Computational Type Theory and Interactive Theorem Proving with Coq

Lecture Notes Summer 2019
Introduction to Computational Logic

Version of August 2, 2019

Gert Smolka
Department of Computer Science
Saarland University

Contents

1 Getting Started

We start with basic ideas from type theory and Coq. The main issues we discuss are inductive types, recursive functions, and equational reasoning using structural induction. We will see inductive types for booleans, natural numbers, and pairs. On these types we will define functions using equations. This will involve functions that are recursive, cascaded (i.e., return functions), higher-order (i.e., take functions as arguments), and polymorphic (i.e., take types as leading arguments). Recursion will be limited to structural recursion so that termination can be checked automatically.

Our main interest is in proving equations involving recursive functions (e.g., commutativity of addition, $x + y = y + x$). This will involve proof steps known as conversion, rewriting, structural case analysis, and structural induction. Equality will appear in a general form called propositional equality, and in a specialized form called computational equality. Computational equality is a prominent design aspect of type theory that is important for mechanized proofs.

We will follow the equational paradigm and define functions with equations, thus avoiding lambda abstractions and matches. We will mostly define cascaded functions and use the accompanying notation known from functional programming.

Type theory is a foundational theory starting from computational intuitions. Its approach to mathematical foundations is very different from set theory. We may say that type theory explains things computationally while set theory explains things at a level of abstraction where computation is not an issue. When working with type theory, set-theoretic explanations (e.g., of functions) are usually not helpful, so free your mind for a foundational restart.

1.1 Booleans

In Coq, even basic types like the type of booleans are defined as **inductive types**. The type definition for the booleans looks as follows:

$$\begin{aligned} \mathsf{B} &: \mathbb{T} \\ \mathsf{T} &: \mathsf{B} \\ \mathsf{F} &: \mathsf{B} \end{aligned}$$

The definition introduces three constants called **constructors**: the type B and its two values T and F. In the definition each constructor appears with its type. Note that the type B also has a type, which is the **universe** $\mathbb{T}$.

We now define the negation function

$$! : \mathsf{B} \to \mathsf{B}$$

for booleans with two equations:

$$\begin{aligned} !\mathsf{T} &:= \mathsf{F} \\ !\mathsf{F} &:= \mathsf{T} \end{aligned}$$

The **defining equations** also serve as **computation rules**. For computation, the equations are applied as left-to-right rewrite rules. For instance, we have

$$!!!\mathsf{T} = !!\mathsf{F} = !\mathsf{T} = \mathsf{F}$$

by rewriting with the first, the second, and again with the first equation ($!!!\mathsf{T}$ is to be read as $!(!(!\mathsf{T}))$). Computation in Coq is logical and is used in proofs. For instance, the equation

$$!!!\mathsf{T} = !\mathsf{T}$$

follows by computation:

$$\begin{aligned} & !!!\mathsf{T} \\ =\ & !!\mathsf{F} \\ =\ & !\mathsf{T} \\ =\ & \mathsf{F} \end{aligned} \qquad\qquad \begin{aligned} & !\mathsf{T} \\ =\ & \mathsf{F} \end{aligned}$$

We speak of **computational equality** and of **proof by computation**.

Proving the equation

$$!!x = x$$

involving a boolean variable x takes more than computation since none of the defining equations applies. What is needed is **structural case analysis** on the boolean variable x, which reduces the claim $!!x = x$ to two equations $!!\mathsf{T} = \mathsf{T}$ and $!!\mathsf{F} = \mathsf{F}$, which both follow by computation.

Next we define functions for boolean conjunction and boolean disjunction:

$$\begin{aligned} \& &: \mathsf{B} \to \mathsf{B} \to \mathsf{B} \\ \mathsf{T} \,\&\, y &:= y \\ \mathsf{F} \,\&\, y &:= \mathsf{F} \end{aligned} \qquad\qquad \begin{aligned} | &: \mathsf{B} \to \mathsf{B} \to \mathsf{B} \\ \mathsf{T} \mid y &:= \mathsf{T} \\ \mathsf{F} \mid y &:= y \end{aligned}$$

The defining equations introduce asymmetry since they define the functions by case analysis on the first argument. Alternatively, one could define the functions by case analysis on the second argument, resulting in different computation rules. Since the equations defining a function must be **disjoint** and **exhaustive** when applied from left to right, it is not possible to define boolean conjunction and disjunction with equations treating both arguments symmetrically.

Given the definitions of the basic boolean connectives, we can prove the usual boolean indenties with boolean case analysis and computation. For instance, the distributivity law

$$x \,\&\, (y \mid z) = (x \,\&\, y) \mid (x \,\&\, z)$$

follows by case analysis on x and computation, reducing the law to the trivial equations $y \mid z = y \mid z$ and $\mathsf{F} = \mathsf{F}$. Note that the commutativity law

$$x \,\&\, y = y \,\&\, x$$

needs case analysis on both x and y to reduce to computationally trivial equations.

1.2 Numbers

The inductive type for the numbers 0, 1, 2, ...

$$\begin{aligned} \mathsf{N} &: \mathbb{T} \\ 0 &: \mathsf{N} \\ \mathsf{S} &: \mathsf{N} \to \mathsf{N} \end{aligned}$$

has two constructors providing 0 and the successor function S. A number n can now be represented by the term that applies the constructor S n-times to the constructor 0. For instance, the term $\mathsf{S}(\mathsf{S}(\mathsf{S}0))$ represents the number 3. We will use the familiar notations 0, 1, 2, ... for the terms 0, $\mathsf{S}0$, $\mathsf{S}(\mathsf{S}0)$, ... representing the numbers. The constructor representation of numbers dates back to the Dedekind-Peano axioms.

We now define an addition function doing case analysis on the first argument:

$$\begin{aligned} &+ : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &0 + y \;:=\; y \\ &\mathsf{S}x + y \;:=\; \mathsf{S}(x + y) \end{aligned}$$

The second equation is **recursive** because it uses the function '+' being defined at the right hand side.

Coq only admits **total functions**, that is, functions that for every value of the argument type of the function yield a value of the result type of the function. To satisfy this basic requirement, all recursive definitions must be **terminating**. Coq checks termination automatically as part of type checking. To make an automatic termination check possible, recursion is restricted to **structural recursion** on a single inductive argument of a function (an inductive argument is an argument with an inductive type). The definition of '+' is an example of a structural recursion on numbers taking place on the first argument. The recursion appears in the second equation where the argument is $\mathsf{S}x$ and the recursive application is on x.

We define **truncating subtraction** for numbers:

$$\begin{aligned} - &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ 0 - y &:= 0 \\ \mathsf{S}x - 0 &:= \mathsf{S}x \\ \mathsf{S}x - \mathsf{S}y &:= x - y \end{aligned}$$

The primary case analysis is on the first argument, with a nested case analysis on the second argument in the successor case. The equations are exhaustive and disjoint. The recursion happens in the third equation. We say that the recursion is structural on the first argument since the primary case analysis is on the first argument.

Following the scheme we have seen for addition, functions for multiplication and exponentiation can be defined as follows:

$$\begin{aligned} \cdot &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ 0 \cdot y &:= 0 \\ \mathsf{S}x \cdot y &:= y + x \cdot y \end{aligned} \qquad\qquad \begin{aligned} \hat{} &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ x^0 &:= 1 \\ x^{\mathsf{S}n} &:= x \cdot x^n \end{aligned}$$

Exercise 1.2.1 Define functions as follows:

a) A function $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ yielding the minimum of two numbers.

b) A function $\mathsf{N} \to \mathsf{N} \to \mathsf{B}$ testing whether two numbers are equal.

c) A function $\mathsf{N} \to \mathsf{N} \to \mathsf{B}$ testing whether a number is smaller than another number.

Exercise 1.2.2 Rewrite the definition of truncating subtraction such that the primary case analysis is on the second argument.

