まれている等価性検査 (equality checking) を使用して、命題に正確に一致する仮説があるかどうかを調べます。もしそうなら、fail タクティクを使用します。

引数がなければ、期待通りに、fail は通常のタクティクの失敗を通知します。fail に引数 n が渡されると、n は、バックトラック探索の囲む条件 (case) を通して外側に向かってカウントするように使用されます。この場合、fail 1 は、「パターン・マッチングの分岐ではなく、*macth* 全体を失敗させる」ことを示します。fail 1 に達すると、2 番目の条件 (case) は決して試行されません。

第 2 の条件 (case) は、P が仮説と一致しないときに使用され、P が連言 (conjunction) であるかどうかをチェックします。

他の簡略化 (simplification) では連言 (conjunction) をその成分の式 (component formula) に分割することがあるため、これらの成分の少なくともひとつも表現されていないことを確認する必要があります。

これを達成するために、 first タクティカルを適用します。これはタクティクのリストを取り、それらのひとつが失敗しないまで リストを続けます。最後の fail 2 は、first と

その周囲に巻き込まれた match の両方が失敗することを示します。

?*P1* ∧ ?*P2* の場合のボディ は、到達すれば完全に成功するか完全に失敗するかを保証します。

ワイルドカードの場合、P は連言ではありませんから、tactics!idtac % [idtac] を使用します。これは、何もしないことで成功する効果があるため、単独で適用することは愚かなタクティクです。それにもかかわらず、[idtac] は、ここで見ているような場合に便利な代用品 (placeholder) です。

(* 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. *)

存在しないことのチェックを実装したことによって、入力として証明項をとり、結論がまだ存在しない場合だけ、新しい仮説としてその結論を加え、さもなければ失敗するタクティクを作ることは簡単です。

```
Ltac extend pf :=
  let t := type of pf in
    notHyp t; generalize pf; intro.
```

Ltac の有用な *type of* 演算子があります。この演算子は Gallina では実装できませんでしたが、Ltac でサポートするのは簡単です。

最終的に t は *pf* の型を束縛します。t がまだ存在していないことを確認します。もしそうなら、generalize / intro の組み合わせを使って、*pf* によって証明された新しい仮説を追加します。

タクティク generalize は、入力として t（例えば命題の証明）をとり、T が t の型であるとき、G から $T \to G$ に結論を変えます。

これらの定義されたタクティクで、単純な一階論理の式の集合のすべての結果 (consequence) をコンテキストに加えるためのタクティク *completer* を書くことができます。

```
Ltac completer :=
  repeat match goal with
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ H : _ ∧ _ ⊢ _ ] ⇒ destruct H
           | [ H : ?P → ?Q, H' : ?P ⊢ _ ] ⇒ specialize (H H')
           | [ ⊢ ∀ x, _ ] ⇒ intro

           | [ H : ∀ x, ?P x → _, H' : ?P ?X ⊢ _ ] ⇒ extend (H X H')
         end.
```

前の扱ったのと同じ種類の連言と含意を使います。→ は ∀ の 依存しない特別な場合なので、第 4 のルールでは intro で含意も扱うことに注意してください。

第 5 のルールでは、私たちが仮説のひとつにマッチする前提で ∀ (を含む) 事実 H を見つけたら、まだ追加していなければ、H の結論を適切に具体化 (instantiation) したものを追加します。

偽作 (spurious) の変数を導入する、説明上の目的 (didactic purpose) の定理を用いて、すぐに、*completer* が正しく働いていることを確認することができます。

```
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 : ∀ (y x : A), P x → S x.
    completer.

  y : A
  x : A
  H : P x
  H0 : Q x
  H3 : R x
  H4 : S x
  ============================
   S x

    assumption.
  Qed.
End firstorder.
```

かろうじて *completer* の定義における微妙な落とし穴を避けていました。慣れていない目には、オリジナルよりも魅力的に見える別の定義を試してみましょう。

（2 番目の match の場合を少し変更して、これまでに行使されていなかった Ltac の動作の微妙な部分を処理するのに十分スマートなタクティクにしました）。

```
Ltac completer' :=
  repeat match goal with
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ H : ?P ∧ ?Q ⊢ _ ] ⇒ destruct H

           | [ H : ?P → _, H' : ?P ⊢ _ ] ⇒ specialize (H H')




           | [ ⊢ ∀ x, _ ] ⇒ intro

           | [ H : ∀ x, ?P x → _, H' : ?P ?X ⊢ _ ] ⇒ extend (H X H')
         end.

Section firstorder'.
  Variable A : Set.
  Variables P Q R S : A → Prop.
  Hypothesis H1 : ∀ x, P x → Q x ∧ R x.
```

Hypothesis $H2$: $\forall\ x,\ R\ x \rightarrow S\ x.$

Theorem fo' : $\forall\ (y\ x\ :\ A),\ P\ x \rightarrow S\ x.$
  completer'.

$y\ :\ A$
$H1\ :\ P\ y \rightarrow Q\ y \wedge R\ y$
$H2\ :\ R\ y \rightarrow S\ y$
$x\ :\ A$
$H : P\ x$
============================
  $S\ x$

証明できていたゴールを証明できないものに変形 (reducing) することで、量化された定理は、x ではなく y で具体化されてしまいます。

completer' のための最後の match の条件は、適切な仮説とともに量化子を具体化するだけに注意深くしています。なぜ間違った選択が行われたのですか？　Abort.
End firstorder'.

いくつかの例が問題を説明しているはずです。ここでは、match ベースの証明がうまくいくのを見ています：

Theorem t1 : $\forall\ x$: **nat**, x = x.
  match goal with
    | [$\vdash \forall\ x$, _] $\Rightarrow$ trivial
  end.
Qed.

This one fails.

Theorem t1' : $\forall\ x$: **nat**, x = x.
  match goal with
    | [$\vdash \forall\ x$, $?P$] $\Rightarrow$ trivial
  end.

```
User error: No matching clauses for match goal
```

Abort.

問題は、ユニフィケーション変数にローカルに束縛された変数が含まれないことです。

この場合、$?P$ はローカルな量化変数 x を含む $x = x$ に束縛される必要があります。以前のバージョンでワイルドカードを使用することで、この制限は回避されました。なぜこの制限が *completer* タクティクの振る舞いに影響を与えるのかを理解するために、Coq では、含意が縮退 (degenerate) した全称量化の略記であることを思い出してください。それにもかかわらず、Ltac パターンでは、Coq はワイルドカードの含意と全称量化のマッチに満足してしまいます。

Coq 8.2 リリースには、明示的な自由変数の集合を持つユニフィケーション変数用の特別なパターン形式が含まれています。そのユニフィケーション変数は、自由変数から「実」値

への関数に束縛されます。Coq 8.1 以前では、このような回避策はありません。15.5 節でこの手の込んだバインディングフォームの例を見ていきます。

どの Coq バージョンを使用していても、この制限に注意することが重要です。すでに示唆したように、その制限は *completer'* の驚くべき振る舞いの背後にある原因です。

私たちは間違って量化された事実を三段論法と突き合わせ、適切に一致する仮説が利用可能であり、誤った量化子の具体化が選択された異なる行動につながるというチェックを回避します。私たちの初期の *completer* タクティクでは、含意の結論と変数を組み合わせた三段論法を使用しています。

実際、ここで示した動作は Coq バージョン 8.4 に適用されますが、8.4pl1 には適用されません。後者のバージョンでは、通常の Ltac パターン変数がローカルに束縛された変数を含む項に一致することができますが、その変数が後に Gallina 項として使用されると、タクティクの失敗が発生します。

13.3 Ltac による関数プログラミング

Ltac は、構文付き Lisp(Lisp-with-syntax) 風の非常に便利な関数型プログラミングをサポートしています。受け入れてもらうための、プログラムに関係するいくつかの構文上の慣習があります。Ltac の構文はタクティクの記述のために最適化されているので、より標準的な関数プログラムを書く際にいくつかの不都合を扱わなければなりません。

説明のために、簡単なリストの長さを求める関数を記述しましょう。あたかも Gallina のように、*Fixtpoint*（とその注釈 (annotation)）を Ltac に置き換えて書き始めます。

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ls' ⇒ S (length ls')
  end.
```

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

この時点で、Ltac では、パターン変数の名前の先頭に疑問符を付ける必要があることを覚えておいてください。

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒ S (length ls')
  end.
```

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

問題は、Ltac が式 S (length *ls'*) を引数 length *ls'* を持つタクティク S の呼び出しとして扱うことです。Gallina の非終端記号の構文解析を「エスケープ」するために特別なアノテーションを使用する必要があります。

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒ constr:(S (length ls'))
  end.
```

この定義は受け入れられます。このような Ltac の定義をテストするのはちょっと厄介です。ひとつの方法があります。

```
Goal False.
  let n := length (1 :: 2 :: 3 :: nil) in
    pose n.
```

```
  n := S (length (2 :: 3 :: nil)) : nat
  ============================
   False
```

特定の項に、等号でセットされた (set equal to)、新しい変数でもって、証明のコンテキストを拡張する pose タクティクを使用します。

pose n の代わりに idtac n を使用することもできました。これは、コンテキストを変更せずに結果を出力します。

n の値は、長さの計算の 1 ステップだけ展開されます。ここで起こったことは、constr 非終端記号にエスケープすることによって、私たちが定義している Ltac 関数 length は、Gallina の関数 length を参照したことです。

```
Abort.
```

```
Reset length.
```

覚えておくべきことは、タクティクによって作られた Gallina 項は、Ltac 呼び出しを他の Gallina 項 に直接挿入するのではなく、let または同様の手法を介して明示的に束縛する必要があるということです。

```
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.

標準のリストの map 関数のこの例が示すように、Ltac で無名関数の式とローカルな関数定義を使用することもできます。

```
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 関数は暗黙の引数を持つことができません。出力リストの要素の型 (carried type of the output list) である T を明示的に渡す必要があることは驚くようです。f は Gallina の項ではなく Ltac の項であり、Ltac プログラムは動的に型付けされているので、*type of f* を使うことはできません。関数 f は、非常に構文的な方法を使用して、異なる入力に対して異なる型の項を返すことを決定することができます。constr:(@nil T) を constr:nil に置き換えることもできませんでした。これは、パラメータを nil に推論するために使用する強く型付けされたコンテキストがないためです。幸いにも、constr:(x' :: ls'') の中には十分なコンテキストを持っています。

複雑なタクティクの表現を別のタクティクに引数として渡したいときには、*map* を呼び出す際にしたように、「非終端のエスケープ」の反対の方向を採用する必要がある場合があります。

```
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.

Ltac スクリプトの中の各位置には、既定の適用可能な非終端記号があります。ここで、constr と ltac は、Gallina と Ltac の条件をそれぞれ念頭に置く価値のある主要なオプションです。明示的なコロン表記は、既定の非終端選択を無効にするためにいつでも使用することができますが、Gallina として解析されるコードではこのような上書きは使用できなくな

ります。ltac 非終端の関数アプリケーションは、Gallina ではなく Ltac のアプリケーションとして扱われます。そのような関数への **引数** はデフォルトで constr で解析されます。この選択は、私たちが書いているすべての証明スクリプトにすべて頼っていることがわかるまで、奇妙に見えるかもしれません！例えば、apply タクティクは Ltac 関数であり、その引数を Ltac ではなく Gallina の項として解釈するのは当然です。上記の *map* の呼び出しのように、ltac 接頭辞を使用して Ltac 関数の引数を Ltac 項として解析します。いくつかの単純なケースでは、Ltac 項は余分なプレフィックスなしで渡されることがあります。例えば、Ltac の意味を持ち、Gallina の意味を持たない識別子は自動的に Ltac で解釈されます。

Ltac の機能プログラムをデバッグしたいときには、もうひとつの問題があります。呼び出された引数のデバッグ・トレースを出力する *length* バージョンを提供するには、次のコードが必要です。

```
Reset length.

Ltac length ls :=
  idtac ls;
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒
      let ls'' := length ls' in
        constr:(S ls'')
  end.
```

Coq はタクティクの定義を受け入れますが、コードには致命的な欠陥があり、常に動的なタイプのエラーにつながります。

```
Goal False.

  let n := length (1 :: 2 :: 3 :: nil) in
    pose n.

Error: variable n should be bound to a term.

Abort.
```

ここで何がうまくいかないのでしょうか？答えは、Ltac の、純粋に関数プログラミング言語と命令的プログラミング言語の両方としての二重状態 (dual status) と関係しています。

基本的なプログラミング言語は純粋に関数的ですが、タクティクスクリプトはそのようなプログラムによって返される「データ型」のひとつで、Coq は証明状態を変更する命令的なセマンティクスを使ってそのようなスクリプトを実行します。

[41, 31] Haskell の モナド・プログラミング (monadic programming) に精通している読者は、類似点を認識しているかもしれません。副作用を伴う Haskell プログラムは、**命令型言語におけるプログラムのコード** (the code of programs in an imperative language) を返す純粋なプログラムと考えることができます。

一部の帯域外 (out-of-band) なメカニズムは、これらの派生プログラムを実行する責任を負います。このようにして、Haskell は純粋なままで、通常の入出力の副作用などをサポートします。Ltac は同じ基本メカニズムを使用しますが、動的に型付けされます。ここで埋め込まれた命令的言語には、これまで適用されてきたすべてのタクティクが含まれます。

基本的な idtac も組み込みの命令型プログラムなので、純粋に機能的なコードと自動的には混合しないかもしれません。実際には、セミコロン演算子だけで Ltac コードのスパンを組み込みのタクティクのスクリプトとしてマークしています。純粋な関数型言語では順序付けの必要がないので、副作用がないため、式を実行してからその値を捨て、別の式に移る理由はありません。

Haskell のモナド（それ自体は巧妙な考え方であると思われますが）に対する類推を避けるための代替的な説明は次のとおりです。Ltac のタクティクのプログラムは、後で実行されるときに望ましい証明の変形 (proof modification) を実行する関数を返します。これらの関数は、他のデータ型や数や Gallina 項とは異なります。Gallina 項でのみ計算された length の以前の正しく動作していたバージョンですが、idtac とセミコロンの使用で示されるように、新しい関数は暗黙的に関数を返しています。しかし、新しいバージョンの length の再帰的呼び出しは、出力として Gallina 項をタクティクの関数ではないと予想するように構成されています。その結果、基本的な動的な型エラーが発生します。おそらくファーストクラスのタクティクのスクリプトが関与しています。

結果は Haskell に似ています。純粋なプログラムを「モナド化」して、副作用にアクセスできるようにする必要があります。問題は組み込まれたタクティクの言語には return の構造がないことです。証明スクリプトは、結果を計算するのではなく、定理を証明することに関するものです。プログラムを **継承渡しスタイル** (continuation-passing style) [35] に変換する必要があるやや難しい回避策を適用することができます。

```
Reset length.

Ltac length ls k :=
  idtac ls;
  match ls with
    | nil ⇒ k O
    | _ :: ?ls' ⇒ length ls' ltac:(fun n ⇒ k (S n))
  end.
```

新しい length は、新しい入力を受け取ります: **継続** (continuation) k は、*length* を呼び出した途中でどのようなプロセスがあったとしても継続するために呼び出される関数です。k に渡される引数は、*length* の戻り値と考えることができます。

```
Goal False.
  length (1 :: 2 :: 3 :: nil) ltac:(fun n ⇒ pose n).
```

```
(1 :: 2 :: 3 :: nil)
(2 :: 3 :: nil)
(3 :: nil)
nil
```

```
Abort.
```

私たちは最初に期待した関数引数のトレースを正確に見て、その後の証明に状態を調べると、変数 n に値 3 が追加されていることがわかります。

Haskell の IO モナドとの比較を考えると、言及する価値のある微妙なことがあります。Haskell の I/O 計算は、実世界のある状態から別の状態への変換器と、返す純粋な値を（理

論的に言えば、少なくとも）表現します。状態の一部は、ヒープ割り当ての変更可能な参照の場合のように、プログラム固有のものもありますが、プログラムが、現実世界に元に戻すことができない副作用を及ぼすお気に入りの例「ミサイル発射」の線に沿っているものもあります。

対照的に、Ltac のスクリプトは、単に 2 つの単純な種類の可変状態を制御するものと考えることができます。第 1 に、証明のサブゴールの現在のシーケンスが存在します。第 2 に、前の章で見たように、証明検索によって導入されたユニフィケーション変数に発見された値を部分的に割り当てます（たとえば、`eauto`）。重要なことは、`match`、`auto`、および、その他の Ltac を構成する組み込みのものによって導かれたバックトラッキングの間には、**この状態のあらゆる変異を元に戻すことができる**ということです。Ltac 証明スクリプトには状態がありますが、それは純粋にローカルなものであり、すべての変更は可逆的です。これは証明検索のための非常に便利なセマンティクスです。

13.4 再帰的な証明探索

量化子をどのように具体化するかの決定は、自動化された一階の定理の証明の最も難しい部分のひとつです。与えられた問題によっては、最終的に命題の推論だけを適用することで、量化子のインスタンス可能な全てからなる有限長のシーケンスを考えることができます。これは、ほぼすべてのゴールに対して悪い考えですが、Ltac の再帰的な証明探索の手順の素晴らしい例になります。

一階の証明のために最大の「依存鎖 (dependency chain)」の長さを考慮することができます。仮説の鎖の長を 0 とし、量化された事実を具体化したものの鎖長をその事実の長さより、1 だけ大きいものと定義します。タクティク *inster n* は、鎖の長さの最大 *n* で、可能なすべての証明を試すことを意図しています。

```
Ltac inster n :=
  intuition;
    match n with
      | S ?n' ⇒
        match goal with
          | [ H : ∀ x : ?T, _, y : ?T ⊢ _ ] ⇒ generalize (H y); inster n'
        end
    end.
```

このタクティクは、命題の単純化を適用することから始まります。次に、鎖長が残っているかどうかをチェックし、そうでない場合は失敗します。それ以外の場合は、適切に型付けされたローカル変数を使用して量化された仮説を具体化するすべての可能な方法を試行します。再帰呼び出し *inster n'* が失敗した場合、`match goal` は、そのパターンを証明の状態とユニファイする別の方法を探していることに気付くことが重要です。したがって、この少量のコードは、バックトラック *mactch* がどのように徹底的な検索を可能にするかについてのエレガントなデモンストレーションを提供します。

ふたつの短い例で *inster* の有効性を検証することができます。組み込みの `firstorder` タクティク（余計な引数なし）は、最初のものを証明することができますが、2 番目のもの

は証明できません。

```
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, P (g x x) → 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.
```

inster の定義に採用されているスタイルは、関数的なプログラマーには直観に反するように見えます。通常、関数プログラムは、明示的な状態変化を再帰関数にの引数に蓄積します。Ltac では、現在のサブゴールの状態は常に暗黙的です。それにもかかわらず、一般的な命令型プログラミングとは対照的に、この前の節最後の議論を思い出すなら、この状態への変更を元に戻すのは簡単です。実際、このような「undo」は、`match` 内の失敗で自動的に起こります。このように、Ltac プログラミングは、`first` タクティカルの行に沿って合成演算子 (composition operator) をサポートするステートフルな failure モナドを持つ Haskell のプログラミングと似ています。

関数プログラミングの純粋主義者は、このようなプログラミングの提案に対して憤慨して反応するかもしれません。それにもかかわらず、他の種類の「モナド・プログラミング」と同様に、多くの問題が ML や Haskell での明示的で純粋な証明操作よりも、Ltac で解決するのがはるかに簡単です。実証するために、論理的含意いのための基本的な簡略化手順を書くことにします。

この手順は、分離論理 [36] に影響を受けています。ここでは、式のなかの連言 (conjuncts in formulas) は「リソース」と見なされ、含意の両辺で等しい連言を「交差させる」ことによって完全性を失うことはありません。このプロセスは、モジュール性の理由から、式が任意の入れ子ツリー構造（連言での分岐）を持つことができ、存在量化子を含むことができるという事実によって複雑になります。これは、マッチングプロセスが量化子を「下にいく」ことに役立ち、実際に、存在量化子をどのように具体化するかを決定するのに役立ちます。

我々のタクティクが扱う含意をさまざまな補題で「配管」として現れる含意から区別するために、ラッパー定義、ノーテーション、およびタクティクを定義します。

```
Definition imp (P1 P2 : Prop) := P1 → P2.
Infix "-->" := imp (no associativity, at level 95).
Ltac imp := unfold imp; firstorder.
```

imp に関するこれらの補題は、我々が書くタクティクに役立つでしょう。

```
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 pick_prem1 : ∀ P Q R S,
  (P ∧ (Q ∧ R) -> S)
  → ((P ∧ Q) ∧ R -> S).
  imp.
Qed.

Theorem pick_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 pick_conc1 : ∀ P Q R S,
  (S -> P ∧ (Q ∧ R))
  → (S -> (P ∧ Q) ∧ R).
  imp.
Qed.

Theorem pick_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.
```

私たちの *matcher* タクティクを作る上での最初の作業 (the first order of business) は、式の木を検索するための補助的なサポートになります。*search_prem* タクティクは imp の前提のすべての部分式 (subformula) で、引数 *tac* を実行するように実装されます。木を走査するとき、*search_prem* は上記の補題のいくつかを適用して、異なる部分式 (subformula) をゴールの頭部にもっていくことで、ゴールを書き換えます。

すなわち、その前提が、いくつかの Q に対して、$P \wedge Q$ のかたちに再配置されるところで、含意の前提のそれぞれの部分式 P に対して、P を 「have a turn」しようとします。

タクティク *tac* はこのかたちでゴールを見ることを期待し、前提の最初の連言に注意を集中するでしょう。

```
Ltac search_prem tac :=
  let rec search P :=
    tac
    || (apply and_True_prem; tac)
    || match P with
         | ?P1 ∧ ?P2 ⇒
           (apply pick_prem1; search P1)
           || (apply pick_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.
```

search_prem がどのように機能するかを理解するために、最初に、最後の match に進みます。前提が連言で始まる場合は、各連言で *search* 手続きを呼び出します。*search* P の呼び出しは、ある Q のための前提が $P \wedge Q$ の形式であるという不変性を期待し維持します。

tac が常に失敗したときに永久にループするのを避けるために、一種の再帰の尺度として減少する、P を明示的に渡します。

2 番目の match の場合は、制御を *search* に渡す前に、この不変量を実現するために可換性 (commutativity) の補題を呼び出します。

最終的な match の場合では *tac* を直接適用しようとしますが、それが失敗すると、余分な **True** 結合を追加してゴールの形を変え、*tac* をもう一度呼び出します。現在のサブゴールを変更せずに引数のタクティクが成功した場合、progress タクティカルは失敗します。

search 関数自体は最後の match の最後の場合と同じトリックを試みますが、||演算子を使ってひとつの手法を試してみます。さらに、いずれも動作しない場合、P が連言であるかどうかをチェックします。そうであれば、それはそれぞれの連言に対して再帰的に自分自身を呼び出し、最初に 結合性 (associativity) と 可換性の補題を適用してゴールの式の不変性を維持します。

また、imp の結論を通して木の検索を行う双対 (dual) 関数 *search_conc* が必要です。 *)

```
Ltac search_conc tac :=
  let rec search P :=
    tac
```

```
    || (apply and_True_conc; tac)
    || match P with
         | ?P1 ∧ ?P2 ⇒
           (apply pick_conc1; search P1)
           || (apply pick_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.
```

ここでは、検索戦略によってアプリケーションに適したいくつかの補題を証明することができます。前提を扱う補題は、P に関心をもった $P \land Q \to R$ の形式でなければならず、結論を扱う補題は、Q に関心をもった P –> $Q \land R$ の形式でなければなりません。

```
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.
```

取り消しが結論のすべての構成要素を取り除いた場合の証明を完成させるための「基本ケース」補題も必要です。

```
Theorem imp_True : ∀ P,
```

```
  P -> True.
  imp.
Qed.
```

今や、最終的な *matcher* タクティクは簡単です。

最初に、すべての変数をスコープに `intro` し、単純な前提の単純化を試み、**False** を見つけたら証明を完成させ、見つかった存在量化子を取り除きます。

その後、結論を検索します。

True の連言を取り除き、ユニフィケーション変数を導入することによって、束縛変数のための存在量化子を取り除き、一致する前提を探して取り除きます。

最後に、それ以上の進展がなければ、ゴールが自明かどうかをみて、imp_True によって解くことができます。それぞれの場合において、簡単にするために、`apply` の代わりに、より安価なユニフィケーション・アルゴリズムである `simple apply` タクティクを使います。

```
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.
```

私たちの戦術は簡単な例を証明するのに成功します。

```
Theorem t2 : ∀ P Q : Prop,
  Q ∧ (P ∧ False) ∧ P -> P ∧ Q.
  matcher.
Qed.
```

生成された証明では、検索手法の動作の痕跡が見つかります。

```
Print t2.
```

```
t2 =
fun P Q : Prop ⇒
comm_prem (pick_prem1 (pick_prem2 (False_prem (P:=P ∧ P ∧ Q) (P ∧ Q))))
     : ∀ P Q : Prop, Q ∧ (P ∧ False) ∧ P -> P ∧ Q
```

自動化が完了した後に人間の介入が必要な場合に *matcher* が適していることがわかります。

```
Theorem t3 : ∀ P Q R : Prop,
  P ∧ Q -> Q ∧ R ∧ P.
  matcher.
```

```
  ============================
   True -> R
```

私たちのタクティクは、それが取り消すことができたそれらの連言を取り消し、intuition と同じように、私たちのための簡略化されたサブゴールを残しました。 *)

```
Abort.
```

matcher タクティクは、量化の具体化を推測することにも成功します。実際の作業を行う Match 補題の使用にあるのはユニフィケーションです。

```
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 ⇒
      pick_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)
```

この証明の項はひと口で、手作業で構築していないのはうれしいことです！

13.5 ユニフィケーション変数の生成

タクティクを作成するための最後の有用な要素は、新しいユニフィケーション変数を明示的に割り当てることです。eauto のような戦術は、柔軟な証明検索をサポートするためにユニフィケーション変数を内部的に導入しています。eauto とその親戚は **後ろ向き** の推論をしていますが、ユニフィケーション変数が同様の理由で有用であるとき、しばしば同様に **前向き** 推論をしたいからです。

例えば、全称定化された仮説の定化子を具体化するタクティクを書くことができます。タクティクは、適切な具体化が何であるかを知る必要はありません。むしろ、これらの選択肢は代用品 (placeholder) で満たされています。後で、特化するための (specialized) 仮説を適用すると、構文的なユニフィケーションによって、具体的な値が決まることを願っています。

タクティクを書く準備が整う前に、一度にひとつずつ材料を試すことができます。

```
Theorem t5 : (∀ x : nat, S x > x) → 2 > 1.
  intros.
```

$H : \forall\ x :$ **nat**, $S\ x > x$

```
============================
 2 > 1
```

一般的に、H を具体化するには、最初に x に使用する値の名前を付ける必要があります。

```
evar (y : nat).
```

```
H : ∀ x : nat, S x > x
y := ?279 : nat
============================
 2 > 1
```

証明コンテキストは新しい変数 y で拡張され、新しいユニフィケーション変数 ?279 と等しくなるように割り当てられています。?279 と H を具体化する必要があります。エイリアス y だけではなく、新しいユニフィケーション変数を取得するために、Ltac の `eval` 要素使用して式 y で自明な展開 (unfolding) を実行します。タクティク（例えば、`simpl`、`compute` など）で見られたのと同じ簡約戦略 (reduction strategies) です。

```
let y' := eval unfold y in y in
  clear y; specialize (H y').
```

```
H : S ?279 > ?279
============================
  2 > 1
```

具体化は成功しました。`apply` のユニフィケーションを使って ?279 の固有な値を計算することで証明を終えることができます。

```
  apply H.
Qed.
```

今使っているパターンをカプセル化して、特定の仮説のすべての量化子を具体化する方法を書くことができます。

```
Ltac insterU H :=
  repeat match type of H with
           | ∀ x : ?T, _ ⇒
             let x := fresh "x" in
               evar (x : T);
               let x' := eval unfold x in x in
                 clear x; specialize (H x')
         end.

Theorem t5' : (∀ x : nat, S x > x) → 2 > 1.
  intro H; insterU H; apply H.
```

```
Qed.
```

この特定の例は、apply 自体が本来の目的を達成していたので、やや馬鹿げています。別の前方推論は、存在量化の定量化で終わる仮説により有用です。例を見る前に、私たちが渡すベースとなる仮説を除去 (clear) していない *insterU* の変種の Ltac の構成要素 fresh を使用して、(引数の) 文字列で示唆されるよい名前に基づいた、まだ使用されていない仮説名を生成します。

```
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.
```

eauto も firstorder もこのゴールを証明するほどには賢くはありません。量化子を使っていくつかの作業を自分で行い、do タクティカルによって、一定回数のタクティクを繰り返すことで、証明を省略します。

```
    do 2 insterKeep H1.
```

証明状態は *H1* のふたつの一般的なインスタンスで拡張されています。

```
  H' : ∃ u : B, P ?4289 u
  H'0 : ∃ u : B, P ?4288 u
  ============================
   ∃ u1 : B, ∃ u2 : B, P (f v1 v2) (g u1 u2)
```

通常の eauto はまだ目標を証明することができませんので、ふたつの新しい存在量化子を削除します。（**ex** は、∃ 構文の使用がコンパイルされるもとになる型ファミリであることを思い出してください）

```
    repeat match goal with
             | [ H : ex _ ⊢ _ ] ⇒ destruct H
           end.
```

ゴールは、論理プログラミングで解くには、とても単純です。　　eauto.

```
  Qed.
End t6.
```

insterU タクティクは、含意も含む量化仮説ではあまりうまくいきません。最後の例をわずかに変更して問題を見ることができます。新しい単項の述語 Q を導入し、仮説 *H1* の追加の要求を表示するのに使います。

```
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 t7 : ∀ 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.

No more subgoals but non-instantiated existential variables :
Existential 1 =

?4384 : [A : Type
         B : Type
         Q : A → Prop
         P : A → B → Prop
         f : A → A → A
         g : B → B → B
         H1 : ∀ v : A, Q v → ∃ u : B, P v u
         H2 : ∀ (v1 : A) (u1 : B) (v2 : A) (u2 : B),
                P v1 u1 → P v2 u2 → P (f v1 v2) (g u1 u2)
         v1 : A
         v2 : A
         H : Q v1
         H0 : Q v2
         H' : Q v2 → ∃ u : B, P v2 u ⊢ Q v2]
```

別の存在変数 (existential variable) についても同様の行があります。ここで、「存在変数」とは、「ユニフィケーション変数」とも呼ばれるものを意味します。証明の過程で、ユニフィケーション変数?4384 が導入されましたが、ユニフィケーションされませんでした。ユニフィ

ケーション変数は、証明検索を構造化するための単なる器 (device) です。Gallina 言語の証明項にはそれらが含まれていません。したがって、変数を具体化せずに証明項を生成することはできません。

エラーメッセージは、?4384 が変数と仮説が表示されている特定の証明状態の *Q v2* の証明であることを示しています。それは?4384 が *H1* に渡す証明の値として *insterU* によって作成されたことが判明しました。Gallina では、含意は単に ∀ 量化の縮退したかたちであるため、∀ と一致させる *insterU* コードもこの含意と一致したことを思い出してください。この文脈で *Q v2* の証明は他の証明と同じくらい良いので、どの証明が適切かを正確に判断するためにユニフィケーションを使う機会は決してありません。同様に *insterU* の引数にある含意の問題を予期しています。

```
  Abort.
End t7.

Reset insterU.
```

∀ *x* : ?*T*, ... の型 *T* の型をパターンマッチさせる *T* に型 Prop、*x* の具体化は証明のために考える必要があります。したがって、新しい統一変数を選択するのではなく、ユーザーが指定した戦術 *tac* を適用します。*tac* が *T* を証明するのに失敗した場合、

デフォルトの量化の処理を続けるのではなく、具体化を中止します。また、が目標を完全に解決しない場合、solve [*t*] は失敗します。*)

```
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) by solve [ tac ];
                     specialize (H H'); clear H')
                 || fail 1
               | _ ⇒
                 let x := fresh "x" in
                   evar (x : T);
                   let x' := eval unfold x in x in
                     clear x; specialize (H x')
             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 t7 : ∀ v1 v2, Q v1 → Q v2 → ∃ u1, ∃ u2, P (f v1 v2) (g u1 u2).
```

まだ P の事実を知らない変数について Q 仮説を見つけて適用しようとするタクティクで *insterKeep* を呼び出すことによってゴールを証明することができます。

Coq の証明エンジンで、ファーストクラスのタクティクの引数が、match で始まらなければならないという奇妙な制限を回避するために、このタクティクのコードを idtac; で始める必要があります。

```
    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.
```

存在変数 (existential variables) を明示的に具体化することはしばしば役に立ちます。組み込みタクティクはそれをするひとつの方法を提供します。

```
Theorem t8 : ∃ p : nat × nat, fst p = 3.
  econstructor; instantiate (1 := (3, 2)); reflexivity.
Qed.
```

上記の 1 は、現在のゴールに現れる存在変数を特定するもので、最後に存在するものが番号 1 に割り当てられ、2 番目に割り当てられた番号 2 が割り当てられます。名前のついた存在（変数）は、どこでも:=の右側の項に置き換えられます。

instantiate タクティクは探索的な証明 (exploratory proving) のためには便利かもしれませんが、変化する定理には適応しにくい非常に脆い証明スクリプトにつながります。

実在すると判っている項に値を割り当てるために使用できるタクティクを持つことは、しばしば有用です。婉曲な (roundabout) 実装技術を採用することで、この機能を一般化するタクティクを作ることができます。特に、我々の戦術 *equate* は、2 つの項が等しいことを主張します。項のひとつが存在していれば、それはどこにでも置き換えられます。

```
Ltac equate x y :=
```

```
  let dummy := constr:(eq_refl x : x = y) in idtac.
```

eq_refl で $x = y$ を証明することができない場合、この方法は失敗します。関連する変数 *dummy* を無視して、ユニフィケーションの副作用のみをチェックします。*equate* によって先の例の脆弱なバージョンを作ることができます。

```
Theorem t9 : ∃ p : nat × nat, fst p = 3.
  econstructor; match goal with
                  | [ ⊢ fst ?x = 3 ] ⇒ equate x (3, 2)
                end; reflexivity.
Qed.
```