		$x + 0 = x$	induction x
1		$0 + 0 = 0$	computational equality
2	$\mathsf{IH} : x + 0 = x$	$\mathsf{S}x + 0 = \mathsf{S}x$	conversion
		$\mathsf{S}(x + 0) = \mathsf{S}x$	rewrite IH
		$\mathsf{S}x = \mathsf{S}x$	computational equality

Figure 1.1: Proof diagram for Equation 1.1

1.3 Structural Induction

We will discuss proofs of the equations

$$x + 0 = x \tag{1.1}$$
$$x + \mathsf{S}y = \mathsf{S}(x + y) \tag{1.2}$$
$$x + y = y + x \tag{1.3}$$
$$(x + y) - y = x \tag{1.4}$$

None of the equations can be shown with structural case analysis and computation alone. In each case **structural induction** on numbers is needed. Structural induction strengthens structural case analysis by providing an **inductive hypothesis** in the successor case. Figure 1.1 shows a **proof diagram** for Equation 1.1. The **induction rule** reduces the **initial proof goal** to two **subgoals** appearing in the lines numbered 1 and 2. The subgoals are obtained by structural case analysis and by adding the inductive hypothesis (IH) in the successor case. The inductive hypothesis makes it possible to close the proof of the successor case by conversion and rewriting. A **conversion step** applies computation rules without closing the proof. A **rewriting step** rewrites with an equation that is either assumed or has been established as a lemma. In the example above, rewriting takes place with the inductive hypothesis, an assumption introduced by the induction rule.

We will explain later why structural induction is a valid proof principle. For now we can say that inductive proofs are recursive proofs.

We remark that rewriting can apply an equation in either direction. The above proof of Equation 1.1 can in fact be shortened by one line if the inductive hypothesis is applied from right to left as first step in the second proof goal.

Note that Equations 1.1 and 1.2 are symmetric variants of the defining equations of the addition function '+'. Once these equations have been shown, they can be used for rewriting in proofs.

Figure 1.2 shows a proof diagram giving an inductive proof of Equation 1.4. Note that the proof rewrites with Equation 1.1 and Equation 1.2, assuming that the equations have been proved before.

		$x + y - y = x$	induction y
1		$x + 0 - 0 = x$	rewrite Equation 1.1
		$x - 0 = x$	case analysis x
1.1		$0 - 0 = 0$	comp. equality
1.2		$\mathsf{S}x - 0 = \mathsf{S}x$	comp. equality
2	$\mathsf{IH} : x + y - y = x$	$x + \mathsf{S}y - \mathsf{S}y = x$	rewrite Equation 1.2
		$\mathsf{S}(x + y) - \mathsf{S}y = x$	conversion
		$x + y - y = x$	rewrite IH
		$x = x$	comp. equality

Figure 1.2: Proof diagram for Equation 1.4

One reason for showing inductive proofs as proof diagrams is that proof diagrams explain how one construct proofs in interaction with Coq. With Coq one states the initial proof goal and then enters commands called **tactics** performing the **proof actions** given in the rightmost column of our proof diagrams. The induction tactic displays the subgoals and automatically provides the inductive hypothesis. Except for the initial claim, all the equations appearing in the proof diagrams are displayed automatically by Coq, saving a lot of tedious writing. Replay all proof diagrams shown in this chapter with Coq to understand what is going on.

A **proof goal** consists of a **claim** and a list of assumptions called **context**. The proof rules for structural case analysis and structural induction reduce a proof goal to several subgoals. A proof is complete once all subgoals have been closed.

A proof diagram comes with three columns listing assumptions, claims, and proof actions.[1] Subgoals are marked by hierarchical numbers and horizontal lines. Our proof diagrams may be called **have-want-do digrams** since they come with separate columns for assumptions we *have*, claims we *want* to prove, and actions we *do* to advance the proof.

Exercise 1.3.1 Give a proof diagram for Equation 1.2. Follow the layout of Figure 1.2.

Exercise 1.3.2 Shorten the given proofs for Equations 1.1 and 1.4 by applying the inductive hypothesis from right to left thus avoiding the conversion step.

Exercise 1.3.3 Prove $x + y - x = y$.

Exercise 1.3.4 Prove that addition is associative: $(x + y) + z = x + (y + z)$. Give a proof diagram.

[1] For now our proof diagrams just have the inductive hypothesis as assumption but this will change as soon as we prove claims with implication, see Chapter 3.

Exercise 1.3.5 Prove the distributivity law $(x+y)\cdot z = x\cdot z + y\cdot z$. You will need associativity of addition.

Exercise 1.3.6 Prove that addition is commutative (Equation 1.3). You will need Equation 1.1 and 1.2 as lemmas.

Exercise 1.3.7 Prove that multiplication is commutative. You will need lemmas.

Exercise 1.3.8 Define a maximum function $M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ and prove

$$M(x+y)x = x+y \quad\text{and}\quad Mx(x+y) = x+y$$

Try to prove $Mxy = Myx$ (commutativity) by induction on x and notice that the inductive hypothesis must be strengthened to $\forall y.\, Mxy = Myx$ for the proof to go through. This strengthening of the inductive hypothesis is logically admissible. Proofs with quantified inductive hypotheses will be discussed in detail in Section 6.5.

1.4 Ackermann Function

The following equations specify a function $A : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ known as **Ackermann function**:

$$\begin{aligned} A0y &= \mathsf{S}y \\ A(\mathsf{S}x)0 &= Ax1 \\ A(\mathsf{S}x)(\mathsf{S}y) &= Ax(A(\mathsf{S}x)y) \end{aligned}$$

As is, the equations cannot serve as a definition since the recursion is not structural in either the first or the second argument. The problem is with the nested recursive application $A(\mathsf{S}x)y$ in the third equation.

However, we can define a structurally recursive function satisfying the given equations. The trick is to use a **higher-order** auxiliary function:[2]

$$\begin{aligned} A' &: (\mathsf{N}\to\mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\ A'h0 &:= h1 \\ A'h(\mathsf{S}y) &:= h(A'hy) \end{aligned} \qquad\qquad \begin{aligned} A &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ A0 &:= \mathsf{S} \\ A(\mathsf{S}x) &:= A'(Ax) \end{aligned}$$

Verifying that A satisfies the three specifying equations is straightforward. Here is a verification of the third equation:

$$\begin{aligned} & A(\mathsf{S}x)(\mathsf{S}y) \\ =\; & A'(Ax)(\mathsf{S}y) \\ =\; & Ax(A'(Ax)y) \end{aligned} \qquad\qquad \begin{aligned} & Ax(A(\mathsf{S}x)y) \\ =\; & Ax(A'(Ax)y) \\ & \end{aligned}$$

[2]A higher-order function is a function taking a function as argument.

Note that the three specifying equations all hold by computation (i.e., both sides of the equations reduce to the same term). Thus verifying the equations with Coq is trivial.

The three equations specifying A are exhaustive and disjoint. They are also terminating, which can be seen with a lexical argument: Either the first argument is decreased, or the first argument stays unchanged and the second argument is decreased.

Recall that Coq only admits total functions. If we define a function with equations, three properties must be satisfied: The equations must be exhaustive and disjoint, and if there is recursion, the recursion must be structural for one of the arguments of the function. All three conditions are checked automatically.

Consider the equations

$$\begin{aligned} E(0) &= \mathsf{T} \\ E(1) &= \mathsf{F} \\ E(\mathsf{S}(\mathsf{S}n)) &= E(n) \end{aligned}$$

specifying a function $E : \mathsf{N} \to \mathsf{B}$ that checks whether its argument is even. The recursion appearing in the third argument is not structural. We can define a structurally recursive function satisfying the three equations as follows:

$$\begin{aligned} E(0) &= \mathsf{T} \\ E(\mathsf{S}n) &= !E(n) \end{aligned}$$

The first and the second specifying equation hold by computation. The third specifying equation holds by conversion and rewriting with $!!b = b$.

We remark that Coq accepts the three specifying equations as definition for E. Coq in fact accepts recursive definitions satisfying a **guard condition** generalising structural recursion. We will stick to structural recursion throughout this text since it turns out that all functions that matter can be defined with just structural recursion as basic recursion. Later we will see a technique that reduces general terminating recursion to structural recursion.

Exercise 1.4.1 Prove $\mathsf{E}(n \cdot 2) = \mathsf{T}$.

1.5 Pairs and Polymorphic Functions

We have seen that booleans and numbers can be accommodated in Coq with inductive types. We will now see that (ordered) pairs (x, y) can also be accommodated with an inductive type definition.

A pair (x, y) combines two values x and y into a single value such that the components x and y can be recovered from the pair. Moreover, two pairs are equal if and only if they have the same components. Thus we have $(3, 2+3) = (1+2, 5)$ and $(1, 2) \neq (2, 1)$.