このテクニックは、広範囲のゴールを解決するための再帰な繰り返しのあるタクティクの中でさらに役立ちます。

Chapter 14

リフレクションによる証明

最後の章では、証明に対する非常にヒューリスティックなアプローチが強調されていました。この章では、代替の技術、_proof by reflection **リフレクションによる証明** [2] を検討します。Gallina によって、決定手続き (decision procedures、決定するための手続き) を正しさの証明とともに書いて、これらの手続きが非常に短い証明で書けることを示します (appeal)。

項の *reflection* **リフレクション** は、Gallina の命題を構文が表す帰納型の値に変換する必要があるために、適用されます。それで、Gallina プログラムがそれらを解析し、そのような項を元の形式に戻すことを **リフレクトする** (*reflecting* it) と呼びます。

14.1 偶数であることの証明

特定の自然数の定数が偶数であることを証明することは、確かに自動的にできることです。前章で学習した Ltac プログラミング手法は、このような手順を簡単に実装できます。

```
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
```

... 等々。この手順は常に（少なくとも無限のリソースを持つマシンでは）機能しますが、重大な欠点があります。これは、256 が偶数であるという証明を印刷 (print) するときに表示さ

れます。最終的な証明項は、入力値に対して長さが超線形 (super-linear) です。Coq の暗黙的な引数の仕組みは、Even_SS のパラメータ n に与えられた値を隠しているので、ここでは証明項はここでは線形にしか見えません。

また、証明項は内部的に構文木として表現され、ノード表現を共有する機会がありますが、この章では、ほぼ同等の 2 つの尺度、証明文の長さを簡単なテキストの長さとして、または、用語の構文木のノード数として測定します。

時には明らかに大きな証明項でも、内部では十分な共有を持っているため、予想以上のメモリを消費することはありませんが、共有のない (sharing-free) バージョンのサイズが十分に小さいことを保証することによって、そのような共有を推論する必要はありません。

定数の偶数性を検証するための自明で信頼できるプログラムを書くことができるので、超線形 (Superlinear) な偶数性の証明項は残念なものに見えます。証明チェッカーは、必要に応じて私たちのプログラムを単に呼び出すことができます。

静的な型付けが、私たちのタクティクが常に適切に動作することを保証しないことも残念です。同様の手法の他の呼び出しは、動的な型エラーで失敗する可能性があり、十分に複雑なゴールを証明しようとするまで、これらのエラーの背後にあるバグについてはわかりません。

リフレクションによる証明の手法は、両方の不満に対処します。入力のサイズを超えて一定のサイズ (constant size) のオーバーヘッドを持つ上記の例のような証明を書くことができ、私たちは Gallina で書かれた検証された決定手続き (decision procedures) でそれを行います。

この例では、MoreSpecif モジュール（この本のソースに含まれています）の型を使用して、認証された偶数性のチェッカーを作成します。

Print **partial**.

Inductive **partial** (P : Prop) : Set := Proved : $P \rightarrow [P]$ | Uncertain : $[P]$

partial P の値は、P のオプショナルな証明です。 表記 $[P]$ は **partial** P の略です。

Local Open Scope *partial_scope*.

私たちは **partial** 型の表記をいくつか取り上げます。これらは、以前に、型の記述 (specification types) で見た表記法のいくつかと重複しているため、別々に開くことが必要な別のスコープに配置されました。

```
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.
```

これで、依存パターンマッチング (dependent pattern-matching) を使用して、驚くべきことを行う関数を書くことができます。*partialPut* が与えられている場合、partialOut は **partial**

値に証明が含まれていれば P の証明を返し、それ以外の場合は **True** の（使用しない useless) の証明を返します。ML と Haskell のプログラミングの観点からは、このような型を書くことは不可能ですが、return アノテーションを使用すれば自明です。

```
Definition partialOut (P : Prop) (x : [P]) :=
  match x return (match x with
                    | Proved _ ⇒ P
                    | Uncertain ⇒ True
                  end) with
    | Proved pf ⇒ pf
    | Uncertain ⇒ I
  end.
```

このような関数を定義するのは奇妙に思えるかもしれませんが、以前の *prove_even* タクティクのリフレクティブのバージョンを書くのに非常に便利です：*)

```
Ltac prove_even_reflective :=
  match goal with
    | [ ⊢ isEven ?N] ⇒ exact (partialOut (check_even N))
  end.
```

どの自然数を考えているのかを決め、適切な check_even の呼び出しから証明を引き出すことによって、その偶数性を「証明」します。正確に型 P の証明項が与えられたとき、exact タクティクは、命題 P を証明します。

```
Theorem even_256' : isEven 256.
  prove_even_reflective.
Qed.

Print even_256'.

even_256' = partialOut (check_even 256)
     : isEven 256
```

私たちは、証明のゴールの周りに一定のラッパーを見ることができます。偶数の場合は、この形式の証明で十分です。証明項のサイズは、チェックされている数では線形で、

その数の単項式の 2 つの繰り返しが含まれています。そのうちのひとつは partialOut の暗黙の引数の上に隠されています。

奇数でこのタクティクを試してみたらどうなりますか？

```
Theorem even_255 : isEven 255.
  prove_even_reflective.

User error: No matching clauses for match goal
```

ありがたいことに、タクティクは失敗します。間違ったことをより正確に見るために、match の本体を手動で実行することができます。

```
  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"
```

いつものように、型チェッカーはエラーメッセージを単純化するためにリダクション (式の簡約) を実行しません。最初の項を自分自身で簡約した場合、check_even 255 は No に簡約され、最初の項は **True** と等しく、**isEven** 255 とユニファイしないのは確かです。

```
Abort.
```

タクティク *prove_even_reflective* は、リフレクティブです。なぜなら、Gallina 内の証明検索のプロセス（この場合は自明なもの）を実行するために、Ltac の唯一の使用は目標を check_even の適切な使用に変換するためです。

14.2 自明なトートロジーの言語の文法の具象化 (reifying)

このような自明なトートロジーのリフレクティブな証明をしたいかもしれません：

```
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)
```

予想されるように、tauto が作る証拠は、自然演繹規則の明示的な応用が含まれています。大きな式の場合、これは、入力のサイズを超える、証明のサイズの線形量のオーバーヘッドを追加する可能性があります。

このクラスのゴールに対してリフレクティブな手順を書くには、「リフレクティブによる証明」の実際の「リフレクティブ」な部分に入る必要があります。Gallina のいかなるところでも、Prop を条件分析 (case-analyze) することは不可能です。Prop を *reify* **具象化** して、解析できる型にする必要があります。この帰納型は良い候補です：

```
Inductive taut : Set :=
| TautTrue : taut
| TautAnd : taut → taut → taut
| TautOr : taut → taut → taut
| TautImp : taut → taut → taut.
```

この構文を *reflect* **リフレクト** して Prop に戻すための再帰関数を書きます。このような関数は *interpretation functions* **翻訳関数**とも呼ばれ、前の例で、小さなプログラミング言語に意味論を与えるために使用しています。

```
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.
```

tautDenote の範囲のすべての式が真であることを証明するのは簡単です。

```
Theorem tautTrue : ∀ t, tautDenote t.
  induction t; crush.
Qed.
```

特定の式を証明するために tautTrue を使用するには、その構文を具象化する手順 (reification process) を実装する必要があります。再帰的な Ltac 関数がその仕事をします。

```
Ltac tautReify P :=
  match P with
    | True ⇒ TautTrue
    | ?P1 ∧ ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautAnd t1 t2)
    | ?P1 ∨ ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautOr t1 t2)
    | ?P1 → ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautImp t1 t2)
  end.
```

tautReify が利用可能なので、リフレクティブなタクティクを終わらせるのは簡単です。ゴールの式を見て、それを具象化 (reify) し、その具象化した式に tautTrue を適用します。

```
Ltac obvious :=
  match goal with
    | [ ⊢ ?P ] ⇒
      let t := tautReify P in
        exact (tautTrue t)
  end.
```

証明項の詳細は言及せずに、*obvious* がもとの例を解くことを証明することができます。

```
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)
```

純粋な Ltac による実装で、リフレクティブなタクティクがどのように改善するかを考える価値があります。式の具象化の手順は以前と同じようにアドホックなので、そこではほとんど利益を得ることはできません。一般に、証明は数式の変換よりも複雑になります。ここで適用する「一般的な証明の規則 (generic proof rule)」は、再帰的な Ltac 関数よりもはるかに優れた正式な基礎に当てはまります。依存型の証明は、どの入力式でも「機能する」ことを保証します。この利点は、すでに見てきた、証明のサイズの改善に加えられます。

以前の例の偶数性のテストでは、検証された決定手続きが証明できない入力ゴールを健全に処理する (sound handling) ために関数 partialOut を使用していたことを指摘する価値があります。ここで、手続き tautTrue（帰納的証明は再帰的手続きと見なすことができることを思い出してください）は、**taut** で表現できるどんなゴールをも証明できるので、余分なステップは必要ありません。

14.3 モノイド式の簡略化

リフレクションによる証明では、ゴールの中のすべての構文をエンコードする必要はありません。特殊な推論を適用できない場合でも、任意の断片が注入 (injection) を許すために、構文の型に「変数」を挿入することができます。その可能性を、モノイドの等式 (monoid equations) を正規化するためのタクティクを書くことによって探求します。

```
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.
```

モノイドの代数構造の任意のインスタンスを特徴付ける変数と仮説を追加します。結合法則を満たす (associative)2 項演算子とそれに対する単位元があります。

モノイドの式についての木の型の式を定義するのは簡単です。Var コンストラクタは、私たちがモデル化できない部分式の「キャッチ・オール」ケースです。これらの部分式は、実際の Gallina 変数であるべきで、また、私たちのタクティクが理解できない関数を使うことができます。

```
  Inductive mexp : Set :=
```

```
| Ident : mexp
| Var : A → mexp
| Op : mexp → mexp → mexp.
```

次に、翻訳関数 (interpretation function) を書きます。

```
Fixpoint mdenote (me : mexp) : A :=
  match me with
    | Ident ⇒ e
    | Var v ⇒ v
    | Op me1 me2 ⇒ mdenote me1 + mdenote me2
  end.
```

結合法則を使って (via associativity) 式をリストにフラット化 (flattening) することで式を正規化するので、モノイドの値のリストのための表示関数 (denotation function) があると便利です。

```
Fixpoint mldenote (ls : list A) : A :=
  match ls with
    | nil ⇒ e
    | x :: ls' ⇒ x + mldenote ls'
  end.
```

フラット化関数は、それ自体、実装するのに簡単です。

```
Fixpoint flatten (me : mexp) : list A :=
  match me with
    | Ident ⇒ nil
    | Var x ⇒ x :: nil
    | Op me1 me2 ⇒ flatten me1 ++ flatten me2
  end.
```

この関数は、私たちの 表示 (*denote*) 関数に関して簡単な正当性証明を持っています。

```
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.
```

簡約化タクティクの主な道具となる定理を証明するのは簡単です。

```
Theorem monoid_reflect : ∀ me1 me2,
  mldenote (flatten me1) = mldenote (flatten me2)
  → mdenote me1 = mdenote me2.
  intros; repeat rewrite flatten_correct; assumption.
```

Qed.

mexp 型への具象化を実装します。

```
Ltac reify me :=
  match me with
    | e ⇒ Ident
    | ?me1 + ?me2 ⇒
      let r1 := reify me1 in
      let r2 := reify me2 in
        constr:(Op r1 r2)
    | _ ⇒ constr:(Var me)
  end.
```

最終的な monoid タクティクは、モノイドの項のふたつからなる等式のゴールに作用します。それぞれを具象化 (reify) し、monoid_reflect を適用し、mldenote 使って簡単化することで、ゴールが具現化したバージョンを参照するように変更します。tactics!changechange タクティクは、結論の式を定義として等しいものに置き換えることを思い出してください。

```
Ltac monoid :=
  match goal with
    | [ ⊢ ?me1 = ?me2 ] ⇒
      let r1 := reify me1 in
      let r2 := reify me2 in
        change (mdenote r1 = mdenote r2);
          apply monoid_reflect; simpl
  end.
```

このような定理の短い作業をすることができます：

```
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 タクティクは、等式の両辺を正準化 (canonicalized) しているので、反射性 (reflexivity) によって証明を完成させることができます。　`reflexivity.`

Qed.

証明の形式 (the form) を見るのは面白いです。

```
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))
```

$(eq\_refl\ (a + (b + (c + (d + e)))))$
$\quad : \forall\ a\ b\ c\ d : A,\ a + b + c + d = a + (b + c) + d$

証明項は、具現化された形式の等式のオペランドの再記述だけを含み、共有された (shared) 正準形 (canonical form) への反射性の使用が続きます。

```
End monoid.
```

この基本的なアプローチの拡張は、Coq でパッケージ化された、`ring` と `field` タクティクの実装で使われています。

14.4 賢いトートロジー・ソルバー

これで、以前のトートロジー・ソルバーの例を再検討する準備が整いました。私たちは、真理が構文的に明らかでない式を含むようにタクティクの範囲を広げたいと考えています。最後の例で任意のモノイド式を許可したのと同じように、任意の式の注入 (injection) を許可したいと思うでしょう。私たちはより豊かな理論で作業しているので、注入された異なる式の間で等値を使用できることが重要です。例えば、Gallina 関数は、ふたつの P が等しいかどうかを比較する方法がないため、式を Imp (Var P) (Var P) のような値に変換することによって $P \rightarrow P$ を証明することはできません。

これらの基準を満たす素晴らしい実装に到達するために、`quote` タクティクとそれに関連するライブラリを紹介します。

```
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.
```

`quote` タクティクは `Prop` から *Formula* への単射 (injection) を実装しますが、望むほどスマートではありません。特に、関数型を特別に扱いたいので、関数型が構文的に符号化したい構造体の一部であれば混乱 (confused) します。論理的な意味を表現するために関数型を使用することに気づかないように `quote` することをトリックするには、最後の章で行ったように、含意のラッパー定義を宣言する必要があります。

```
Definition imp (P1 P2 : Prop) := P1 → P2.
Infix "->" := imp (no associativity, at level 95).
```

ここでは、表示関数 (denotation function) を定義することができます。

```
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.
```

varmap のファミリーは、**index** の値から map を実装します。この場合、変数から Prop への map として代入を定義します。翻訳関数 (interpretation function) formulaDenote は代入として動作し、varmap_find 関数を使って Atomic の場合の代入を調べます。varmap_find の最初の引数は、変数が見つからない場合のデフォルト値です。

```
Section my_tauto.
  Variable atomics : asgn.

  Definition holds (v : index) := varmap_find False v atomics.
```

特定の変数が真であるためにいくつかの省略形を定義し、標準ライブラリの ListSet モジュールに基づいて、（驚くことではありませんが）リストを集合として見ることを提供する、便利な関数を定義する準備が整いました。

```
  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.
```

index の集合のすべてのメンバーに対して真となる命題を表わすものを定義し、この表記についてのいくつかの補題を証明します。

```
  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.
```

ここで、仮説の分解 (deconstruction) を実装する関数 forward を書くことができ、複合式を、Or を使って導入された可能性のあるすべての場合 (case) をカバーする一連のアトミックな論理式の集合に展開することができます。

複数のケースの考慮を処理するために、この関数は継続引数 (continuation argument) を取ります。これはそれぞれのケースに対して 1 回呼び出されます。

forward 関数は、第 6 章のスタイルで依存性の型を持ち、正確さを保証します。forward のへの引数は、ゴールの式 f、真であると仮定してもよい原子式の集合 *known*、仮定の式 *hyp*、および、*hyp* によって含意される新しい真実を保持するために *known* を拡張するときに呼び出す次への継続 (success continuation) *cont* です。

```
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.
```

backward 関数は、最終のゴールの分析を実装します。含意を扱うために forward を呼び出

します。

```
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.
```

backward の周りの単純なラッパーは、部分的な決定のための手続きの通常のタイプを与えます。

```
Definition my_tauto : ∀ f : formula, [formulaDenote atomics f].
  intro; refine (Reduce (backward nil f)); crush.
Defined.
End my_tauto.
```

最終的なタクティクの実装はかなり簡単です。まず、Prop の束縛しないすべての量化子を intro します。次に、具象化を実装する quote タクティクを呼び出します。最後に、partialOut と my_tauto の Gallina 関数を使用して正確な証明を構築します。

```
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.
```

いくつかの例は、タクティクがどのように機能するかを示しています。

```
Theorem mt1 : True.
  my_tauto.
Qed.