Pairs whose components are numbers can be accommodated with the inductive definition

$$\begin{aligned} \mathsf{Pair} &: \mathbb{T} \\ \mathsf{pair} &: \mathsf{N} \to \mathsf{N} \to \mathsf{Pair} \end{aligned}$$

A function swapping the components of a pair can now be defined with a single equation:

$$\begin{aligned} &\mathsf{swap} : \mathsf{Pair} \to \mathsf{Pair} \\ &\mathsf{swap}\,(\mathsf{pair}\; x\; y) \;:=\; \mathsf{pair}\; y\; x \end{aligned}$$

Using structural case analysis for pairs, we can prove the equation

$$\mathsf{swap}\,(\mathsf{swap}\, p) \;=\; p$$

for all pairs p (that is, for a variable p of type Pair). Note that structural case analysis on pairs considers only a single case because there is only a single value constructor for pairs.

Above we have defined pairs where both components are numbers. Given two types X and Y we can repeat the definition to obtain pairs whose first component has type X and whose second component has type Y. We can do much better, however, by defining pair types for all component types in one go:

$$\begin{aligned} \times &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{pair} &: \forall X Y.\, X \to Y \to X \times Y \end{aligned}$$

This inductive definition gives us a type constructor '$\times$' for **product types** $X \times Y$ and a **polymorphic value constructor** for pairs. The value constructor comes with a **polymorphic function type** saying that pair takes four arguments, where the first argument X and the second argument Y are types fixing the types of the third and the fourth argument. We can write **partial applications** of the value constructor pair:

$$\begin{aligned} \mathsf{pair}\,\mathsf{N} &: \forall Y.\, \mathsf{N} \to Y \to \mathsf{N} \times Y \\ \mathsf{pair}\,\mathsf{N}\,\mathsf{B} &: \mathsf{N} \to \mathsf{B} \to \mathsf{N} \times \mathsf{B} \\ \mathsf{pair}\,\mathsf{N}\,\mathsf{B}\,0 &: \mathsf{B} \to \mathsf{N} \times \mathsf{B} \\ \mathsf{pair}\,\mathsf{N}\,\mathsf{B}\,0\,\mathsf{T} &: \mathsf{N} \times \mathsf{B} \end{aligned}$$

We can also define a **polymorphic swap function** serving all pair types:

$$\begin{aligned}&\mathsf{swap} : \forall X\,Y.\, X \times Y \to Y \times X\\ &\mathsf{swap}\ X\ Y\ (\mathsf{pair}\,\_\,\_\ x\ y) \ := \ \mathsf{pair}\ Y\ X\ y\ x\end{aligned}$$

Note that the first two arguments of pair in the left hand side of the defining equation are given with the **wildcard symbol** _. The reason for this device is that the first two arguments of pair are **parameter arguments** that don't contribute relevant information in the left hand side of a defining equation.

1.6 Implicit Arguments

If we look at the type of the polymorphic pair constructor

$$\mathsf{pair} : \forall X\,Y.\, X \to Y \to X \times Y$$

we see that the first and second argument of pair are the types of the third and fourth argument. This means that the first and second argument can be derived from the third and fourth argument. This fact can be exploited in Coq by declaring the first and second argument of pair as **implicit arguments**. Implicit arguments are not written explicitly but are derived and inserted automatically. This way we can write pair 0 T for pair N B 0 T. If in addition we declare the type arguments of

$$\mathsf{swap} : \forall X\,Y.\ X \times Y \to Y \times X$$

as implicit arguments, we can write

$$\mathsf{swap}\,(\mathsf{swap}\,(\mathsf{pair}\ x\ y)) \ = \ \mathsf{pair}\ x\ y$$

for the otherwise bloated equation

$$\mathsf{swap}\ Y\,X\,(\mathsf{swap}\ X\,Y\,(\mathsf{pair}\ X\,Y\ x\ y)) \ = \ \mathsf{pair}\ X\,Y\ x\ y$$

We will routinely use implicit arguments for polymorphic constructors and functions in this text.

With implicit arguments, we go one step further and use the standard notations for pairs:

$$(x,y) \ := \ \mathsf{pair}\,x\,y$$

With this final step we can write the definition of swap as follows:

$$\begin{aligned}&\mathsf{swap} : \forall X\,Y.\, X \times Y \to Y \times X\\ &\mathsf{swap}\ (x,y) \ := \ (y,x)\end{aligned}$$

Note that it took us considerable effort to recover the usual mathematical notation for pairs in the typed setting of Coq. There were three successive steps:

1. Polymorphic function types and functions taking types as arguments. We remark that types are first-class values in Coq.
2. Implicit arguments so that type arguments can be derived automatically from other arguments.
3. The usual notation for pairs.

Finally, we define two functions providing the first and the second **projection** for pairs:

$$\begin{array}{ll} \pi_1 : \forall X\,Y.\ X \times Y \to X & \qquad \pi_2 : \forall X\,Y.\ X \times Y \to Y \\ \pi_1\,(x, y) := x & \qquad \pi_2\,(x, y) := y \end{array}$$

We can now prove the **η-law** for pairs

$$(\pi_1 a, \pi_2 a) = a$$

by structural case analysis on the variable $a : X \times Y$.

Exercise 1.6.1 Write the η-law and the definitions of the projections without using the notation (x, y) and without implicit arguments.

Exercise 1.6.2 Let a be a variable of type $X \times Y$. Write proof diagrams for the equations $\mathsf{swap}\,(\mathsf{swap}\,a) = a$ and $(\pi_1 a, \pi_2 a) = a$.

1.7 Iteration

If we look at the equations (all following by computation)

$$\begin{aligned} 3 + x &= \mathsf{S}(\mathsf{S}(\mathsf{S}x)) \\ 3 \cdot x &= x + (x + (x + 0)) \\ x^3 &= x \cdot (x \cdot (x \cdot 1)) \end{aligned}$$

we see a common scheme we call **iteration**. In general, iteration takes the form $f^n\,x$ where a step function f is applied n-times to an initial value x. With the notation $f^n\,x$ the equations from above generalize as follows:

$$\begin{aligned} n + x &= \mathsf{S}^n x \\ n \cdot x &= (+x)^n\,0 \\ x^n &= (\cdot\,x)^n\,1 \end{aligned}$$

The partial applications $(+x)$ and $(\cdot\,x)$ supply only the first argument to the functions for addition and multiplication. They yield functions $\mathsf{N} \to \mathsf{N}$, as suggested by the **cascaded function type** $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ of addition and multiplication.

		$n \cdot x = \mathsf{iter}\ (+x)\ n\ 0$	induction n
1		$0 \cdot x = \mathsf{iter}\ (+x)\ 0\ 0$	comp. equality
2	$\mathsf{IH} : n \cdot x = \mathsf{iter}\ (+x)\ n\ 0$	$\mathsf{S}n \cdot x = \mathsf{iter}\ (+x)\ (\mathsf{S}n)\ 0$	conversion
		$x + n \cdot x = x + \mathsf{iter}\ (+x)\ n\ 0$	rewrite IH
		$x + \mathsf{iter}\ (+x)\ n\ 0 = x + \mathsf{iter}\ (+x)\ n\ 0$	comp. equality

Figure 1.3: Correctness of multiplication with iter

We formalize the notation $f^n\,x$ with a polymorphic function:

$$\begin{aligned} &\mathsf{iter} : \forall X.\ (X \to X) \to \mathsf{N} \to X \to X \\ &\mathsf{iter}\ X\ f\ 0\ x \ := \ x \\ &\mathsf{iter}\ X\ f\ (\mathsf{S}n)\ x \ := \ f(\mathsf{iter}\ X\ f\ n\ x) \end{aligned}$$

We will treat X as implicit argument of iter. The equations

$$\begin{aligned} 3 + x &= \mathsf{iter}\ \mathsf{S}\ 3\ x \\ 3 \cdot x &= \mathsf{iter}\ (+x)\ 3\ 0 \\ x^3 &= \mathsf{iter}\ (\cdot\, x)\ 3\ 1 \end{aligned}$$

now hold by computation. More generally, we can prove the following equations by induction on n:

$$\begin{aligned} n + x &= \mathsf{iter}\ \mathsf{S}\ n\ x \\ n \cdot x &= \mathsf{iter}\ (+x)\ n\ 0 \\ x^n &= \mathsf{iter}\ (\cdot\, x)\ n\ 1 \end{aligned}$$

Figure 1.3 gives a proof diagram for the equation for multiplication.