Print mt1.
```

mt1 = partialOut (my_tauto (Empty_vm Prop) Truth)
  : **True**

すべての部分式は formulaDenote によって処理されるため、my_tauto に空の **varmap** が適用されていることがわかります。

```
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
```

重要なことに、$x = y$ の両方のインスタンスは同じインデックス End_idx で表されます。このインデックスの値は **varmap** に 1 回だけ現れなければならず、**varmap** は、**index** の値が木の根から葉までのパスを示す、二進木 (binary trees) として表現されます。

```
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)
```

ゴールにはみっつの異なる原子式が含まれていて、3 要素の「varmap」が生成されていることがわかります。

特に自明な定理のために my_tauto と tauto によって生成される証明項の反復のレベルの違いを観察することは面白いかもしれません。

```
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
```

伝統的な tauto タクティクは、証明項のサイズの 2 次 (quadratic) 的な「爆発 (blow-up)」を導入するのに対し、my_tauto によって生成される証明は常に線形の大きさを持ちます。

14.4.1 変数を持つ項の手動の具象化

上記の quote タクティクの動作は手品のように見えます。どういうわけか、それは任意の型の副項 (subterms) どうしが等しいかどうかの比較 (equality comparison) を実行するので、これらの副項は同じ具象化された (reified) 変数でされるでしょう。

quote は OCaml で実装されていますが、具象化のプロセス (reification process) を Ltac で完全にコードすることができます。仕事をより簡単にするために、変数を **nat** として表現し、参照される変数の値からなる単純なリストにインデックスを付けます。

プロセスの最初のステップは、項をクロールし、エンコードする位置に表示されるすべての値の重複のないリストを変数として構築することです。便利な補助関数 (helper function) は、重複を防止ながら、要素をリストに追加ます。Ltac のパターンマッチングを使用して Gallina 項についての等値のテスト (equality test) を実装する方法に注意してください。これは単純な文法的な等値 (syntactic equality) であり、より豊かな定義による等値 (definitional equality) ではありません。また、異なるリストの要素が異なる Gallina に型を持つことを可能にするために、ネストされたタプルとしてリストを表現します。

```
Ltac inList x xs :=
  match xs with
    | tt ⇒ false
    | (x, _) ⇒ true
    | (_, ?xs') ⇒ inList x xs'
  end.

Ltac addToList x xs :=
  let b := inList x xs in
    match b with
      | true ⇒ xs
      | false ⇒ constr:(x, xs)
    end.
```

ここで、項を表すために使用する変数値のリストを計算する再帰関数を書くことができます。

```
Ltac allVars xs e :=
  match e with
    | True ⇒ xs
    | False ⇒ xs
    | ?e1 ∧ ?e2 ⇒
      let xs := allVars xs e1 in
        allVars xs e2
    | ?e1 ∨ ?e2 ⇒
      let xs := allVars xs e1 in
        allVars xs e2
    | ?e1 → ?e2 ⇒
      let xs := allVars xs e1 in
        allVars xs e2
    | _ ⇒ addToList e xs
  end.
```

値をリストの内の位置にマップする方法も必要です。

```
Ltac lookup x xs :=
```

```
  match xs with
    | (x, _) ⇒ O
    | (_, ?xs') ⇒
      let n := lookup x xs' in
        constr:(S n)
  end.
```

次の構成要素 (building block) は、許可されたすべての変数値のリストを指定して、項を具象化する手順です。

変数のリストに含まれていない副項 (subterm) を含む項の具象化を試みると、タクティクの失敗が引き起こされる可能性があるため、この手続きは部分的 (partial) になります (free to make this procedure partial)。出力項の型は、**index** が **nat** に置き換えられた **formula** のコピーで、アトミックな式のコンストラクタの型です。

```
Inductive formula' : Set :=
| Atomic' : nat → formula'
| Truth' : formula'
| Falsehood' : formula'
| And' : formula' → formula' → formula'
| Or' : formula' → formula' → formula'
| Imp' : formula' → formula' → formula'.
```

私たち自身の Ltac 手続きを書くときは、ラッパーを導入する必要はなく、通常の → 演算子で直接作業することができます。

```
Ltac reifyTerm xs e :=
  match e with
    | True ⇒ constr:Truth'
    | False ⇒ constr:Falsehood'
    | ?e1 ∧ ?e2 ⇒
      let p1 := reifyTerm xs e1 in
      let p2 := reifyTerm xs e2 in
        constr:(And' p1 p2)
    | ?e1 ∨ ?e2 ⇒
      let p1 := reifyTerm xs e1 in
      let p2 := reifyTerm xs e2 in
        constr:(Or' p1 p2)
    | ?e1 → ?e2 ⇒
      let p1 := reifyTerm xs e1 in
      let p2 := reifyTerm xs e2 in
        constr:(Imp' p1 p2)
    | _ ⇒
      let n := lookup e xs in
        constr:(Atomic' n)
  end.
```

最後に、すべての作品をまとめています。 Ltac *reify* :=

```
match goal with
  | [ ⊢ ?G ] ⇒ let xs := allVars tt G in
    let p := reifyTerm xs G in
      pose p
end.
```

クイックテストは、私たちが正しく具象化 (reification) を行っていることを検証します。

```
Theorem mt3' : ∀ x y z,
  (x < y ∧ y > z) ∨ (y > z ∧ x < S y)
  → y > z ∧ (x < y ∨ x < S y).
  do 3 intro; reify.
```

私たちの簡単なタクティクは、変換された項を新しい変数として追加します：

```
f := Imp'
       (Or' (And' (Atomic' 2) (Atomic' 1)) (And' (Atomic' 1) (Atomic' 0)))
       (And' (Atomic' 1) (Or' (Atomic' 2) (Atomic' 0))) : formula'
```

```
Abort.
```

新しい構文型と式の実際の意味を結びつけなければならないので、リフレクティブなタクティクを完成させるためにはもっと多くの作業が必要ですが、詳細は以前の quote による実装と同じです。

14.5 束縛のもとで再帰する具象化タクティクを作る

これまでのすべての例では、量化子や、fun による関数抽象などの項の構文を具現化する形式から離れていました。そのような場合は、異なる副項が自由変数の異なる集合を参照することを許されるという事実によって複雑になります。このようなハードルを解消するためにはいくつかの巧妙さが必要ですが、すこしの単純なパターンで十分です。関数抽象の本体が便利に Coq の関数で表されている単純な依存型の言語の例を考えてみましょう。

```
Inductive type : Type :=
| Nat : type
| NatFunc : type → type.
```

```
Inductive term : type → Type :=
| Const : nat → term Nat
| Plus : term Nat → term Nat → term Nat
| Abs : ∀ t, (nat → term t) → term (NatFunc t).
```

```
Fixpoint typeDenote (t : type) : Type :=
  match t with
    | Nat ⇒ nat
    | NatFunc t ⇒ nat → typeDenote t
```

```
  end.
Fixpoint termDenote t (e : term t) : typeDenote t :=
  match e with
    | Const n ⇒ n
    | Plus e1 e2 ⇒ termDenote e1 + termDenote e2
    | Abs _ e1 ⇒ fun x ⇒ termDenote (e1 x)
  end.
```

ここでは、具象化のタクティクで素朴 (naïve) な最初の試みがあります。

```
Ltac refl' e :=
  match e with
    | ?E1 + ?E2 ⇒
      let r1 := refl' E1 in
      let r2 := refl' E2 in
        constr:(Plus r1 r2)

    | fun x : nat ⇒ ?E1 ⇒
      let r1 := refl' E1 in
        constr:(Abs (fun x ⇒ r1 x))

    | _ ⇒ constr:(Const e)
  end.
```

関数を一般的に扱うために、パターンの変数の形式 @?X を使用して、明示的に宣言された新しく導入された変数について X が言及するようにします。@?X の使用のあとには、言及する可能性のあるローカル変数のリストを続ける必要があります。

変数 X は、それらの変数の値に渡って Gallina 関数を表します。 例えば：

```
Reset refl'.
Ltac refl' e :=
  match e with
    | ?E1 + ?E2 ⇒
      let r1 := refl' E1 in
      let r2 := refl' E2 in
        constr:(Plus r1 r2)

    | fun x : nat ⇒ @?E1 x ⇒
      let r1 := refl' E1 in
        constr:(Abs r1)

    | _ ⇒ constr:(Const e)
  end.
```

抽象化の場合 $E1$ を、x の値から抽象化した本体の値への関数に、束縛します。残念ながら、再帰呼び出しは成功すると決まったわけではありません。

同じ抽象パターンに一致し、別の再帰呼び出しを実行するなど、無限再帰を使用します。最後の具象化によって作業手順が得られます。重要なアイデアは、*refl'* へのすべての入力を**再帰中に導入された変数の値に対する関数** として考えることです。

```
Reset refl'.
Ltac refl' e :=
  match eval simpl in e with
    | fun x : ?T ⇒ @?E1 x + @?E2 x ⇒
      let r1 := refl' E1 in
      let r2 := refl' E2 in
        constr:(fun x ⇒ Plus (r1 x) (r2 x))

    | fun (x : ?T) (y : nat) ⇒ @?E1 x y ⇒
      let r1 := refl' (fun p : T × nat ⇒ E1 (fst p) (snd p)) in
        constr:(fun u ⇒ Abs (fun v ⇒ r1 (u, v)))

    | _ ⇒ constr:(fun x ⇒ Const (e x))
  end.
```

$@?X$ のパターンでは、関数の形で加算の場合でもどのように動作するのか注意してください。抽象化の場合、自由変数を表すために使用される型を拡張することによって新しい変数が導入されます。すべての自由変数を表すために T 型を使用しました。抽象化の本体内の自由変数値を表す型の型を T × **nat** に拡張します。ペアとその射影 (projection) による少しのブックキーピング (a bit of bookkeeping) は、再帰呼び出しで渡す抽象本体の適切なバージョンを生成します。このような項の再パッケージングがすべてパターンマッチングを妨げないようにするために、*refl'* の本体の最初の行に関数引数を追加します。

今やもうひとつのタクティクが、具象化を適用する方法の例を提供します。具象化できる項どうしの等式であるゴールを考えてみましょう。このようなゴールを termDenote への適切な呼び出しの間で等式に変更したいと考えています。

```
Ltac refl :=
  match goal with
    | [ ⊢ ?E1 = ?E2 ] ⇒
      let E1' := refl' (fun _ : unit ⇒ E1) in
      let E2' := refl' (fun _ : unit ⇒ E2) in
        change (termDenote (E1' tt) = termDenote (E2' tt));
          cbv beta iota delta [fst snd]
  end.

Goal (fun (x y : nat) ⇒ x + y + 13) = (fun (_ z : nat) ⇒ z).
  refl.

  ============================
   termDenote
     (Abs
        (fun y : nat ⇒
```

Abs (fun $y0$: **nat** ⇒ Plus (Plus (Const y) (Const $y0$)) (Const 13)))) =
termDenote (Abs (fun _ : **nat** ⇒ Abs (fun $y0$: **nat** ⇒ Const $y0$)))

```
Abort.
```

ここでのエンコーディングは、Coq の関数を使用して、妥当な条件で束縛を表現しているため、特定の関数機能を実現するのが難しくなります。

また、変数を数値で表現する方法もあります。これは、項の引数が fst と snd の合成だけであることを検出することによって、変数参照を識別する若干スマートな具象化関数を書くことによって行うことができます。構成の順序から、変数の数値を読み取ることができます。読者のために詳細を練習問題（しかし、自明なものではありません！）として残しています。

Part IV

大局的

Chapter 15

大きな証明をする

「定理証明」という言葉が、「理論」という言葉によく似ていることは、やや残念です。ソフトウェアの研究者や実務家のほとんどは、機械化された定理証明が根本的に実用的でないと信じています。確かに、最近まで、高階論理について証明された定理の大部分の進歩は、主に理論的なものでした。しかし、21 世紀初頭から、重大な検証作業において証明支援系 (proof assistants) の使用が急増しました。その仕事の列線 (line) はまだ新しいものですが、私は大規模な形式的な証明で効果的に仕事をする方法についていくつかの教訓を掘り起こすのは時期尚早ではないと思います。したがって、この章では、大規模な Coq の開発を構造化し、保守するためのヒントを示します。

15.1 Ltac アンチパターン

この本では、「完全自動化」になるまで、証明が完了していないというある意味で、珍しいスタイルに従っています。そのような定理の各々は、単一ののタクティクによって証明されます。Ltac はチューリング完全なプログラミング言語なので、セミコロンのような演算子を使ってステップを組み合わせることで、任意のヒューリスティックスをひとつのタクティクに絞り込むことは難しくありません。対照的に、「野良 (in the wild)」での大部分の Ltac による証明は、個々のタクティクのピリオドに続いて実行される多くのステップで構成されています。セミコロンで終了した証明手順とピリオドで終了した手順を区別することは本当に価値があるでしょうか？

私は、これが実際には非常に重要な違いであり、重要な検証の領域の大多数に深刻な影響を及ぼしている、と主張しています。実証的な領域が関与する、より面白くない、退屈な (drudge) とした仕事ほど、単一のタクティクで定理を証明することがより重要になります。自動化の観点からは、単一のタクティクによる証明は非常に効果的であり、証明がより興味深い詳細によって埋められているので、自動化はますます重要になります。この節では、より一般的な証明のスタイルの結果のいくつかの例を示します。

実行中の例として、算術式の基本言語、そのためのインタプリタ、および、式のなかのすべての定数を拡大するトランスレータを考えてみましょう。

```
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.
```

実際に乗算 (times) を実装するという、非常に手作業の証明を書くことができます。

```
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.
```

ふたつの帰納的ケースを分離するために空白（空行）を使用しますが、これらの空行に実際には意味的な内容がないことに注意してください; Coq は、その空行が証明の実際の場合分けの構造 (case structure) と一致するよう強制しません。2 番目のケースでは、自動的に生成された仮説名が明示的に記述されています。結果として、定理のステートメントに対する無害な変更は、この証明を無効にすることができます。

```
Reset eval_times.
Theorem eval_times : ∀ 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.
```

帰納的な仮説は、*IHe1* と *IHe2* ではなく、いまでは *IHx1* と *IHx2* という名前になりました。

```
Abort.
```

証明の中で後で参照するすべての名前に対して明示的な束縛子 (binder) を与えるために、`induction` のより明示的な呼び出しを使うことにします。

```
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.
```

`induction` に、*intro* **パターン** を渡し、| 文字を使用して、帰納法の場合分けを区別します。ある場合では、?を書いて Coq に自動的に名前を生成するように要求し、その名前を対応する新しい変数に割り当てるための明示的な名前を書きます。

証明のの脆弱さを避けるために intro パターンを使用するには、変数が導入された順番についての、一見重要でない事実を把握する必要があることは明らかです。

したがって、e を x で置き換えるとスクリプトは動作し続けますが、複雑になっています。 おそらく、どちらの証明も特に追いかけるのは容易ではありません。

その不満のカテゴリーは、静的な人工物としての証明を理解することと関係があります。一般的なプログラミングと同じように、深刻なプロジェクトでは、仕様変更に伴い証明の進化をサポートすることが重要になる傾向があります。上の例のような構造化されていない証明は、定理にあわせて更新するのが非常に難しい場合があります。たとえば、times を修正してバグを導入するときに、最後の証明のスクリプトがどのように機能するかを考えてみましょう。

```
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.
```

スクリプトをステップ・バイ・ステップで進めることなく、何が間違っているのか分かりますか？問題は trivial が決して失敗しないということです。もともと、trivial は、反射性に従う等式を証明することに成功していました。times の変更は、その等式がもはや真実でない場合につながります。trivial の呼び出しは、幸いにも、偽の等式を適所に残し、帰納法のふたつめの条件のためのタクティクの範囲を続けます。残念ながら、それらのタクティクは代わりに **ひとつめ** の場合に適用されてしまいます。

trivial についての問題は、代わりに solve [trivial] と書くことで「解決」することができ、予期しないことが起こった場合に早期にエラーが通知されます。しかし、根本的な問題は、タクティク呼び出しの構文が、それが生成するサブゴールの数を意味するものではないということです。この問題のはるかに混乱する例が考えられます。例えば、補題 L が余分な仮説をとるように修正された場合、apply L の使用は以前より多くの副目標を生成するようになります。古い構造化されていない証明スクリプトは、不適切なサブゴールに適用されたタクティクにより、絶望的に混乱します。構造が不足しているため、集められた、エラーが発生した証明スクリプト内の正確なポイントの知識は、比較的少ししかありません。

```
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.
```

多くの実際の開発では、本質的に構造化されていない証明を、慎重な字下げ規則、文書としての役割を果たすための偶発的 (idempotence) なタクティクの大文字小文字の使い分 (case-marker) けなどを適用することによって構造化するようにしています。これらの戦略のすべては、今見せた抽象化の失敗と同じ種類の障害に苦しんでいます。あなたが証明スクリプトでインデントを気にしていると感じたら、スクリプトがうまく構成されていないという兆候です。

現在の証明を単一のタクティクで書き直すことができます。

```
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.
```

セミコロン演算子の形式を使用して、生成された各サブゴールに対して異なる戦術を指定することができます。この変更により、スクリプトの堅牢性が向上します: もはや、あるケースから別のケースに適用された戦術について心配する必要はありません。それでも、証明スクリプトは特に読めるものではありません。たぶん大部分の読者は、なぜ定理が正しいのかを説明するのに役立つとは思わないでしょう。*bullets* **ブレット** や、Coq 8.4 で提供されている中括弧を使用したスクリプトでも言えます。

これは、Coq によって強制される方法で証明構造を表現して、上記のようなコードをセ

ミコロンの代わりにピリオドをつけて、対話的に進めることができます。

対話的なスクリプトの再生 (replay) が容易になりますが、読みやすさの助けには本当になりません。

証明したい定理の拡張を考えると、状況は悪化します。乗算のノードを **exp** 型に追加し、証明がどのように費用を払うか (fares) を見てみましょう。

```
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.
```

驚くことではありませんが、古い証明は失敗します。なぜなら、それはふたつの帰納の場合分け (case) があることが明示しているからです。スクリプトを更新するには、帰納の場合分けが生成される順序を少なくとも覚えておく必要があります。これにより、新しい場合分けを適切な場所に挿入することができます。それでも、明示的な一連の場合分けの内で発生した場合に、証明のステップを対話的に進めることができないため、場合分けを追加するのは苦労します。

```
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.
```

今、この本のほとんどで、私たちが従ってきた証拠のスタイルがどれほど素晴らしいかを知る立場にあります。

```
Reset eval_times.

Hint Rewrite mult_plus_distr_l.

Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e; crush.
Qed.
```

このスタイルは手強い (hard) 真実によって動機づけられます: ひとりの手作業による証明スクリプトはほとんどの場合、ほとんどすべての人に納得できません。私は、ステップ・バイ・ステップの形式的な証明は情報を伝達するための貧弱な方法だと主張します。したがって、可能な限り、ステップを切り出し、自動化することもできます。

証明の説明的 (illustrative) な価値はどうでしょうか？ほとんどの非形式的な証明は、証明の大きなアイデアを伝えるために読み込まれます。どうすれば、induction *e*; *crush* を読むことが、なんらかの大きなアイデアを伝えることができるでしょうか？

私の立場は、標準的な自動化が見つけることができるアイデアは結局はあまり大きくなく、**本当に** 大きなアイデアはヒントとして追加された補題を使って表現されるべきであるということです。

ひとつの例が私が意味するものを説明するのに役立つはずです。加算と乗算の結合性 (associativity) を使って、式を書き換える関数を考えてみましょう。

```
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 _ ⇒ _ | _ ⇒ _ 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 タクティクはこのゴールをどのように完成 (finish) するか知りません。ゴールを手動で完成しないといけません。

```
  rewrite ← IHe2; crush.
```

しかし、この洞察を別の補題に分ければ、証明は理解しやすくなり、保守することも容易になります。

```
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 _ ⇒ _ | _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
Qed.
```

その制限があると、複雑な帰納法による証明はそれぞれの帰納法の場合分け (case) に対して、ひとつのヒントに依存する可能性があります。各ヒントの補題は関連する場合分けを再現 (restate) する可能性があります。手動の証明スクリプトと比較して、わかりやすい結果が得られます。スクリプトは、場合分けが生成される順序に依存する必要がなくなりました。補題は完全な証明の文脈を含んでいるので、補題はタクティクのコードの断片であるよりも、別々にこなす (digest) 方が簡単です。このような文脈は、スクリプトを対話的に踏むことによってモノリシックな手作業の証明から抽出することができます。

より一般的な状況は、大きな帰納法は、自動化が短い作業を行う、いくつかの簡単な場合を有することです。それ以外の場合、自動化は標準的な単純化を実行します。これらのケースの中には、かなり複雑な証明が必要なものもあります。

そのような場合は、その補題のステートメントが場合分けの単純化されたバージョンをコピーしたのならば、それ自身のヒントの補題に相当するかもしれません。

あるいは、主な定理のための証明スクリプトは、特定の場合を対象とするいくつかの自動化コードで拡張することができるかもしれません。

そのような対象のスクリプトは、証明の階層構造や、ケースの順序付けや、名前の束縛の構造 (name binding structure) の知識がなくても、読まれ理解される可能性があるため、手作業による証明よりも望ましいです。

Coq 戦術の一般的なスタイルの代わりに、*declarative* **宣言的** スタイルがあります。これは今日最も頻繁に Isar [43] 言語に関連付けられています。宣言的な証明スクリプトは、人間の可読性を目指して、サブゴール構造とローカル名の導入について非常に明示的です。

自動証明のコーディングは、各々の深刻な定理のための新しい自動化を構築する価値がないという考えに関連した証明言語の範囲外であるとみなされます。

この本では、定理に特化した自動化の多くの例を示しました。私は、重要な結果を得るために重要であると信じています。

宣言的な証明スクリプトを使用すると、定理の変更のためにスクリプトを変更しやすくなりますが、本書の代替 *adaptive* **適応形** スタイルでは、多くのバージョンの定理で *same* **同じ** スクリプトを使用できます。

おそらく私は、形式的な証明の全体が必然的に、人々には面白くない詳細で構成されていると考える悲観論者ですが、補題の選択を通じて証明固有の詳細を伝えることに焦点を当てることが、私の好みです。

さらに、適応形 (adaptive) の Ltac スクリプトには、独立して理解できる一連の自動化が含まれています。たとえば、大きな `repeat match` ループでは、それぞれのケースを別々に消化することができます。これは、スクリプトの階層構造をより一般的なスタイルで理解しようとするのとは大きく異なっています。

適応形のスクリプトは可変な束縛に依存 (rely) しますが、一般的に非常に小さなスコープでしか使用だけです。一方、従来のスクリプトを理解するには、コードのページ全体で、潜在的なローカル変数の同一性 (identities) を追跡する必要があります。

すべての定理を自動的に（適応形の証明スクリプトの意味で）証明するのはなぜ理にかなっているでしょうが、すべてのプログラムを自動的に構築するのはそうではありあせん。私の見解では、*program synthesis* **プログラム合成**は広範なアプリケーションに適した非常に便利なアイデアです！実際には、仕様から自動的にプログラムを見つけるのに困難な障害があります。

典型的な仕様は、プログラム特性の記述において網羅的ではありません。例えば、特定の機械アーキテクチャ上の性能の詳細は、しばしば省略されます。

結果として、合成されたプログラムは、ある意味では正しいかもしれませんが、別の観点から欠陥に苦しみます。プログラム合成の研究はこの問題に対処する方法を生み出し続けますが、定理証明のための状況は基本的にに異なります。

数学的な実践に続いて、私たちが気にする形式証明の唯一の性質は、それが証明する定理であり、この特性を自動的にチェックすることは自明です。

言い換えれば、証明を受け入れられるものとする、簡単な基準 (criterion) では、自動探索は簡単です。

もちろん、実際には、長期の保守を容易にするための証明の理解可能性 (understandability) にも気を配ります。これは上に概説した技術を動機付けるものであり、次の節ではいくつかの関連するアドバイスを提供します。

15.2 自動化証明のデバックと保守

完全に自動化された証明は、仕様の変更に対して自動的に適応することの可能性を広げているので、望ましいです。狭い領域内のうまく設計されたスクリプトは、それが解決する問題の定式化に多くの変更を生き残ることができます。それでも、高階論理を使って作業しているので、ほとんどの定理は明らかな決定可能な定理にはなりません。長く使われている自動化された証明のほとんどは更新が必要であることは避けられません。

証明を更新する準備が整う前に、それらの証明を最初に書く必要があります。完全に自動化されたスクリプトは仕様の変更に対して最も堅牢ですが、新しいすべての証明をその形式で直接書き込むことは困難です。代わりに、定理を、探索的な証明 (exploratory proving) で開始し、それを徐々に適切な自動化された形式に修正することは有用です。

第 8 章のこの定理を考えてみましょう。ほとんど手作業で証明することから始まり、各ステップの後に、さっさと済ますべきこと (low-hanging fruit) を済ますために *crush* を呼びます。手作業では、場合分け分析の対象となる式を選択する必要があります。

```
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.

Theorem reassoc_correct : ∀ e, eval (reassoc e) = eval e.
  induction e; crush;
    match goal with
      | [ ⊢ context[match ?E with Const _ ⇒ _ | _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
```

ひとつのサブゴールが残ります：

```
  ============================
   eval e1 × (eval e3 + 1 - 1) × eval e4 = eval e1 × eval e2
```

不適切に選択された書き換え規則選択され (fired)、ゴールを別のヒントが適用されなくなった式 (form) に変更しました。私たちが多くのヒントを持つ大きな開発の真っ只中にいると想像してください。問題をどのように診断しますか？最初に、帰納的な証明のどの場合分けが間違っているのかわからないかもしれません。自動化した手順を分けて手動で適用すると便利です。

```
  Restart.

  Ltac t := crush; match goal with
                     | [ ⊢ context[match ?E with Const _ ⇒ _ | _ ⇒ _ end] ] ⇒
                       destruct E; crush
                   end.

  induction e.
```

単純化が起こる前にサブゴールを見るので、（最初のサブゴールは）定数の場合分けを見ていることがわかります。

```
t.
```

加算 (addition) のための、次のサブゴールも問題なく済まされます (discharged)。

```
t.
```

最後のサブゴールはは乗算 (multiplication) のためのものであり、ここでは、上で説明した状態で、証明は立ち往生します。

```
t.
```

この時点で t は何をしていますか？`info` コマンドはこの種の質問に答えるのに役立ちます。これを書いている時点で、最近の Coq のリリースでは、`info` はもう機能していませんが、私は復帰することを期待しています。

```
Undo.
info t.

== simpl in *; intuition; subst; autorewrite with core in *;
   simpl in *; intuition; subst; autorewrite with core in *;
   simpl in *; intuition; subst; destruct (reassoc e2).
   simpl in *; intuition.

   simpl in *; intuition.

   simpl in *; intuition; subst; autorewrite with core in *;
      refine (eq_ind_r
                (fun n : nat ⇒
                 n × (eval e3 + 1 - 1) × eval e4 = eval e1 × eval e2) _ IHe1);
      autorewrite with core in *; simpl in *; intuition;
   subst; autorewrite with core in *; simpl in *;
   intuition; subst.
```

t の実行の詳細なトレースが表示されます。非常に一般的な *crush* タクティクを使用しているので、これらのステップの多くは効果がなく、より一般的なタクティクのインスタンスとしてのみ発生します。詳細をコピー＆ペーストして、どこが間違っているかを確認することができます。

```
Undo.
```

私たちは任意にスクリプトを複数の塊に分割しています。最初のいくつかは悪影響 (harm) を及ぼさないようです。

```
simpl in *; intuition; subst; autorewrite with core in *.
simpl in *; intuition; subst; autorewrite with core in *.
simpl in *; intuition; subst; destruct (reassoc e2).
```

```
simpl in *; intuition.
simpl in *; intuition.
```

次のステップは、最終的に証明されないいサブゴールに至らせる原因 (culprit) として、明かにされます。

```
simpl in *; intuition; subst; autorewrite with core in *.
```

私たちはさらに責任を明確にする (assign blame) ためにステップを分けることができます。

```
Undo.
```

```
simpl in *.
intuition.
subst.
autorewrite with core in *.
```

これらの 4 つのタクティクのうち最後のものは、書き換えを行ないました。何が起こったのかを正確に知ることができます。

info コマンドは証明のステップの階層の視点 (view) を示し、元のトレースに表示されたステップのひとつ info を適用することで、より詳細なレベルまでズームダウンすることができます。

```
Undo.
```

```
info autorewrite with core in *.
```

```
== refine (eq_ind_r (fun n : nat ⇒ n = eval e1 × eval e2) _
              (confounder (reassoc e1) e3 e4)).
```

書き換えの方法はやや凝ったもの (baroque) ですが、定理 confounder が最終的な原因 (culprit) であることがわかります。この時点で、そのヒントを取り除き、鍵となる補題 rewr の代替バージョンを証明するか、あるいは他のいくつかの救済方法を考え出すことができます。この種の問題を解決するには、問題が明らかになったならば、比較的容易になる傾向があります。

```
Abort.
```

時に、古い証明スクリプトが完成しないようにしても、開発への変更は望ましくないパフォーマンスへの影響 (consequences) をもたらすことがあります。パフォーマンスへの影響が十分に厳しい場合は、実際的な意味で、証明スクリプトが壊れたとみなすことができます。

パフォーマンスについての驚きの一例を以下に示します。

```
Section slow.
  Hint Resolve trans_eq.
```

問題の中心的な要素は、推移性 (transitivity) をヒントとして追加することです。推移性を利用すると、証明検索で指数関数的な探索空間を探索することが容易になり、後でトラブルを起こします。

```
Variable A : Set.
Variables P Q R S : A → A → Prop.
```

Variable f : $A \to A$.

Hypothesis $H1$: $\forall$ x y, P x $y \to$ Q x $y \to$ R x $y \to$ f x = f y.
Hypothesis $H2$: $\forall$ x y, S x $y \to$ R x y.

単純な補題をとても早く証明し、 Time コマンドを使っていかに早く証明できるか計測できます。

Lemma slow : $\forall$ x y, P x $y \to$ Q x $y \to$ S x $y \to$ f x = f y.
Time eauto 6.

```
Finished transaction in 0. secs (0.068004u,0.s)
```

Qed.

ここで、別の仮説を追加します。これは無益なもの (innocent) です：実際には、定理としても証明可能です。

Hypothesis $H3$: $\forall$ x y, x = $y \to$ f x = f y.

Lemma slow' : $\forall$ x y, P x $y \to$ Q x $y \to$ S x $y \to$ f x = f y.
Time eauto 6.

```
Finished transaction in 2. secs (1.264079u,0.s)
```

検索時間があまりにも長くなったのはなぜでしょうか？info コマンドはあまり役に立ちません。検索の結果だけを表示するだけで、有用でないと判明したすべてのパスが表示されるわけではありません。

Restart.
info eauto 6.

== intro x; intro y; intro H; intro $H0$; intro $H4$;
simple eapply $trans\_eq$.
simple apply eq_refl.

simple eapply $trans\_eq$.
simple apply eq_refl.

simple eapply $trans\_eq$.
simple apply eq_refl.

simple apply $H1$.
eexact H.

eexact $H0$.

simple apply $H2$; eexact $H4$.

この出力は、証明検索に時間がかかりすぎる理由を教えてくれませんが、推論をヒントとして追加したことを忘れてしまった場合に役立つヒントを提供します。eauto タクティクは深さ優先探索を適用しており、実際のアクションが終わっている証明スクリプトは、それぞれの呼び出しがひとつの副目標を済ます (discharege) ために反射性を使用する無意味な呼び出しの連鎖の中に埋もれてしまいます。

eauto の深さを指定する引数への各増分は、推論の別の愚かな使用を追加します。この浪費された証明の努力は、証明検索が誤ったステップを決してしない限り、線形時間オーバーヘッドを追加するだけです。新しい仮説を追加する前に、間違った手順はありませんでしたが、何らかの形で追加すると、新しい欠陥のあるパスが可能になりました。有効にしたパスを理解するために、debug コマンドを使用できます。

```
Restart.
debug eauto 6.
```

出力は大きな証明木です。木の始まりは何が起こっているかを明らかにするには十分です：

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 eq_refl
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 eq_refl
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 eq_refl
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 eq_refl
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 *depth*=0 eapply *trans_eq*
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.2 *depth*=1 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.2.1 *depth*=0 eapply *trans_eq*
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.3 *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
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1 *depth*=1 eapply *trans_eq*
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.1 *depth*=1 apply eq_refl
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.1.1 *depth*=0 eapply *trans_eq*
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.2 *depth*=1 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.2.1 *depth*=0 eapply *trans_eq*
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.3 *depth*=0 eapply *trans_eq*

最初の選択肢 eauto は $H3$ を適用することです。なぜなら、$H3$ は一致するすべての仮説とヒントの仮説が少ないためです。しかしながら、生成された単一の仮説は証明できないことがわかりました。それは、推移的、反射性、平等の対称性の指数関数的なツリーでそれを証明しようとすることを止めるものではありません。証明実行の際の顕著な時間のすべてを占めるのは、最初の apply $H3$ の子供たちです。より現実的な開発では、この出力を debug として使用して、推移性をヒントとして追加することは悪い考えであることに気付くかもしれません。

```
Qed.
End slow.
```

上記の状況が悪化すると、Require Import のようなコマンドを使ってライブラリモジュールをインポートすると、非常に最近のバージョンの Coq には、データベースからヒントを取り除くためのメカニズムが含まれていますが、適切な解決策は、ヒントをエクスポートする際には非常に慎重でなければなりません。ヒントを明示的に呼び出されたときにのみ使用できるように、名前付きデータベースにヒントを入れることを検討してください（第 13 章を参照)。

あまりにも多くのメモリを使用する証明スクリプトで終わるのも簡単です。タクティクが実行されるにつれて、重大な証拠検索では多くの可能な手段が考慮されるため、証明項の生成は避けられます。また、使用されない副証明 (subproof) に関する証明項語を作成したくありません。代わりに、タクティックの実行は、Qed を実行したときにのみ実行されるように、*thunks*（中断された計算、クロージャで表される）を維持します。これらのサンクは大量のスペースを消費することがあります。thunks を早期に強制することでメモリを大幅に節約できたとしても、証明スクリプトが使用可能なメモリを使い果たしてしまいます。

abstract タクティカルは、いくつかのサブゴールを独自の補題として証明することによって、thunks を強制するのに役立ちます。

例えば、証明 induction x; *crush* は、多くの場合で、induction x; abstract *crush* に変更することによって大幅に少ないメモリを使用することができます。abstract の主な制限は、初期状態では未定の統一変数がなく、完全に証明されたサブゴールにしか適用できないということです。それでも、多くの大規模な自動証明は、abstract を介して膨大なメモリ節約を実現することができます。