Exercise 1.7.1 Verify the equation $\mathsf{iter}\ \mathsf{S}\ 2 = \lambda x.\mathsf{S}(\mathsf{S}x)$ by computation.

Exercise 1.7.2 Prove $n + x = \mathsf{iter}\ \mathsf{S}\ n\ x$ and $x^n = \mathsf{iter}\ (\cdot\, x)\ n\ 1$ by induction.

Exercise 1.7.3 (Shift) Prove $\mathsf{iter}\ f\ (\mathsf{S}n)\ x = \mathsf{iter}\ f\ n\ (fx)$ by induction.

Exercise 1.7.4 (Factorials) Factorials $n!$ can be computed by iteration on pairs $(k, k!)$. Find a function f such that $(n, n!) = f^n(0, 1)$. Define a factorial function with the equations $0! = 1$ and $(\mathsf{S}n)! = \mathsf{S}n \cdot n!$ and prove $(n, n!) = f^n(0, 1)$ by induction on n.

Exercise 1.7.5 (Even) $\mathsf{iter}\ !\ n\ \mathsf{T}$ tests whether n is even. Prove $\mathsf{iter}\ !\ (n \cdot 2)\ b = b$ and $\mathsf{iter}\ !\ (\mathsf{S}(n \cdot 2))\ b = !b$.

1.8 Notational Conventions

We are using notational conventions common in type theory and functional programming. In particular, we omit parentheses in types and applications relying on the following rules:

$$\begin{array}{rcl} s \to t \to u & \rightsquigarrow & s \to (t \to u) \\ stu & \rightsquigarrow & (st)u \end{array}$$

For the arithmetic operations we assume the usual rules, so $\cdot$ binds before $+$ and $-$, and all three of them are left associative. For instance:

$$x + 2 \cdot y - 5 \cdot x + z \quad \rightsquigarrow \quad ((x + (2 \cdot y)) - (5 \cdot x)) + z$$

1.9 Final Remarks

The pure equational language we have seen in this chapter is a sweet spot in the type-theoretic landscape. With a minimum of luggage we can define interesting functions, explore equational computation, and prove equational properties using structural induction. Higher-order functions, polymorphic functions, and the concomitant types are elegantly accommodated in this equational language.

We have seen how booleans, numbers, and pairs can be accommodated as **inductive data types** using constructors, and how cascaded functions on data types can be defined using equations. Since every defined function must determine a unique result for every argument of its argument type, the equations defining a function are required to be exhaustive and disjoint, and recursion is constrained to be structural on a single argument. This way logically invalid equations like $f\,x = !(f\,x)$ or $f\,\mathsf{T} = \mathsf{T}$ together with $f\,\mathsf{T} = \mathsf{F}$ are excluded.

Here is a list of important technical terms introduced in this chapter:

- Booleans, numbers, pairs, inductive data types
- (Parameterised) inductive type definition, constructors
- Defining equations, computation rules, computational equality
- Exhaustiveness, disjointness, termination of defining equations
- Cascaded function types, partial applications
- Polymorphic function types, implicit arguments
- Structural recursion, structural case analysis
- Structural induction, inductive hypothesis
- Conversion steps, rewriting steps
- Proof digrams, proof goals, subgoals, proof actions (tactics)

2 Computational Primitives

Type theory and Coq are complex constructions providing many layers of abstraction on a minimal logic kernel. Here we will explain the equational definition of functions with computational primitives known as lambda abstractions, recursive abstractions, matches, and plain definitions. We will discuss the accompanying reduction rules (e.g., β-reduction), which compute unique normal forms for well-typed terms. We then define computational equality based on normal forms, α-equivalence, and η-equivalence.

2.1 Computational Definition of Functions

So far, we have defined functions through equations. In Coq, equational definitions of functions are translated into computational definitions using low level primitives. Figure 2.1 shows computational definitions of functions whose equational definitions we have discussed in Chapter 1. Figure 2.1 also shows a computational definition of a function $D : \mathsf{N} \to \mathsf{N}$ doubling its argument. An equational definition of this function looks as follows:

$$\begin{aligned} D\,0 &:= 0 \\ D(\mathsf{S}x) &:= \mathsf{S}(\mathsf{S}(Dx)) \end{aligned}$$

The primitives used in computational definitions are plain definitions, lambda abstractions, matches, and recursive abstractions. We discuss these primitives one by one in the following.

A **plain definition**

$$c^{\tau} := s$$

binds a name c to a term s, where the name c is given the type τ and the term s must have type τ. The binding of c is not visible in s, that is, c cannot be used recursively in s. We say that a plain definition $c^{\tau} := s$ defines a **constant** c. In practice, the type τ may be omitted, in which case it will be assumed as the type of the term s defining c.

$$
\begin{aligned}
! &:= \lambda x^{\mathsf{B}}.\ \textsc{match}\ x\ [\,\mathsf{T} \Rightarrow \mathsf{F} \mid \mathsf{F} \Rightarrow \mathsf{T}\,] \\
+ &:= \textsc{fix}\ f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}\ x^{\mathsf{N}}.\ \lambda y^{\mathsf{N}}.\ \textsc{match}\ x\ [\,0 \Rightarrow y \mid \mathsf{S}x' \Rightarrow \mathsf{S}(fx'y)\,] \\
- &:= \textsc{fix}\ f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}\ x^{\mathsf{N}}.\ \lambda y^{\mathsf{N}}.\ \textsc{match}\ x\ [\,0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \textsc{match}\ y\ [\,0 \Rightarrow x \mid \mathsf{S}y' \Rightarrow fx'y'\,]\,] \\
\mathsf{swap} &:= \lambda X^{\mathbb{T}}.\ \lambda Y^{\mathbb{T}}.\ \lambda p^{X\times Y}.\ \textsc{match}\ p\ [\,(x,y) \Rightarrow (y,x)\,] \\
D &:= \textsc{fix}\ f^{\mathsf{N}\to\mathsf{N}}\ x^{\mathsf{N}}.\ \textsc{match}\ x\ [\,0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(fx'))\,]
\end{aligned}
$$

Figure 2.1: Computational definitions of functions

A **lambda abstraction**

$$\lambda x^{\tau}.s$$

describes a function that for an argument x of type τ returns the value described by the term s. The **argument variable** x usually appears in the **body** s. In practice, the type τ of the argument variable may be omitted if it is clear from the body.

A **recursive abstraction**

$$\textsc{fix}\ f^{\sigma\to\tau}\ x^{\sigma}.\ s$$

describes a recursive function f taking an argument x. The variable f is not visible outside the recursive abstraction. The argument type σ must be an inductive type and the recursion must be on x. The types of the variables f and x may be omitted if they can be derived (as is the case in the examples in Figure 2.1). In Coq slang, recursive abstractions are often called fixpoints.

A recursive abstraction can take several arguments, where the recursive argument is always the last argument. Extra arguments preceding the recursive argument are needed so that dependently typed recursive functions can be defined, something we will need in later chapters (Exercise 12.1.7).

A **match**

$$\textsc{match}\ s\ [\,cx_1 \dots x_n \Rightarrow t \mid \cdots\,]$$

describes a structural case analysis on the value of a term s, which must have an inductive type. For every value constructor c of the inductive type a **rule** $cx_1 \dots x_n \Rightarrow t$ must be given, where the variables in the **pattern** $cx_1 \dots x_n$ must be distinct. A match realizes an exhaustive and disjoint case analysis.