15.3 モージュール

前章のリフレクションによる証明の例は、Ltac スクリプトよりもより強力な形式的保証を伴う抽象的な証明戦略を実装する機会を示しています。Coq の **モジュール・システム** は、一般的な定理 (generic theorems) の厳格な開発のための別なツールを提供します。この機能は、Standard ML [21] と OCaml にあるモジュールz・システムからインスピレーションを受けています。以下の議論では、これらのシステムのひとつについて熟知していることを前提とします。

ML のモジュールは、抽象型 (abstract types) とそれらの型に対する操作のグループ化を容易にします。また、モジュール間の関数である **関手** (*functors*) もサポートされています。関手の標準的な例は、キーとなるのドメインと、関連する比較演算子を記述するモジュール

からデータ構造の実装を構築するものです。

依存型を持つ基本言語にモジュールを追加すると、モジュールとファンクタを使用して、代数で一般的な推論の種類を正式化することが可能になります。例えば、以下のモジュールのシグネチャは、群 (group) と呼ばれる代数構造の本質を捉えています。群は、台集合 (carrier set) G、結合性を満たす二項演算 f、f の左単位元 id、f の左逆元ある演算 i で構成されます。

```
Module Type GROUP.
  Parameter G : Set.
  Parameter f : G → G → G.
  Parameter id : G.
  Parameter i : G → G.

  Axiom assoc : ∀ a b c, f (f a b) c = f a (f b c).
  Axiom ident : ∀ a, f id a = a.
  Axiom inverse : ∀ a, f (i a) a = id.
End GROUP.
```

多くの便利な定理は任意の群を保持していますが、他のモジュールのシグネチャでそのような定理文をいくつか取ります。

```
Module Type GROUP_THEOREMS.
  Declare Module M : GROUP.

  Axiom ident' : ∀ a, M.f a M.id = a.

  Axiom inverse' : ∀ a, M.f a (M.i a) = M.id.

  Axiom unique_ident : ∀ id', (∀ a, M.f id' a = a) → id' = M.id.
End GROUP_THEOREMS.
```

これらの定理の一般的な証明は、入力が任意の群 M であるファンクタで実装されます。

```
Module GroupProofs (M : GROUP) : GROUP_THEOREMS with Module M := M.
```

ML のように、Coq はシグネチャをモジュールに帰属させるための複数のオプションを提供します。ここでは、*opaque ascription* を実装するコロン演算子を使用して、シグネチャによって公開されていないモジュールのすべての詳細を隠します。もうひとつのオプションは、実装の詳細を隠すことなく、シグネチャの互換性をチェックする <: 演算子による *transparent ascription* です。ここでは、*op* を使用します。このような洗練がなければ、あまり有用でないいくつかの未知のグループについての定理を証明する出力モジュールを得ることになります。Coq の不透明な帰納法は、型のチェックだけでなく、識別子の *definitions* もタイプチェックと定理証明の意味を持つため、ML の中で関連性 (analogues) がないと望ましくない結果をもたらす可能性があることにも注意してください。

```
  Module M := M.
```

構築しているモジュールが GROUP_THEOREMS 署名を満たしていることを確認するために、関手の引数の M に追加のローカル名を追加します。

```
  Import M.

  Theorem inverse' : ∀ a, f a (i a) = id.
    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 id = a.
    intro.
    rewrite ← (inverse a).
    rewrite ← assoc.
    rewrite inverse'.
    apply ident.
  Qed.

  Theorem unique_ident : ∀ id', (∀ a, M.f id' a = a) → id' = M.id.
    intros.
    rewrite ← (H id).
    symmetry.
    apply ident'.
  Qed.
End GroupProofs.
```

整数は + について群をなすことをしめします。

```
Require Import ZArith.
Open Scope Z_scope.

Module Int.
  Definition G := Z.
  Definition f x y := x + y.
  Definition id := 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 id a = a.
    unfold f, id; crush.
  Qed.
  Theorem inverse : ∀ a, f (i a) a = id.
    unfold f, i, id; crush.
  Qed.
End Int.
```

次に、一般的な群の定理の整数固有のバージョンを生成することができます。

```
Module IntProofs := GroupProofs(Int).
Check IntProofs.unique_ident.
```

IntProofs.unique_ident
 : $\forall$ *e'* : *Int.G*, ($\forall$ *a* : *Int.G*, *Int.f e' a* = *a*) $\rightarrow$ *e'* = *Int.e*

Int.G のような射影 (projection) は、我々がそれらに割り当てた具体的な値と定義的に等しいことが知られているので、上記の定理は、以下のより自然な再記述を簡単な結果としてもたらします：

```
Theorem unique_ident : ∀ id', (∀ a, id' + a = a) → id' = 0.
  exact IntProofs.unique_ident.
Qed.
```

ML のように、モジュール・システムは、大規模な開発を構築する効果的な方法を提供します。ML とは異なり、Coq モジュールは表現力を追加せず、依存型のレコードのフィールド (inhabitant) として任意のモジュールを実装することができます。

多くの場合、依存レコードよりも使いやすくするセカンド・クラス (second-class nature) のモジュールです。

モジュールは非常に限定された方法でしか使用できないので、上記の例が示すように、特別なコマンドと編集モードで便利なモジュールコーディングをサポートする方が簡単です。

レコードの同形 (isomorphic) の実装は、モジュールのサブタイプ化やモジュールのフィールドのインポートなどの利便性の欠如に悩まされていました。

一方、すべてのモジュール値は静的に決定されなければならないので、モジュールは、例えば、特定の関数パラメータに基づいて、通常の関数の定義内で計算されないことがあります。

15.4 ビルド・プロセス

ソフトウェア開発のように、大規模な Coq プロジェクトは、複数のファイルに分割してライブラリに分解すると、はるかに管理しやすくなります。Coq と Proof General はこれらの活動を非常にうまくサポートしています。

ディレクトリ `LIB` に格納され、ファイル `A.v`、`B.v`、および `C.v` の間で分割される Lib という名前のライブラリを考えてみましょう。シンプルな Makefile は、標準的な Coq ツール `coq_makefile` を使ってライブラリをコンパイルします。

```
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
```

Makefile は、プロジェクトに含めるファイル名のリストを保持する変数 `VS` を定義することから始まります。主なターゲットは `coq` です。これは `Makefile.coq` と呼ばれる補助的な Makefile の構成に依存します。別のルールは、そのファイルを構築する方法を説明します。カレント・ディレクトリのファイルをライブラリ Lib に属するとみなすために、

`-R`

フラグを使用して、`coq_makefile` を呼び出します。この Makefile は、（たとえば）`X.v` が `X.vo` にコンパイルされるように、各モジュールのコンパイルされたバージョンを構築します。

ここで、`B.v` のコードは、実行後に `A.v` の定義を参照することがあります

`Require Import` Lib.A.

ライブラリ Lib はモジュールとして表示され、

`A.v`

から定義されるサブモジュール A を含みます。これらは Coq のモジュール・システムの意味での本物のモジュールであり、ファンクタ (functor) などに渡すことができます。

`Require Import` コマンドは、さらにふたつの基本的 (primitive) なコマンドの便利な組み合わせです。`Require` コマンドは、名前付きモジュールを含む `.vo` ファイルを見つけ、モジュールがメモリにロードされていることを確認します。

`Import` コマンドは、名前付きモジュールのすべてのトップレベルの定義を現在の名前空間にロードし、対応する `.vo` ファイルを持たないローカルなモジュールで使用できます。

別のコマンド、`Load` は、名前付きファイルの内容をそのまま挿入するためのものです。証明スクリプトを再実行するのを避け、コードを変更することなくディレクトリ構造の再編成を容易にするので、モジュールベースのコマンド (suhara: Load ではなく、Require と Import) を使用する方が一般的に適しています。

今度は、`CLIENT` ディレクトリにある、Client と呼ばれる独自の Makefile を持つ、別の開発用ライブラリを使用したいと考えます。

```
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
```

`coq_makefile`

の呼び出しを変更して、ライブラリ LIB がどこにあるかを示します。現在、`D.v` と `E.v` は、実行後の LIB モジュール *A* の定義を参照できます。

`Require Import` LIB.A.

そして、`E.v` は、実行によって `D.v` の定義を参照できます。

`Require Import` CLIENT.D.

ライブラリを複数のファイルに分割すると便利ですが、クライアントのコードが個別にライブラリのモジュールをインポートすることも不便です。たとえば、余分なソースファイル "Lib.v" を LIB のディレクトリと Makefile に追加することで、両方にとっての最良を得ることができます。このファイルには、次の行だけが含まれています：

`Require Export` LIB.A LIB.B LIB.C.

ここで、クライアントのコードはすべての LIB のモジュールから、すべての定義を単純に実行することでインポートできます。

`Require Import` LIB.

上記のふたつの Makefile は多くのコードを共有しているので、実際には、複数のライブラリ固有の Makefile に含まれる（定義を）共通の Makefile に定義すると便利です。

残りの成分は、Proof General でライブラリのコードのファイルを編集する適切な方法です。第 2 章の `.emacs` コードのこのスニペットを思い出してください。このスニペットは、Proof General にこの本に関連するライブラリを見つける場所を教えてくれます。

```
(custom-set-variables
  ...
  '(coq-prog-args '("-R" "/path/to/cpdt/src" "Cpdt"))
  ...
)
```

現在の例を対話的に編集するには、適切な場所を指すようにフラグを変更するだけです。

```
(custom-set-variables
  ...
; '(coq-prog-args '("-R" "/path/to/cpdt/src" "Cpdt"))
  '(coq-prog-args '("-R" "LIB" "Lib" "-R" "CLIENT" "Client"))
  ...
)
```

複数のプロジェクトに取り組んでいるときは、この設定の複数のバージョンを.emacs ファイルに残しておいて、いつでもそのうちのひとつを除いてすべてをコメントアウトすると便利です。プロジェクトの間で切り替えるには、コメント構造を変更して Emacs を再起動します。

あるいは、ディレクトリにローカルな設定の方法を再訪し、

CLIENT

の .dir-locals.el ファイルに次のように書き込むことができます：

```
((coq-mode . ((coq-prog-args .
  ("-emacs-U" "-R" "LIB" "Lib" "-R" "CLIENT" "Client")))))
```

このアプローチの欠点は、コードのユーザが、そのようなファイルに置くことが許されている任意の Emacs Lisp プログラムを信頼したくないことで、マッピングを手動で追加することが好きなことです。

Coq の比較的最近のバージョンでは、このすべてに対するもう一つのより原理的なアプローチがサポートされています。プロジェクトの設定とソースファイルのリストは、最新のバージョンの、coq_makefile、Proof General、CoqIDE で一様に処理される _CoqProject という名前の単一ファイルに保存されます。詳細については、Coq マニュアルを参照してください。

Chapter 16

A Taste of Reasoning About Programming Language Syntax

Reasoning about the syntax and semantics of programming languages is a popular application of proof assistants. Before proving the first theorem of this kind, it is necessary to choose a formal encoding of the informal notions of syntax, dealing with such issues as variable binding conventions. I believe the pragmatic questions in this domain are far from settled and remain as important open research problems. However, in this chapter, I will demonstrate two underused encoding approaches. Note that I am not recommending either approach as a silver bullet! Mileage will vary across concrete problems, and I expect there to be significant future advances in our knowledge of encoding techniques. For a broader introduction to programming language formalization, using more elementary techniques, see *Software Foundations*[1] by Pierce et al.

This chapter is also meant as a case study, bringing together what we have learned in the previous chapters. We will see a concrete example of the importance of representation choices; translating mathematics from paper to Coq is not a deterministic process, and different creative choices can have big impacts. We will also see dependent types and scripted proof automation in action, applied to solve a particular problem as well as possible, rather than to demonstrate new Coq concepts.

I apologize in advance to those readers not familiar with the theory of programming language semantics. I will make a few remarks intended to relate the material here with common ideas in semantics, but these remarks should be safe for others to skip.

We will define a small programming language and reason about its semantics, expressed as an interpreter into Coq terms, much as we have done in examples throughout the book. It will be helpful to build a slight extension of *crush* that tries to apply functional extensionality, an axiom we met in Chapter 12, which says that two functions are equal if they map equal inputs to equal outputs. We also use `f_equal` to simplify goals of a particular form that will come up with the term denotation function that we define shortly.

```
Ltac ext := let x := fresh "x" in extensionality x.
```

[1] http://www.cis.upenn.edu/~bcpierce/sf/

```
Ltac pl := crush; repeat (match goal with
                           | [ ⊢ (fun x ⇒ _) = (fun y ⇒ _) ] ⇒ ext
                           | [ ⊢ _ _ _ ?E _ = _ _ _ ?E _ ] ⇒ f_equal
                           | [ ⊢ ?E ::: _ = ?E ::: _ ] ⇒ f_equal
                           | [ ⊢ hmap _ ?E = hmap _ ?E ] ⇒ f_equal
                         end; crush).
```

At this point in the book source, some auxiliary proofs also appear.

Here is a definition of the type system we will use throughout the chapter. It is for simply typed lambda calculus with natural numbers as the base type.

```
Inductive type : Type :=
| Nat : type
| Func : type → type → type.

Fixpoint typeDenote (t : type) : Type :=
  match t with
    | Nat ⇒ nat
    | Func t1 t2 ⇒ typeDenote t1 → typeDenote t2
  end.
```

Now we have some choices as to how we represent the syntax of programs. The two sections of the chapter explore two such choices, demonstrating the effect the choice has on proof complexity.

16.1 Dependent de Bruijn Indices

The first encoding is one we met first in Chapter 9, the *dependent de Bruijn index* encoding. We represent program syntax terms in a type family parameterized by a list of types, representing the *typing context*, or information on which free variables are in scope and what their types are. Variables are represented in a way isomorphic to the natural numbers, where number 0 represents the first element in the context, number 1 the second element, and so on. Actually, instead of numbers, we use the **member** dependent type family from Chapter 9.

```
Module FirstOrder.
```

Here is the definition of the **term** type, including variables, constants, addition, function abstraction and application, and let binding of local variables.

```
Inductive term : list type → type → Type :=
| Var : ∀ G t, member t G → term G t

| Const : ∀ G, nat → term G Nat
| Plus : ∀ G, term G Nat → term G Nat → term G Nat
```

```
| Abs : ∀ G dom ran, term (dom :: G) ran → term G (Func dom ran)
| App : ∀ G dom ran, term G (Func dom ran) → term G dom → term G ran

| Let : ∀ G t1 t2, term G t1 → term (t1 :: G) t2 → term G t2.
Implicit Arguments Const [G].
```

Here are two example term encodings, the first of addition packaged as a two-argument curried function, and the second of a sample application of addition to constants.

```
Example add : term nil (Func Nat (Func Nat Nat)) :=
  Abs (Abs (Plus (Var (HNext HFirst)) (Var HFirst))).

Example three_the_hard_way : term nil Nat :=
  App (App add (Const 1)) (Const 2).
```

Since dependent typing ensures that any term is well-formed in its context and has a particular type, it is easy to translate syntactic terms into Coq values.

```
Fixpoint termDenote G t (e : term G t) : hlist typeDenote G → typeDenote t :=
  match e with
    | Var _ _ x ⇒ fun s ⇒ hget s x

    | Const _ n ⇒ fun _ ⇒ n
    | Plus _ e1 e2 ⇒ fun s ⇒ termDenote e1 s + termDenote e2 s

    | Abs _ _ _ e1 ⇒ fun s ⇒ fun x ⇒ termDenote e1 (x ::: s)
    | App _ _ _ e1 e2 ⇒ fun s ⇒ (termDenote e1 s) (termDenote e2 s)

    | Let _ _ _ e1 e2 ⇒ fun s ⇒ termDenote e2 (termDenote e1 s ::: s)
  end.
```

With this term representation, some program transformations are easy to implement and prove correct. Certainly we would be worried if this were not the the case for the *identity* transformation, which takes a term apart and reassembles it.

```
Fixpoint ident G t (e : term G t) : term G t :=
  match e with
    | Var _ _ x ⇒ Var x

    | Const _ n ⇒ Const n
    | Plus _ e1 e2 ⇒ Plus (ident e1) (ident e2)

    | Abs _ _ _ e1 ⇒ Abs (ident e1)
    | App _ _ _ e1 e2 ⇒ App (ident e1) (ident e2)

    | Let _ _ _ e1 e2 ⇒ Let (ident e1) (ident e2)
  end.
```

```
Theorem identSound : ∀ G t (e : term G t) s,
  termDenote (ident e) s = termDenote e s.
  induction e; pl.
Qed.
```

A slightly more ambitious transformation belongs to the family of *constant folding* optimizations we have used as examples in other chapters.

```
Fixpoint cfold G t (e : term G t) : term G t :=
  match e with
    | Plus G e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
      let maybeOpt := match e1' return _ with
                        | Const _ n1 ⇒
                          match e2' return _ with
                            | Const _ n2 ⇒ Some (Const (n1 + n2))
                            | _ ⇒ None
                          end
                        | _ ⇒ None
                      end in
      match maybeOpt with
        | None ⇒ Plus e1' e2'
        | Some e' ⇒ e'
      end

    | Abs _ _ _ e1 ⇒ Abs (cfold e1)
    | App _ _ _ e1 e2 ⇒ App (cfold e1) (cfold e2)

    | Let _ _ _ e1 e2 ⇒ Let (cfold e1) (cfold e2)

    | e ⇒ e
  end.
```

The correctness proof is more complex, but only slightly so.

```
Theorem cfoldSound : ∀ G t (e : term G t) s,
  termDenote (cfold e) s = termDenote e s.
  induction e; pl;
    repeat (match goal with
              | [ ⊢ context[match ?E with Var _ _ _ ⇒ _ | _ ⇒ _ end] ] ⇒
                dep_destruct E
            end; pl).
Qed.
```

The transformations we have tried so far have been straightforward because they do not have interesting effects on the variable binding structure of terms. The dependent de Bruijn representation is called *first-order* because it encodes variable identity explicitly; all such representations incur bookkeeping overheads in transformations that rearrange binding structure.