Computational definitions of functions not using any syntactic convenience are called **kernel definitions**. While Coq provides many conveniences for the definition of functions, it translates every function definition into a kernel definition using only the computational primitives we have seen in this section.

$$\begin{array}{lll} !\mathsf{T} & \succ\ (\lambda x.\ \textsc{match}\ x\ [\mathsf{T} \Rightarrow \mathsf{F} \mid \mathsf{F} \Rightarrow \mathsf{T}])\ \mathsf{T} & \text{unfolding of } ! \\ & \succ\ \textsc{match}\ \mathsf{T}\ [\mathsf{T} \Rightarrow \mathsf{F} \mid \mathsf{F} \Rightarrow \mathsf{T}] & \beta\text{-reduction} \\ & \succ\ \mathsf{F} & \text{match reduction} \end{array}$$

Figure 2.2: Reduction chain for $!\mathsf{T}$

$$\begin{array}{lll} D(\mathsf{S}0) & \succ\ (\textsc{fix}\ f\ x.\ \textsc{match}\ x\ [\,0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(fx'))\,])\ (\mathsf{S}0) & \delta \\ & =\ \hat{D}\ (\mathsf{S}0) & \\ & \succ\ (\lambda fx.\ \textsc{match}\ x\ [0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(fx'))])\ \hat{D}\ (\mathsf{S}0) & \textsc{fix} \\ & \succ\ (\lambda x.\ \textsc{match}\ x\ [0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(\hat{D}x'))])\ (\mathsf{S}0) & \beta \\ & \succ\ \textsc{match}\ (\mathsf{S}0)\ [0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(\hat{D}x'))] & \beta \\ & \succ\ (\lambda x'.\ \mathsf{S}(\mathsf{S}(\hat{D}x')))\ 0 & \textsc{match} \\ & \succ\ \mathsf{S}(\mathsf{S}(\hat{D}0)) & \beta \\ & \succ\ \mathsf{S}(\mathsf{S}((\lambda x.\ \textsc{match}\ x\ [0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(\hat{D}x'))])\ 0)) & \textsc{fix},\ \beta \\ & \succ\ \mathsf{S}(\mathsf{S}(\textsc{match}\ 0\ [0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(\hat{D}x'))])) & \beta \\ & \succ\ \mathsf{S}(\mathsf{S}0) & \textsc{match} \end{array}$$

where $\hat{D}$ is the term defining D

Figure 2.3: Reduction chain for $D(\mathsf{S}0)$

Exercise 2.1.1 Translate the equational definitions of the functions asked for in Exercise 1.2.1 into kernel definitions.

2.2 Reduction Rules

Computation is performed through **reduction rules** for defined constants, lambda abstractions, matches, and recursive abstractions. Figures 2.2 and 2.3 show examples for reduction chains obtained with the reduction rules.

The **reduction rule for defined constants**

$$c \succ s \qquad\qquad \text{provided } c := s$$

is called **δ-reduction** and replaces a constant with the term defining it. One also speaks of **unfolding** of c.

The **reduction rule for lambda abstractions**

$$(\lambda x.s)\,t \;\succ\; s^x_t$$

is called **β-reduction** and replaces an application $(\lambda x.s)t$ with the term s^x_t obtained from the term s by replacing every free occurence of the variable x with the term t. Terms of the form $(\lambda x.s)\,t$ are called **β-redexes**.

The **reduction rule for matches**

$$\textsc{match}\; cs\; [\;\cdots\; cx \Rightarrow t\; \cdots\;] \;\succ\; (\lambda x.t)s$$

replaces a match on cs with an application applying the body of the rule selected by the constructor c to s. The scheme is given here for single argument constructors, the generalization to no argument and several arguments is straightforward.

The **reduction rule for recursive abstractions**

$$(\textsc{fix}\, f x.\, s)\, t \;\succ\; (\lambda f.\lambda x.\, s)\,(\textsc{fix}\, f x.\, s)\, t$$
$$\text{provided } t \text{ is an application of a constructor}$$

reduces an application of a recursive abstraction to an application passing the recursive abstraction as an argument. The constraint that the argument term t is an application of a constructor is essential so that application of the reduction rules terminates.

Coq implements the rule for recursive abstractions such that it includes the β-reduction needed for passing down the recursive abstraction:

$$(\textsc{fix}\, f x.\, s)\, t \;\succ\; (\lambda x.\, s^f_{\textsc{fix}\, f x.s})\, t$$
$$\text{provided } t \text{ is an application of a constructor}$$

Coq's computational primitives also include **let expressions**

$$\textsc{let}\; x^T = s\; \textsc{in}\; t$$

providing for local definitions. The reduction rule for let expressions

$$\textsc{let}\; x = s\; \textsc{in}\; t \;\succ\; t^x_s$$

is called **ζ-rule**.

The reduction rules are computation rules at a low level. While Coq routinely performs reductions at this level, this is not feasible for humans. However, humans can simulate low level reductions with high-level reductions rewriting with the defining equations of functions. For instance,

$$\mathsf{S}(\mathsf{S}x) + y = \mathsf{S}(\mathsf{S}x + y) = \mathsf{S}(\mathsf{S}(x + y))$$

is a high-level reduction chain that applies the second defining equation of $+$ twice. The high-level reduction chain expands into a low level reduction chain with many intermediate steps where the second and third occurence of $+$ will be unfolded. Altogether, the low-level reduction takes 12 steps (1 delta reduction, 2 fix reductions, 2 match reductions, 6 beta reductions).

Given that one can simulate and verify low-level reductions with Coq, it will not be necessary to discuss the reduction rules in more detail.

Exercise 2.2.1 Write the reduction chain for $1 + y$ in the style of Figure 2.3. Verify your reduction steps with Coq.

Exercise 2.2.2 Write the reduction chain for $\mathsf{swap}\ X\ Y\ (\mathsf{pair}\ X\ Y\ x\ y)$ in the style of Figure 2.3. Verify your reduction steps with Coq.

2.3 Well-Typed Terms and Normal Forms

Coq and its type theory come with a typing discipline admitting only **well-typed terms**. The reduction rules and the typing discipline are designed such that application of the reduction rules to a well-typed term always terminates. Thus one can simplify every term to a **normal form** to which no reduction rule applies. The reduction rules are designed such that normal forms are unique. Terms to which no reduction rule applies are also called **normal**.

Terms denote values and reduction simplifies terms such that the value of a term is left unchanged. We may say that **reduction preserves values**.

Reduction also preserves types. That is, if we reduce a term of type τ, we always get terms of type τ.

It is important that logical reasoning only involves well-typed terms. Coq guarantees through type checking that only well-typed terms are involved.

2.4 Computational Equality

Abstractions, matches, and lets involve **bound variables** that are local to the terms introducing them. The names of bound variables do not matter. Two terms are **α-equivalent** if they are equal up to renaming of bound variables. For instance, $\lambda x^{\mathsf{N}}.x$ and $\lambda y^{\mathsf{N}}.y$ are α-equivalent.

For lambda abstractions there is also the notion of **η-equivalence**. Suppose the term s describes a function $\sigma \to \tau$. Then the term $\lambda x^{\sigma}.sx$ describes the same function as the term s, provided the variable x does not occur free in s. We say that the terms $\lambda x^{\sigma}.sx$ and s are **η-equivalent** and call the resulting equivalence relation on terms η-equivalence. The equation $(\lambda x.\mathsf{S}x) = \mathsf{S}$ holds by η-equivalence.

Two terms are **computationally equal** if and only if their normal forms are equal up to α-equivalence and η-equivalence. Computational equality is an algorithmically decidable equivalence relation. Proofs of computational equality are routine checks needing no further explanation.

Computational equality is **compatible with the term structure**. That is, if we replace a subterm of a term s with a term that is computationally equal and has the same type, we obtain a term that is computationally equal to s.

We also say that two terms are **convertible** if they are computationally equal. This makes the connection to the conversion steps appearing in Chapter 1.

The most complex operation the reduction rules build on is **substitution** s_t^x. Substitution is needed for β-reduction and must be performed such that local binders do not capture free variables. To make this possible, substitution must be allowed to rename local variables. For instance, $(\lambda x.\lambda y.fxy)y$ must not reduce to $\lambda y.fyy$ but to a term $\lambda z.fyz$ where the new bound variable z avoids capture of the variable y. We speak of **capture-free substitution**.

We mention that computational equality is also known as *definitional equality*.

Exercise 2.4.1 Verify that the following equations hold by computational equality.

a) $(+)1 = \mathsf{S}$

b) $(+)2 = \lambda x.\,\mathsf{S}(\mathsf{S}x)$

c) $(+)(3-2) = \mathsf{S}$

d) $(\lambda x.\,1+x) = \mathsf{S}$

e) $(\lambda x.\,3+x-2) = \mathsf{S}$

f) $\mathsf{iter}\ \mathsf{S}\ 2 = \lambda x.\,\mathsf{S}(\mathsf{S}x)$

Note that all right hand sides are normal terms. Thus it suffices to compute the normal forms of the left hand sides and then check whether the two normal forms are equal up to α- and η-equivalence.

2.5 Canonical Terms and Values

We use **terms** as syntactic descriptions of **semantic objects**. Semantic objects include booleans, numbers, functions, and types. We often talk about semantic objects ignoring their syntactic representation as terms. In an implementation, however, semantic objects are always represented through syntactic descriptions.