As an example of a tricky transformation, consider one that removes all uses of "`let` x = *e1* `in` *e2*" by substituting *e1* for x in *e2*. We will implement the translation by pairing the "compile-time" typing environment with a "run-time" value environment or *substitution*, mapping each variable to a value to be substituted for it. Such a substitute term may be placed within a program in a position with a larger typing environment than applied at the point where the substitute term was chosen. To support such context transplantation, we need *lifting*, a standard de Bruijn indices operation. With dependent typing, lifting corresponds to weakening for typing judgments.

The fundamental goal of lifting is to add a new variable to a typing context, maintaining the validity of a term in the expanded context. To express the operation of adding a type to a context, we use a helper function insertAt.

```
Fixpoint insertAt (t : type) (G : list type) (n : nat) {struct n} : list type :=
  match n with
    | O => t :: G
    | S n' => match G with
                | nil => t :: G
                | t' :: G' => t' :: insertAt t G' n'
              end
  end.
```

Another function lifts bound variable instances, which we represent with **member** values.

```
Fixpoint liftVar t G (x : member t G) t' n : member t (insertAt t' G n) :=
  match x with
    | HFirst G' => match n return member t (insertAt t' (t :: G') n) with
                     | O => HNext HFirst
                     | _ => HFirst
                   end
    | HNext t'' G' x' => match n return member t (insertAt t' (t'' :: G') n) with
                           | O => HNext (HNext x')
                           | S n' => HNext (liftVar x' t' n')
                         end
  end.
```

The final helper function for lifting allows us to insert a new variable anywhere in a typing context.

```
Fixpoint lift' G t' n t (e : term G t) : term (insertAt t' G n) t :=
  match e with
    | Var _ _ x => Var (liftVar x t' n)
```

```
    | Const _ n ⇒ Const n
    | Plus _ e1 e2 ⇒ Plus (lift' t' n e1) (lift' t' n e2)

    | Abs _ _ _ e1 ⇒ Abs (lift' t' (S n) e1)
    | App _ _ _ e1 e2 ⇒ App (lift' t' n e1) (lift' t' n e2)

    | Let _ _ _ e1 e2 ⇒ Let (lift' t' n e1) (lift' t' (S n) e2)
  end.
```

In the `Let` removal transformation, we only need to apply lifting to add a new variable at the *beginning* of a typing context, so we package lifting into this final, simplified form.

```
Definition lift G t' t (e : term G t) : term (t' :: G) t :=
  lift' t' O e.
```

Finally, we can implement `Let` removal. The argument of type **hlist** (**term** *G'*) *G* represents a substitution mapping each variable from context *G* into a term that is valid in context *G'*. Note how the Abs case (1) extends via lifting the substitution *s* to hold in the broader context of the abstraction body *e1* and (2) maps the new first variable to itself. It is only the `Let` case that maps a variable to any substitute beside itself.

```
Fixpoint unlet G t (e : term G t) G' : hlist (term G') G → term G' t :=
  match e with
    | Var _ _ x ⇒ fun s ⇒ hget s x

    | Const _ n ⇒ fun _ ⇒ Const n
    | Plus _ e1 e2 ⇒ fun s ⇒ Plus (unlet e1 s) (unlet e2 s)

    | Abs _ _ _ e1 ⇒ fun s ⇒ Abs (unlet e1 (Var HFirst ::: hmap (lift _) s))
    | App _ _ _ e1 e2 ⇒ fun s ⇒ App (unlet e1 s) (unlet e2 s)

    | Let _ t1 _ e1 e2 ⇒ fun s ⇒ unlet e2 (unlet e1 s ::: s)
  end.
```

We have finished defining the transformation, but the parade of helper functions is not over. To prove correctness, we will use one more helper function and a few lemmas. First, we need an operation to insert a new value into a substitution at a particular position.

```
Fixpoint insertAtS (t : type) (x : typeDenote t) (G : list type) (n : nat) {struct n}
  : hlist typeDenote G → hlist typeDenote (insertAt t G n) :=
  match n with
    | O ⇒ fun s ⇒ x ::: s
    | S n' ⇒ match G return hlist typeDenote G
                               → hlist typeDenote (insertAt t G (S n')) with
                   | nil ⇒ fun s ⇒ x ::: s
                   | t' :: G' ⇒ fun s ⇒ hhd s ::: insertAtS t x n' (htl s)
                 end
```

```
  end.
Implicit Arguments insertAtS [t G].
```

Next we prove that liftVar is correct. That is, a lifted variable retains its value with respect to a substitution when we perform an analogue to lifting by inserting a new mapping into the substitution.

```
Lemma liftVarSound : ∀ t' (x : typeDenote t') t G (m : member t G) s n,
  hget s m = hget (insertAtS x n s) (liftVar m t' n).
  induction m; destruct n; dep_destruct s; pl.
Qed.

Hint Resolve liftVarSound.
```

An analogous lemma establishes correctness of lift'.

```
Lemma lift'Sound : ∀ G t' (x : typeDenote t') t (e : term G t) n s,
  termDenote e s = termDenote (lift' t' n e) (insertAtS x n s).
  induction e; pl;
    repeat match goal with
             | [ IH : ∀ n s, _ = termDenote (lift' _ n ?E) _
                 ⊢ context[lift' _ (S ?N) ?E] ] ⇒ specialize (IH (S N))
           end; pl.
Qed.
```

Correctness of lift itself is an easy corollary.

```
Lemma liftSound : ∀ G t' (x : typeDenote t') t (e : term G t) s,
  termDenote (lift t' e) (x ::: s) = termDenote e s.
  unfold lift; intros; rewrite (lift'Sound _ x e O); trivial.
Qed.

Hint Rewrite hget_hmap hmap_hmap liftSound.
```

Finally, we can prove correctness of unletSound for terms in arbitrary typing environments.

```
Lemma unletSound' : ∀ G t (e : term G t) G' (s : hlist (term G') G) s1,
  termDenote (unlet e s) s1
  = termDenote e (hmap (fun t' (e' : term G' t') ⇒ termDenote e' s1) s).
  induction e; pl.
Qed.
```

The lemma statement is a mouthful, with all its details of typing contexts and substitutions. It is usually prudent to state a final theorem in as simple a way as possible, to help your readers believe that you have proved what they expect. We follow that advice here for the simple case of terms with empty typing contexts.

```
Theorem unletSound : ∀ t (e : term nil t),
  termDenote (unlet e HNil) HNil = termDenote e HNil.
  intros; apply unletSound'.
```

```
Qed.
End FirstOrder.
```

The Let removal optimization is a good case study of a simple transformation that may turn out to be much more work than expected, based on representation choices. In the second part of this chapter, we consider an alternate choice that produces a more pleasant experience.

16.2 Parametric Higher-Order Abstract Syntax

In contrast to first-order encodings, *higher-order* encodings avoid explicit modeling of variable identity. Instead, the binding constructs of an *object language* (the language being formalized) can be represented using the binding constructs of the *meta language* (the language in which the formalization is done). The best known higher-order encoding is called *higher-order abstract syntax* (HOAS) [32], and we can start by attempting to apply it directly in Coq.

```
Module HigherOrder.
```

With HOAS, each object language binding construct is represented with a *function* of the meta language. Here is what we get if we apply that idea within an inductive definition of term syntax.

```
Inductive term : type → Type :=
| Const : nat → term Nat
| Plus : term Nat → term Nat → term Nat

| Abs : ∀ dom ran, (term dom → term ran) → term (Func dom ran)
| App : ∀ dom ran, term (Func dom ran) → term dom → term ran

| Let : ∀ t1 t2, term t1 → (term t1 → term t2) → term t2.
```

However, Coq rejects this definition for failing to meet the strict positivity restriction. For instance, the constructor Abs takes an argument that is a function over the same type family **term** that we are defining. Inductive definitions of this kind can be used to write non-terminating Gallina programs, which breaks the consistency of Coq's logic.

An alternate higher-order encoding is *parametric HOAS*, as introduced by Washburn and Weirich [42] for Haskell and tweaked by me [5] for use in Coq. Here the idea is to parameterize the syntax type by a type family standing for a *representation of variables*.

```
Section var.
  Variable var : type → Type.

  Inductive term : type → Type :=
  | Var : ∀ t, var t → term t
```

```
  | Const : nat → term Nat
  | Plus : term Nat → term Nat → term Nat

  | Abs : ∀ dom ran, (var dom → term ran) → term (Func dom ran)
  | App : ∀ dom ran, term (Func dom ran) → term dom → term ran

  | Let : ∀ t1 t2, term t1 → (var t1 → term t2) → term t2.
End var.

Implicit Arguments Var [var t].
Implicit Arguments Const [var].
Implicit Arguments Abs [var dom ran].
```

Coq accepts this definition because our embedded functions now merely take *variables* as arguments, instead of arbitrary terms. One might wonder whether there is an easy loophole to exploit here, instantiating the parameter *var* as **term** itself. However, to do that, we would need to choose a variable representation for this nested mention of **term**, and so on through an infinite descent into **term** arguments.

We write the final type of a closed term using polymorphic quantification over all possible choices of *var* type family.

```
Definition Term t := ∀ var, term var t.
```

Here are the new representations of the example terms from the last section. Note how each is written as a function over a *var* choice, such that the specific choice has no impact on the *structure* of the term.

```
Example add : Term (Func Nat (Func Nat Nat)) := fun var ⇒
  Abs (fun x ⇒ Abs (fun y ⇒ Plus (Var x) (Var y))).

Example three_the_hard_way : Term Nat := fun var ⇒
  App (App (add var) (Const 1)) (Const 2).
```

The argument *var* does not even appear in the function body for add. How can that be? By giving our terms expressive types, we allow Coq to infer many arguments for us. In fact, we do not even need to name the *var* argument!

```
Example add' : Term (Func Nat (Func Nat Nat)) := fun _ ⇒
  Abs (fun x ⇒ Abs (fun y ⇒ Plus (Var x) (Var y))).

Example three_the_hard_way' : Term Nat := fun _ ⇒
  App (App (add' _) (Const 1)) (Const 2).
```

Even though the *var* formal parameters appear as underscores, they *are* mentioned in the function bodies that type inference calculates.

16.2.1 Functional Programming with PHOAS

It may not be at all obvious that the PHOAS representation admits the crucial computable operations. The key to effective deconstruction of PHOAS terms is one principle: treat the *var* parameter as an unconstrained choice of *which data should be annotated on each variable*. We will begin with a simple example, that of counting how many variable nodes appear in a PHOAS term. This operation requires no data annotated on variables, so we simply annotate variables with **unit** values. Note that, when we go under binders in the cases for Abs and Let, we must provide the data value to annotate on the new variable we pass beneath. For our current choice of **unit** data, we always pass tt.

```
Fixpoint countVars t (e : term (fun _ ⇒ unit) t) : nat :=
  match e with
    | Var _ _ ⇒ 1

    | Const _ ⇒ 0
    | Plus e1 e2 ⇒ countVars e1 + countVars e2

    | Abs _ _ e1 ⇒ countVars (e1 tt)
    | App _ _ e1 e2 ⇒ countVars e1 + countVars e2

    | Let _ _ e1 e2 ⇒ countVars e1 + countVars (e2 tt)
  end.
```

The above definition may seem a bit peculiar. What gave us the right to represent variables as **unit** values? Recall that our final representation of closed terms is as polymorphic functions. We merely specialize a closed term to exactly the right variable representation for the transformation we wish to perform.

```
Definition CountVars t (E : Term t) := countVars (E (fun _ ⇒ unit)).
```

It is easy to test that CountVars operates properly.

```
Eval compute in CountVars three_the_hard_way.
```

$= 2$

In fact, PHOAS can be used anywhere that first-order representations can. We will not go into all the details here, but the intuition is that it is possible to interconvert between PHOAS and any reasonable first-order representation. Here is a suggestive example, translating PHOAS terms into strings giving a first-order rendering. To implement this translation, the key insight is to tag variables with strings, giving their names. The function takes as an additional input a string giving the name to be assigned to the next variable introduced. We evolve this name by adding a prime to its end. To avoid getting bogged down in orthogonal details, we render all constants as the string "*N*".

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

```
Fixpoint pretty t (e : term (fun _ ⇒ string) t) (x : string) : string :=
  match e with
    | Var _ s ⇒ s
```

```
    | Const _ ⇒ "N"
    | Plus e1 e2 ⇒ "(" ++ pretty e1 x ++ " + " ++ pretty e2 x ++ ")"

    | Abs _ _ e1 ⇒ "(fun " ++ x ++ " => " ++ pretty (e1 x) (x ++ "'") ++ ")"
    | App _ _ e1 e2 ⇒ "(" ++ pretty e1 x ++ " " ++ pretty e2 x ++ ")"

    | Let _ _ e1 e2 ⇒ "(let " ++ x ++ " = " ++ pretty e1 x ++ " in "
      ++ pretty (e2 x) (x ++ "'") ++ ")"
  end.
Definition Pretty t (E : Term t) := pretty (E (fun _ ⇒ string)) "x".
Eval compute in Pretty three_the_hard_way.
    = "(((fun x => (fun x' => (x + x'))) N) N)"
```

However, it is not necessary to convert to first-order form to support many common operations on terms. For instance, we can implement substitution of terms for variables. The key insight here is to *tag variables with terms*, so that, on encountering a variable, we can simply replace it by the term in its tag. We will call this function initially on a term with exactly one free variable, tagged with the appropriate substitute. During recursion, new variables are added, but they are only tagged with their own term equivalents. Note that this function squash is parameterized over a specific *var* choice.

```
Fixpoint squash var t (e : term (term var) t) : term var t :=
  match e with
    | Var _ e1 ⇒ e1

    | Const n ⇒ Const n
    | Plus e1 e2 ⇒ Plus (squash e1) (squash e2)

    | Abs _ _ e1 ⇒ Abs (fun x ⇒ squash (e1 (Var x)))
    | App _ _ e1 e2 ⇒ App (squash e1) (squash e2)

    | Let _ _ e1 e2 ⇒ Let (squash e1) (fun x ⇒ squash (e2 (Var x)))
  end.
```

To define the final substitution function over terms with single free variables, we define Term1, an analogue to Term that we defined before for closed terms.

```
Definition Term1 (t1 t2 : type) := ∀ var, var t1 → term var t2.
```

Substitution is defined by (1) instantiating a Term1 to tag variables with terms and (2) applying the result to a specific term to be substituted. Note how the parameter *var* of squash is instantiated: the body of Subst is itself a polymorphic quantification over *var*, standing for a variable tag choice in the output term; and we use that input to compute a tag choice for the input term.

```
Definition Subst (t1 t2 : type) (E : Term1 t1 t2) (E' : Term t1) : Term t2 :=
```

```
  fun var ⇒ squash (E (term var) (E' var)).
Eval compute in Subst (fun _ x ⇒ Plus (Var x) (Const 3)) three_the_hard_way.

   = fun var : type → Type ⇒
     Plus
       (App
          (App
             (Abs
                (fun x : var Nat ⇒
                 Abs (fun y : var Nat ⇒ Plus (Var x) (Var y))))
             (Const 1)) (Const 2)) (Const 3)
```

One further development, which may seem surprising at first, is that we can also implement a usual term denotation function, when we *tag variables with their denotations*.

```
Fixpoint termDenote t (e : term typeDenote t) : typeDenote t :=
  match e with
    | Var _ v ⇒ v

    | Const n ⇒ n
    | Plus e1 e2 ⇒ termDenote e1 + termDenote e2

    | Abs _ _ e1 ⇒ fun x ⇒ termDenote (e1 x)
    | App _ _ e1 e2 ⇒ (termDenote e1) (termDenote e2)

    | Let _ _ e1 e2 ⇒ termDenote (e2 (termDenote e1))
  end.

Definition TermDenote t (E : Term t) : typeDenote t :=
  termDenote (E typeDenote).

Eval compute in TermDenote three_the_hard_way.

   = 3
```

To summarize, the PHOAS representation has all the expressive power of more standard first-order encodings, and a variety of translations are actually much more pleasant to implement than usual, thanks to the novel ability to tag variables with data.

16.2.2 Verifying Program Transformations

Let us now revisit the three example program transformations from the last section. Each is easy to implement with PHOAS, and the last is substantially easier than with first-order representations.

First, we have the recursive identity function, following the same pattern as in the previous subsection, with a helper function, polymorphic in a tag choice; and a final function that instantiates the choice appropriately.

```
Fixpoint ident var t (e : term var t) : term var t :=
  match e with
    | Var _ x ⇒ Var x

    | Const n ⇒ Const n
    | Plus e1 e2 ⇒ Plus (ident e1) (ident e2)

    | Abs _ _ e1 ⇒ Abs (fun x ⇒ ident (e1 x))
    | App _ _ e1 e2 ⇒ App (ident e1) (ident e2)

    | Let _ _ e1 e2 ⇒ Let (ident e1) (fun x ⇒ ident (e2 x))
  end.

Definition Ident t (E : Term t) : Term t := fun var ⇒
  ident (E var).
```

Proving correctness is both easier and harder than in the last section, easier because we do not need to manipulate substitutions, and harder because we do the induction in an extra lemma about ident, to establish the correctness theorem for Ident.

```
Lemma identSound : ∀ t (e : term typeDenote t),
  termDenote (ident e) = termDenote e.
  induction e; pl.
Qed.

Theorem IdentSound : ∀ t (E : Term t),
  TermDenote (Ident E) = TermDenote E.
  intros; apply identSound.
Qed.
```

The translation of the constant-folding function and its proof work more or less the same way.

```
Fixpoint cfold var t (e : term var t) : term var t :=
  match e with
    | Plus e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
        match e1', e2' with
          | Const n1, Const n2 ⇒ Const (n1 + n2)
          | _, _ ⇒ Plus e1' e2'
        end

    | Abs _ _ e1 ⇒ Abs (fun x ⇒ cfold (e1 x))
    | App _ _ e1 e2 ⇒ App (cfold e1) (cfold e2)

    | Let _ _ e1 e2 ⇒ Let (cfold e1) (fun x ⇒ cfold (e2 x))
```

```
    | e ⇒ e
  end.

Definition Cfold t (E : Term t) : Term t := fun var ⇒
  cfold (E var).

Lemma cfoldSound : ∀ t (e : term typeDenote t),
  termDenote (cfold e) = termDenote e.
  induction e; pl;
    repeat (match goal with
              | [ ⊢ context[match ?E with Var _ _ ⇒ _ | _ ⇒ _ end] ] ⇒
                dep_destruct E
            end; pl).
Qed.

Theorem CfoldSound : ∀ t (E : Term t),
  TermDenote (Cfold E) = TermDenote E.
  intros; apply cfoldSound.
Qed.
```

Things get more interesting in the Let-removal optimization. Our recursive helper function adapts the key idea from our earlier definitions of squash and Subst: tag variables with terms. We have a straightforward generalization of squash, where only the Let case has changed, to tag the new variable with the term it is bound to, rather than just tagging the variable with itself as a term.

```
Fixpoint unlet var t (e : term (term var) t) : term var t :=
  match e with
    | Var _ e1 ⇒ e1

    | Const n ⇒ Const n
    | Plus e1 e2 ⇒ Plus (unlet e1) (unlet e2)

    | Abs _ _ e1 ⇒ Abs (fun x ⇒ unlet (e1 (Var x)))
    | App _ _ e1 e2 ⇒ App (unlet e1) (unlet e2)

    | Let _ _ e1 e2 ⇒ unlet (e2 (unlet e1))
  end.

Definition Unlet t (E : Term t) : Term t := fun var ⇒
  unlet (E (term var)).
```

We can test Unlet first on an uninteresting example, three_the_hard_way, which does not use Let.

```
Eval compute in Unlet three_the_hard_way.

   = fun var : type → Type ⇒
```

```
App
  (App
     (Abs
        (fun x : var Nat ⇒
         Abs (fun x0 : var Nat ⇒ Plus (Var x) (Var x0))))
     (Const 1)) (Const 2)
```

Next, we try a more interesting example, with some extra Lets introduced in three_the_hard_way.

```
Definition three_a_harder_way : Term Nat := fun _ ⇒
  Let (Const 1) (fun x ⇒ Let (Const 2) (fun y ⇒ App (App (add _) (Var x)) (Var y))).
Eval compute in Unlet three_a_harder_way.
```

```
  = fun var : type → Type ⇒
    App
      (App
         (Abs
            (fun x : var Nat ⇒
             Abs (fun x0 : var Nat ⇒ Plus (Var x) (Var x0))))
         (Const 1)) (Const 2)
```

The output is the same as in the previous test, confirming that Unlet operates properly here.

Now we need to state a correctness theorem for Unlet, based on an inductively proved lemma about unlet. It is not at all obvious how to arrive at a proper induction principle for the lemma. The problem is that we want to relate two instantiations of the same Term, in a way where we know they share the same structure. Note that, while Unlet is defined to consider all possible *var* choices in the output term, the correctness proof conveniently only depends on the case of *var* := typeDenote. Thus, one parallel instantiation will set *var* := typeDenote, to take the denotation of the original term. The other parallel instantiation will set *var* := **term** typeDenote, to perform the unlet transformation in the original term.

Here is a relation formalizing the idea that two terms are structurally the same, differing only by replacing the variable data of one with another isomorphic set of variable data in some possibly different type family.

```
Section wf.
  Variables var1 var2 : type → Type.
```

To formalize the tag isomorphism, we will use lists of values with the following record type. Each entry has an object language type and an appropriate tag for that type, in each of the two tag families *var1* and *var2*.

```
  Record varEntry := {
    Ty : type;
    First : var1 Ty;
    Second : var2 Ty
```

}.

Here is the inductive relation definition. An instance `wf` *G e1 e2* asserts that terms *e1* and *e2* are equivalent up to the variable tag isomorphism *G*. Note how the Var rule looks up an entry in *G*, and the Abs and `Let` rules include recursive `wf` invocations inside the scopes of quantifiers to introduce parallel tag values to be considered as isomorphic.

```
Inductive wf : list varEntry → ∀ t, term var1 t → term var2 t → Prop :=
| WfVar : ∀ G t x x', In {| Ty := t; First := x; Second := x' |} G
  → wf G (Var x) (Var x')

| WfConst : ∀ G n, wf G (Const n) (Const n)

| WfPlus : ∀ G e1 e2 e1' e2', wf G e1 e1'
  → wf G e2 e2'
  → wf G (Plus e1 e2) (Plus e1' e2')

| WfAbs : ∀ G dom ran (e1 : _ dom → term _ ran) e1',
  (∀ x1 x2, wf ({| First := x1; Second := x2 |} :: G) (e1 x1) (e1' x2))
  → wf G (Abs e1) (Abs e1')

| WfApp : ∀ G dom ran (e1 : term _ (Func dom ran)) (e2 : term _ dom) e1' e2',
  wf G e1 e1'
  → wf G e2 e2'
  → wf G (App e1 e2) (App e1' e2')

| WfLet : ∀ G t1 t2 e1 e1' (e2 : _ t1 → term _ t2) e2', wf G e1 e1'
  → (∀ x1 x2, wf ({| First := x1; Second := x2 |} :: G) (e2 x1) (e2' x2))
  → wf G (Let e1 e2) (Let e1' e2').
End wf.
```

We can state a well-formedness condition for closed terms: for any two choices of tag type families, the parallel instantiations belong to the `wf` relation, starting from an empty variable isomorphism.

```
Definition Wf t (E : Term t) := ∀ var1 var2, wf nil (E var1) (E var2).
```

After digesting the syntactic details of Wf, it is probably not hard to see that reasonable term encodings will satisfy it. For example:

```
Theorem three_the_hard_way_Wf : Wf three_the_hard_way.
  red; intros; repeat match goal with
                        | [ ⊢ wf _ _ _ ] ⇒ constructor; intros
                      end; intuition.
Qed.
```

Now we are ready to give a nice simple proof of correctness for unlet. First, we add one hint to apply a small variant of a standard library theorem connecting Forall, a higher-order predicate asserting that every element of a list satisfies some property; and In, the list membership predicate.

```
Hint Extern 1 ⇒ match goal with
                  | [ H1 : Forall _ _, H2 : In _ _ ⊢ _ ] ⇒ apply (Forall_In H1 _ H2)
                end.
```

The rest of the proof is about as automated as we could hope for.

```
Lemma unletSound : ∀ G t (e1 : term _ t) e2,
  wf G e1 e2
  → Forall (fun ve ⇒ termDenote (First ve) = Second ve) G
  → termDenote (unlet e1) = termDenote e2.
  induction 1; pl.
Qed.

Theorem UnletSound : ∀ t (E : Term t), Wf E
  → TermDenote (Unlet E) = TermDenote E.
  intros; eapply unletSound; eauto.
Qed.
```

With this example, it is not obvious that the PHOAS encoding is more tractable than dependent de Bruijn. Where the de Bruijn version had lift and its helper functions, here we have Wf and its auxiliary definitions. In practice, Wf is defined once per object language, while such operations as lift often need to operate differently for different examples, forcing new implementations for new transformations.

The reader may also have come up with another objection: via Curry-Howard, wf proofs may be thought of as first-order encodings of term syntax! For instance, the In hypothesis of rule WfVar is equivalent to a **member** value. There is some merit to this objection. However, as the proofs above show, we are able to reason about transformations using first-order representation only for their inputs, not their outputs. Furthermore, explicit numbering of variables remains absent from the proofs.

Have we really avoided first-order reasoning about the output terms of translations? The answer depends on some subtle issues, which deserve a subsection of their own.

16.2.3 Establishing Term Well-Formedness

Can there be values of type Term t that are not well-formed according to Wf? We expect that Gallina satisfies key *parametricity* [34] properties, which indicate how polymorphic types may only be inhabited by specific values. We omit details of parametricity theorems here, but $\forall$ t (E : Term t), Wf E follows the flavor of such theorems. One option would be to assert that fact as an axiom, "proving" that any output of any of our translations is well-formed. We could even prove the soundness of the theorem on paper meta-theoretically, say by considering some particular model of CIC.

To be more cautious, we could prove Wf for every term that interests us, threading such proofs through all transformations. Here is an example exercise of that kind, for Unlet.

First, we prove that wf is *monotone*, in that a given instance continues to hold as we add new variable pairs to the variable isomorphism.

```
Hint Constructors wf.
Hint Extern 1 (In _ _) ⇒ simpl; tauto.
Hint Extern 1 (Forall _ _) ⇒ eapply Forall_weaken; [ eassumption | simpl ].

Lemma wf_monotone : ∀ var1 var2 G t (e1 : term var1 t) (e2 : term var2 t),
  wf G e1 e2
  → ∀ G', Forall (fun x ⇒ In x G') G
    → wf G' e1 e2.
  induction 1; pl; auto 6.
Qed.

Hint Resolve wf_monotone Forall_In'.
```

Now we are ready to prove that unlet preserves any wf instance. The key invariant has to do with the parallel execution of unlet on two different *var* instantiations of a particular term. Since unlet uses **term** as the type of variable data, our variable isomorphism context G contains pairs of terms, which, conveniently enough, allows us to state the invariant that any pair of terms in the context is also related by wf.

```
Hint Extern 1 (wf _ _ _) ⇒ progress simpl.

Lemma unletWf : ∀ var1 var2 G t (e1 : term (term var1) t) (e2 : term (term var2) t),
  wf G e1 e2
  → ∀ G', Forall (fun ve ⇒ wf G' (First ve) (Second ve)) G
    → wf G' (unlet e1) (unlet e2).
  induction 1; pl; eauto 9.
Qed.
```

Repackaging unletWf into a theorem about Wf and Unlet is straightforward.

```
Theorem UnletWf : ∀ t (E : Term t), Wf E
  → Wf (Unlet E).
  red; intros; apply unletWf with nil; auto.
Qed.
```

This example demonstrates how we may need to use reasoning reminiscent of that associated with first-order representations, though the bookkeeping details are generally easier to manage, and bookkeeping theorems may generally be proved separately from the independently interesting theorems about program transformations.

16.2.4 A Few More Remarks

Higher-order encodings derive their strength from reuse of the meta language's binding constructs. As a result, we can write encoded terms so that they look very similar to their informal counterparts, without variable numbering schemes like for de Bruijn indices. The example encodings above have demonstrated this fact, but modulo the clunkiness of explicit use of the constructors of **term**. After defining a few new Coq syntax notations, we can work with terms in an even more standard form.

```
Infix "->" := Func (right associativity, at level 52).

Notation "^" := Var.
Notation "#" := Const.
Infix "@" := App (left associativity, at level 50).
Infix "@+" := Plus (left associativity, at level 50).
Notation "\ x : t , e" := (Abs (dom := t) (fun x => e))
  (no associativity, at level 51, x at level 0).
Notation "[ e ]" := (fun _ => e).

Example Add : Term (Nat -> Nat -> Nat) :=
  [\x : Nat, \y : Nat, ^x @+ ^y].

Example Three_the_hard_way : Term Nat :=
  [Add _ @ #1 @ #2].

Eval compute in TermDenote Three_the_hard_way.

     = 3

End HigherOrder.
```

The PHOAS approach shines here because we are working with an object language that has an easy embedding into Coq. That is, there is a straightforward recursive function translating object terms into terms of Gallina. All Gallina programs terminate, so clearly we cannot hope to find such embeddings for Turing-complete languages; and non-Turing-complete languages may still require much more involved translations. I have some work [6] on modeling semantics of Turing-complete languages with PHOAS, but my impression is that there are many more advances left to be made in this field, possibly with completely new term representations that we have not yet been clever enough to think up.

Conclusion

I have designed this book to present the key ideas needed to get started with productive use of Coq. Many people have learned to use Coq through a variety of resources, yet there is a distinct lack of agreement on structuring principles and techniques for easing the evolution of Coq developments over time. Here I have emphasized two unusual techniques: programming with dependent types and proving with scripted proof automation. I have also tried to present other material following my own take on how to keep Coq code beautiful and scalable.

Part of the attraction of Coq and similar tools is that their logical foundations are small. A few pages of LaTeX code suffice to define CIC, Coq's logic, yet there do not seem to be any practical limits on which mathematical concepts may be encoded on top of this modest base. At the same time, the *pragmatic* foundation of Coq is vast, encompassing tactics, libraries, and design patterns for programs, theorem statements, and proof scripts. I hope the preceding chapters have given a sense of just how much there is to learn before it is possible to drive Coq with the same ease with which many readers write informal proofs! The pay-off of this learning process is that many proofs, especially those with many details to check, become much easier to write than they are on paper. Further, the truth of such theorems may be established with much greater confidence, even without reading proof details.

As Coq has so many moving parts to catalogue mentally, I have not attempted to describe most of them here; nor have I attempted to give exhaustive descriptions of the few I devote space to. To those readers who have made it this far through the book, my advice is: read through the Coq manual, front to back, at some level of detail. Get a sense for which bits of functionality are available. Dig more into those categories that sound relevant to the developments you want to build, and keep the rest in mind in case they come in handy later.

In a domain as rich as this one, the learning process never ends. The Coq Club mailing list (linked from the Coq home page) is a great place to get involved in discussions of the latest improvements, or to ask questions about stumbling blocks that you encounter. (I hope that this book will save you from needing to ask some of the most common questions!) I believe the best way to learn is to get started using Coq to build some development that interests you.

Good luck!

Bibliography

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[2] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software*, pages 515–529, 1997.

[3] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[4] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[5] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 143–156, 2008.

[6] Adam Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–106, 2010.

[7] Coq Development Team. The Coq proof assistant reference manual, version 8.4. 2012.

[8] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3), 1988.

[9] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934.

[10] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[11] Eduardo Giménez. A tutorial on recursive types in Coq. Technical Report 0221, INRIA, May 1998.

[12] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[13] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Original paper manuscript from 1969.

[14] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, pages 257–267, 1973.

[15] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1984.

[16] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[17] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.

[18] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[19] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. In *Proceedings of the 15th European Symposium on Programming*, pages 54–68, 2006.

[20] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[21] David MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 198–207, 1984.

[22] Conor McBride. Elimination with a motive. In *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 197–216, 2000.

[23] Adam Megacz. A coinductive monad for prop-bounded recursion. In *Proceedings of the ACM Workshop Programming Languages meets Program Verification*, pages 11–20, 2007.

[24] J Strother Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Automated Reasoning Series. Kluwer Academic Publishers, 1996.

[25] J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5k86 floating-point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

[26] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[27] Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9:471–477, 1999.

[28] Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, 1993.

[29] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[30] Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. *Haskell 98 Language and Libraries: The Revised Report*, chapter 4.3.3. 1998.

[31] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[32] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[33] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[34] J.C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[35] John C. Reynolds. The discoveries of continuations. *Lisp Symb. Comput.*, 6(3-4):233–248, November 1993.

[36] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[37] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[38] Leon Sterling and Ehud Shapiro. *The Art of Prolog, 2nd Edition*. MIT Press, 1994.

[39] Thomas Streicher. *Semantical Investigations into Intensional Type Theory*. Habilitationsschrift, LMU München, 1993.

[40] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[41] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[42] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008.

[43] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 167–184, 1999.

[44] Glynn Winskel. *The Formal Semantics of Programming Languages*, chapter 8. MIT Press, 1993.

[45] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, 2003.

Index