As syntactic objects, terms may not be well-typed. Ill-typed terms are semantically meaningless and must not be used for logical reasoning. Ill-typed terms are always rejected by Coq. Working with Coq is the best way to develop a reliable intuition for what goes through as well-typed. When we say term in these notes, we always mean well-typed terms.

A term is **closed** if it has no free variables (bound variables introduced by abstractions and matches are fine). A term is **canonical** if it is both normal and closed.

Coq's type theory is designed such that every canonical term is either a constructor, or a constructor applied to canonical terms, or an abstraction (obtained with λ or FIX), or a function type (obtained with $\to$ or $\forall$), or a universe (we have seen $\mathbb{T}$). Moreover, every closed term reduces to a canonical term.

Semantic objects that can be described through canonical terms are called **values**. The **inhabitants** of a type are the values that can be described through canonical terms of this type. For data types such as B, N, and products of data types, the canonical terms are in one-to-one correspondence with the inhabitants, and we may think of the inhabitants as canonical terms if we wish. For function types the situation is more complicated since different canonical abstractions may represent the same function, for instance, if they are equal up to α- and η-equivalence. So for function types we still know that every inhabitant can be described through a canonical term, but there are usually many different canonical terms describing the same function. In any case, computationally equal canonical terms always describe the same value.

Reduction preserves well-typedness and closedness of a term as well as its type and value. Since the values of a data type may be seen as the canonical terms of the data type, we may say that reduction computes the values of closed terms whose type is a data type. For instance, the term $2 + 3$ reduces to 5.

We call a type **inhabited** if it has at least one inhabitant. The data types we have seen so far are all inhabited. Later we will use types as logical descriptions and uninhabited types will become a regular option.

The inhabitants of a type may also be referred to as the values or **members** or **elements** of a type.

Syntactic objects can be formalised and realised with software, as in the proof assistant Coq. In contrast, semantic objects are objects of our mathematical intuition that are only realised through their syntactic descriptions.

2.6 Notational Conventions

We omit parentheses and λ's relying on two basic rules:

$$\begin{aligned} \lambda x.st &\rightsquigarrow \lambda x.(st) \\ \lambda xy.s &\rightsquigarrow \lambda x.\lambda y.s \end{aligned}$$

To specify the type of a variable or constant, we use one of the notations $x : \tau$ and x^τ, depending on what we feel is more readable. We usually omit the type of a variable if it is clear from the context.

Following Coq, we may write boolean matches with the familiar if-then-else notation:

$$\textsc{if}\ s\ \textsc{then}\ t_1\ \textsc{else}\ t_2 \quad\rightsquigarrow\quad \textsc{match}\ s\ [\,\mathsf{T} \Rightarrow t_1 \mid \mathsf{F} \Rightarrow t_2\,]$$

More generally, we may use the if-then-else notation for all inductive types with exactly two value constructors, exploiting the order of the constructors.

A similar notational device using the let notation is available for inductive types with exactly one constructor. For instance:

$$\textsc{let}\ (x,y) = s\ \textsc{in}\ t \quad\rightsquigarrow\quad \textsc{match}\ s\ [\,\mathsf{pair}\,\_\,\_\,x\,y \Rightarrow t\,]$$

3 Propositions as Types

Coq represents propositions (i.e., logical statements) as types such that the inhabitants of a propositional type serve as proofs of the represented proposition. This type-theoretic approach to logic works amazingly well in practice. It reduces proof checking to type checking and provides a form of logical reasoning known as intuitionistic reasoning.

In this chapter we study the type-theoretic representations of the propositional connectives conjunction, disjunction, implication, and negation. Quantifiers and equality will be considered in later chapters. We use proof diagrams to assist the construction of proof terms for propositions. This way the construction of a proof amounts to the construction of a proof diagram. The construction of a proof diagram is an incremental process that can be carried out efficiently in interaction with the Coq proof assistant.

3.1 Propositions Informally

Proposition are logical statements whose truth or falsity can be established with proofs. Propositions are built from *basic propositions* with *connectives* and *quantifiers*. Here are prominent forms of propositions you will have encountered before.

Name	Notation	Reading
equality	$s = t$	s equals t
truth	$\top$	true
falsity	$\bot$	false
conjunction	$P \land Q$	P and Q
disjunction	$P \lor Q$	P or Q
implication	$P \to Q$	if P then Q
negation	$\neg P$	not P
equivalence	$P \leftrightarrow Q$	P if and only if Q
universal quantification	$\forall x{:}X.\, px$	for all x in X, px
existential quantification	$\exists x{:}X.\, px$	for some x in X, px

3.2 Conjunction, Disjunction, and Implication

Coq represents propositions with **propositional types** that live in a **universe** $\mathbb{P}$ (read Prop). Given a propositional type X, the terms of type X serve as **proofs** of the proposition represented by X. This straightforward design gives us in one go a formalization of propositions, proofs, and provability.

To ease our language, we call propositional types **propositions** in the following. If we want to talk about informal propositions, we will say so explicitly. A proposition is **provable** if it has an inhabitant.

We accommodate **conjunctions** $X \wedge Y$ and **disjunctions** $X \vee Y$ of two propositions X and Y with two inductive definitions:

$$\begin{aligned} \wedge &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\ \mathsf{C} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ X \to Y \to X \wedge Y \end{aligned} \qquad\qquad \begin{aligned} \vee &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\ \mathsf{L} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ X \to X \vee Y \\ \mathsf{R} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ Y \to X \vee Y \end{aligned}$$

With the constructors '$\wedge$' and '$\vee$' we can form conjunctions $X \wedge Y$ and disjunctions $X \vee Y$ from given propositions X and Y. With the polymorphic **proof constructors** C, L, and R we can construct proofs of conjunctions and disjunctions:

- If x is a proof of X and y is a proof of Y, then the term $\mathsf{C}xy$ is a proof of the conjunction $X \wedge Y$.
- If x is a proof of X, then the term $\mathsf{L}x$ is a proof of the disjunction $X \vee Y$.
- If y is a proof of Y, then the term $\mathsf{R}y$ is a proof of the disjunction $X \vee Y$.

Note that we treat the propositional arguments of the polymorphic proof constructors as implicit arguments, something we have seen before with the value constructor for pairs. Since the explicit arguments of the proof constructors for disjunctions determine only one of the two implicit arguments, the other implicit argument needs to be derived from the surrounding context. This works well in practice.

Given two propositions X and Y, we can form the function type $X \to Y$, which again is a proposition. We take propositional function types as representations of **implications**. A proof of an implication $X \to Y$ is thus a function $X \to Y$ that given a proof of X yields a proof of Y. This gives us a computational semantics for implications working well for logical reasoning.

3.3 Normal Proofs

A proof of a proposition is called **normal** if it is a normal term. In this capter we will mostly construct normal proofs. Figure 3.1 shows a series of provable propositions accompanied by normal proofs. The propositions formulate familiar logical laws. Note that we supply as subscripts the implicit arguments of the proof constructors

$X \to X$	$\lambda x.x$
$X \to Y \to X$	$\lambda xy.x$
$X \to Y \to Y$	$\lambda xy.y$
$(X \to Y \to Z) \to (Y \to X \to Z)$	$\lambda fyx.fxy$
$X \to Y \to X \wedge Y$	C_{XY}
$X \wedge Y \to X$	$\lambda h.\ \textsc{match}\ h\ [\,\mathsf{C}xy \Rightarrow x\,]$
$X \wedge Y \to Y$	$\lambda h.\ \textsc{match}\ h\ [\,\mathsf{C}xy \Rightarrow y\,]$
$X \wedge Y \to Y \wedge X$	$\lambda h.\ \textsc{match}\ h\ [\,\mathsf{C}xy \Rightarrow \mathsf{C}yx\,]$
$X \to X \vee Y$	L_{XY}
$Y \to X \vee Y$	R_{XY}
$X \vee Y \to Y \vee X$	$\lambda h.\ \textsc{match}\ h\ [\,\mathsf{L}x \Rightarrow \mathsf{R}_Y x \mid \mathsf{R}y \Rightarrow \mathsf{L}_X y\,]$

The variables X, Y, Z range over propositions.

Figure 3.1: Propositions with normal proofs

C, L, and R when we think it is helpful. We don't give the types of the argument variables of the lambda abstractions since they are obvious from the propositions on the left.

Figure 3.2 shows normal proofs involving matches with nested patterns. Matches with **nested patterns** are a notational convenience for nested plain matches. For instance, the match

$$\textsc{match}\ h\ [\,\mathsf{C}(\mathsf{C}xy)z \Rightarrow \mathsf{C}x(\mathsf{C}yz)\,]$$

with the nested pattern $\mathsf{C}(\mathsf{C}xy)z$ translates into the plain match

$$\textsc{match}\ h\ [\,\mathsf{C}az \Rightarrow \textsc{match}\ a\ [\,\mathsf{C}xy \Rightarrow \mathsf{C}x(\mathsf{C}yz)\,]\,]$$

nesting a second plain match.

We have arrived at a logical system that is quite interesting. Stepping back from the details, one may ask whether the type-theoretic representation of propositions and proofs is adequate, that is, whether all provable propositions are in fact logically valid (*soundness*), and whether enough logically valid propositions are provable (*completeness*). Here logical validity is used as an informal notion not coming with a rigorous mathematical definition. As it comes to the soundness question, we can say that type theory is explicitly designed such that the propositions as types approach is sound. As it comes to the completeness question, there are no straightforward answers and we prefer to postpone a discussion.

$$(X \wedge Y) \wedge Z \to X \wedge (Y \wedge Z)$$
$$\lambda h.\, \textsc{match}\ h\ [\, \mathsf{C}(\mathsf{C}xy)z \Rightarrow \mathsf{C}x(\mathsf{C}yz)\,]$$

$$(X \vee Y) \vee Z \to X \vee (Y \vee Z)$$
$$\lambda h.\, \textsc{match}\ h\ [\, \mathsf{L}(\mathsf{L}x) \Rightarrow \mathsf{L}x \mid \mathsf{L}(\mathsf{R}y) \Rightarrow \mathsf{R}(\mathsf{L}y) \mid \mathsf{R}z \Rightarrow \mathsf{R}(\mathsf{R}z)\,]$$

$$X \wedge (Y \vee Z) \to (X \wedge Y) \vee (X \wedge Z)$$
$$\lambda h.\, \textsc{match}\ h\ [\, \mathsf{C}x(\mathsf{L}y) \Rightarrow \mathsf{L}(\mathsf{C}xy) \mid \mathsf{C}x(\mathsf{R}z) \Rightarrow \mathsf{R}(\mathsf{C}xz)\,]$$

Figure 3.2: Normal proofs with nested patterns

We summarize the basic intuitions behind the normal proofs we have seen in this section:

- A proof of a conjunction $X \wedge Y$ is a pair consisting of a proof of X and a proof of Y.
- A proof of a disjunction $X \vee Y$ is either a proof of X or a proof of Y.
- A proof of an implication $X \to Y$ is a function that given a proof of X returns a proof of Y.

Exercise 3.3.1 Elaborate the normal proofs in Figure 3.2 such that they use nested plain matches. Moreover, annote the implicite arguments of L and R that must be derived from the surrounding context.

3.4 Propositional Equivalence

We capture **propositional equivalence** with the notation

$$X \leftrightarrow Y \;:=\; (X \to Y) \wedge (Y \to X)$$

Thus a propositional equivalence is a conjunction of two implications, and a proof of an equivalence is a pair of two proof-transforming functions. Given a proof of an equivalence $X \leftrightarrow Y$, we can translate every proof of X into a proof of Y, and every proof of Y into a proof of X. Thus we know that X is provable if and only if Y is provable.

Exercise 3.4.1 Give proofs for the equivalences shown in Figure 3.3 formulating well-known properties of conjunction and disjunction.

$X \wedge Y \leftrightarrow Y \wedge X$	$X \vee Y \leftrightarrow Y \vee X$	*commutativity*
$X \wedge (Y \wedge Z) \leftrightarrow (X \wedge Y) \wedge Z$	$X \vee (Y \vee Z) \leftrightarrow (X \vee Y) \vee Z$	*associativity*
$X \wedge (Y \vee Z) \leftrightarrow X \wedge Y \vee X \wedge Z$	$X \vee (Y \wedge Z) \leftrightarrow (X \vee Y) \wedge (X \vee Z)$	*distributivity*
$X \wedge (X \vee Y) \leftrightarrow X$	$X \vee (X \wedge Y) \leftrightarrow X$	*absorption*

Figure 3.3: Equivalence laws for conjunctions and disjunctions

Exercise 3.4.2 Propositional equivalences yield an equivalence relation on propositions that is compatible with conjunction, disjunction, and implication. This high-level speak can be validated by giving proofs for the following propositions:

$X \leftrightarrow X$	reflexivity
$X \leftrightarrow Y \to Y \leftrightarrow X$	symmetry
$X \leftrightarrow Y \to Y \leftrightarrow Z \to X \leftrightarrow Z$	transitivity
$X \leftrightarrow X' \to Y \leftrightarrow Y' \to X \wedge Y \leftrightarrow X' \wedge Y'$	compatibility with $\wedge$
$X \leftrightarrow X' \to Y \leftrightarrow Y' \to X \vee Y \leftrightarrow X' \vee Y'$	compatibility with $\vee$
$X \leftrightarrow X' \to Y \leftrightarrow Y' \to (X \to Y) \leftrightarrow (X' \to Y')$	compatibility with $\to$

3.5 Truth, Falsity, and Negation

We accommodate the propositions **truth** and **falsity** with two inductive definitions:

$$\begin{array}{lcl} \top : \mathbb{P} & \qquad\qquad & \bot : \mathbb{P} \\ \mathsf{I} : \top & & \end{array}$$

By definition, the proposition $\top$ has a single canonical proof I, and the proposition $\bot$ has no canonical proof at all (since it has no proof constructor). This means that the proposition $\bot$ is an empty type.

We now capture **propositional negation** with the notation

$$\neg X \;:=\; X \to \bot$$

Thus a proof of a negation $\neg X$ is a function that given a proof of X yields a proof of $\bot$. Since $\bot$ has no proof, such a function can only be constructed if X has no proof.

We say that we can **disprove** a proposition X if we can prove its negation $\neg X$.

$X \to \neg\neg X$	$\lambda xf.fx$
$X \to \neg X \to Y$	$\lambda xf.\,\textsc{match}\; fx\; [\,]$
$(X \to Y) \to \neg Y \to \neg X$	$\lambda fgx.\,g(fx)$
$\neg X \to \neg\neg\neg X$	$\lambda fg.\,gf$
$\neg\neg\neg X \to \neg X$	$\lambda fx.\,f(\lambda g.gx)$
$\neg\neg X \to (X \to \neg X) \to \bot$	$\lambda fg.\,f(\lambda x.gxx)$
$(X \to \neg X) \to (\neg X \to X) \to \bot$	$\lambda fg.\,\textsc{let}\; x = g(\lambda x.fxx)\; \textsc{in}\; fxx$

Figure 3.4: Proofs for propositions with negations

A logical principle known as **explosion principle** or **ex falso quodlibet** says that from falsity one can derive everything. We can derive the principle with the following normal proof:

$$\bot \to X$$
$$\lambda h.\,\textsc{match}\; h\; [\,]$$

The function takes a proof h of $\bot$ as argument and returns a proof of X. To do so, the function matches on h. Now every rule of the match must yield a proof of X. Since $\bot$ has no constructor, the match has no rule, and hence the typing requirement for the rules is trivially satisfied. One says that it is *vacuously* true that every rule of the match yields a proof of X.

Figure 3.4 shows proofs of propositions involving negation. While checking the proofs, keep in mind that negations $\neg s$ are just abbreviations for implications $s \to \bot$. Note the use of the let expression in the final proof. It introduces a local name x for the term $g(\lambda x.fxx)$ so that we don't have to write it twice. Except for the proof with let all proofs in Figure 3.4 are normal.

Exercise 3.5.1 Give normal proofs for the following propositions:

a) $\neg\bot$

b) $\neg\neg\bot \leftrightarrow \bot$

c) $\neg\neg\top \leftrightarrow \top$

d) $\neg\neg\neg X \;\leftrightarrow\; \neg X$

e) $(X \to \neg\neg Y) \;\leftrightarrow\; (\neg Y \to \neg X)$

f) $\neg(X \leftrightarrow \neg X)$

g) $\neg(X \vee Y) \;\leftrightarrow\; \neg X \wedge \neg Y$

Equivalence (g) is known as **de Morgan law** for disjunction. We don't ask for a proof of the de Morgan law for conjunction since there isn't one using the means we have seen so far.

3.6 Proof Term Construction with Proof Diagrams

The natural direction for proof term construction is top down, in particular as it comes to lambda abstractions and matches. When we construct a proof term top down, we need an information structure keeping track of the types we still have to construct proof terms for and recording the typed variables introduced by surrounding lambda abstractions and rules of matches. It turns out that the proof diagrams we have introduced in Chapter 1 provide the perfect information structure for constructing proof terms.

Here is a proof diagram showing the construction of a proof term for a proposition known as **Russell's law**:

		$\neg(X \leftrightarrow \neg X)$	intros
	$f : X \to \neg X$		
	$g : \neg X \to X$	$\bot$	assert
1		X	apply g
		$\neg X$	intros
	$x : X$	$\bot$	exact fxx
2	$x : X$	$\bot$	exact fxx

The diagram is written top-down beginning with the initial claim. It records the construction of the proof term

$$\lambda h^{X\leftrightarrow\neg X}.\ \textsc{match}\ h\ [\ \mathsf{C}fg \Rightarrow \textsc{let}\ x = g(\lambda x.fxx)\ \textsc{in}\ fxx\ ]$$

for the proposition $\neg(X \leftrightarrow \neg X)$.

Recall that proof diagrams are have-want diagrams that record on the left what we have and on the right what we want. When we start, the proof diagram is **partial** and just consists of the first line. As the proof term construction proceeds, we add further lines and further proof goals until we arrive at a **complete** proof diagram.

The rightmost column of a proof diagram records the actions developing the diagram and the corresponding proof term.

- The action *intros* introduces λ-abstractions and matches.
- The action *assert* creates subgoals for an intermediate claim and the current claim with the intermediate claim assumed. An assert action is realised with a let expression in the proof term.
- The action *apply* applies a function and creates subgoals for the arguments.

		$X \land (Y \lor Z) \leftrightarrow (X \land Y) \lor (X \land Z)$	apply C
1		$X \land (Y \lor Z) \to (X \land Y) \lor (X \land Z)$	intros
	$x : X$		
1.1	$y : Y$	$(X \land Y) \lor (X \land Z)$	$\mathsf{L}(\mathsf{C}xy)$
1.2	$z : Z$	$(X \land Y) \lor (X \land Z)$	$\mathsf{R}(\mathsf{C}xz)$
2		$(X \land Y) \lor (X \land Z) \to X \land (Y \lor Z)$	intros
2.1	$x : X,\ y : Y$	$X \land (Y \lor Z)$	$\mathsf{C}x(\mathsf{L}y)$
2.2	$x : X,\ z : Z$	$X \land (Y \lor Z)$	$\mathsf{C}x(\mathsf{R}z)$

The constructed proof term looks as follows:

$$\begin{aligned}\mathsf{C}\ (\lambda h.\ \textsc{match}\ h\ [\ \mathsf{C}x(\mathsf{L}y) \Rightarrow \mathsf{L}(\mathsf{C}xy) \mid \mathsf{C}x(\mathsf{R}z) \Rightarrow \mathsf{R}(\mathsf{C}xz)])\\ (\lambda h.\ \textsc{match}\ h\ [\ \mathsf{L}(\mathsf{C}xy) \Rightarrow \mathsf{C}x(\mathsf{L}y) \mid \mathsf{R}(\mathsf{C}xz) \Rightarrow \mathsf{C}x(\mathsf{R}z)\ ])\end{aligned}$$

Figure 3.5: Proof diagram for a distributivity law

- The action *exact* proves the claim with a complete proof term. We will not write the word "exact" in future proof diagrams since that an exact action is used will always be clear from the context.

With Coq we can construct proof terms interactively following the structure of proof diagrams. We start with the initial claim and then perform the proof actions using tactics. Coq then maintains the proof goals and displays the assumptions and claims. Once all proof goals are closed, a proof term for the initial claim has been constructed.

Technically, a proof goal consists of a list of assumptions (called context) and a claim. The claim is a type, and the assumptions are typed variables. There may be more than one proof goal open at a point in time and one may navigate freely between open goals.

Interactive proof term construction with Coq is fun since writing, bookkeeping, and verification are done by Coq. Here is a further example of a proof diagram:

	$\neg\neg X \to (X \to \neg X) \to \bot$	intros
$f : \neg\neg x$		
$g : X \to \neg X$	$\bot$	apply f
	$\neg x$	intros
$x : X$	$\bot$	gxx

The proof term constructed is $\lambda fg.f(\lambda x.gxx)$. As announced before, we write the proof action "exact gxx" without the word "exact".

Figure 3.5 gives a proof diagram for a distributivity law involving 6 subgoals. Note the symmetry in the normal proof constructed.

		$\neg\neg(X \to Y) \leftrightarrow (\neg\neg X \to \neg\neg Y)$	apply C, intros
1	$f : \neg\neg(X \to Y)$		
	$g : \neg\neg X$		
	$h : \neg Y$	$\bot$	apply f, intros
	$f' : X \to Y$	$\bot$	apply g, intros
	$x : X$	$\bot$	$h(f'x)$
2	$f : \neg\neg X \to \neg\neg Y$		
	$g : \neg(X \to Y)$	$\bot$	apply g, intros
	$x : X$	Y	exfalso
		$\bot$	apply f
2.1		$\neg\neg X$	intros
	$h : \neg X$	$\bot$	hx
2.2		$\neg Y$	intros
	$y : Y$	$\bot$	$g(\lambda x.y)$

The constructed proof term looks as follows:

$$\begin{array}{l} \mathsf{C}\ (\lambda fgh.\ f(\lambda f'.\ g(\lambda x.\ h(f'x)))) \\ \quad (\lambda fg.\ g(\lambda x.\ \textsc{match}\ f(\lambda h.\ hx)\ (\lambda y.\ g(\lambda x.y))\ [\,])) \end{array}$$

Figure 3.6: Proof diagram for a double negation law using the explosion principle

Figure 3.6 gives a proof diagram for a double negation law. Note the use of the explosion principle in subgoal 2.

Exercise 3.6.1 Give the normal proof obtained with the proof diagram in Figure 3.6.

Exercise 3.6.2 Give proof diagrams for the following propositions:

a) $\neg\neg(X \vee \neg X)$

b) $\neg\neg(\neg\neg X \to X)$

c) $\neg\neg(((X \to Y) \to X) \to X)$

d) $\neg\neg((\neg Y \to \neg X) \to X \to Y)$

Exercise 3.6.3 Give proof diagrams for the following propositions:

a) $\neg\neg(X \vee \neg X)$

b) $\neg(X \vee Y) \ \leftrightarrow\ \neg X \wedge \neg Y$

c) $\neg\neg\neg X \ \leftrightarrow\ \neg X$

d) $\neg\neg(X \wedge Y) \ \leftrightarrow\ \neg\neg X \wedge \neg\neg Y$

e) $\neg\neg(X \to Y) \ \leftrightarrow\ (\neg\neg X \to \neg\neg Y)$

f) $\neg\neg(X \to Y) \ \leftrightarrow\ \neg(X \wedge \neg Y)$

3.7 Notational Issues

Following Coq, we use the precedence order

$$\neg \quad \wedge \quad \vee \quad \leftrightarrow \quad \rightarrow$$

for the logical connectives. Thus we may omit parentheses as in the following example:

$$\neg\neg X \wedge Y \vee Z \leftrightarrow Z \rightarrow Y \quad \rightsquigarrow \quad (((\neg(\neg X) \wedge Y) \vee Z) \leftrightarrow Z) \rightarrow Y$$

The notations $\neg$, $\wedge$, and $\vee$ are right associative. That is, parentheses may be omitted as follows:

$$\begin{aligned} \neg\neg X &\quad \rightsquigarrow \quad \neg(\neg X) \\ X \wedge Y \wedge Z &\quad \rightsquigarrow \quad X \wedge (Y \wedge Z) \\ X \vee Y \vee Z &\quad \rightsquigarrow \quad X \vee (Y \vee Z) \end{aligned}$$

3.8 Type Checking Rules

We have seen that constructing a proof eventually means to construct a term that has the right type. Thus proof checking reduces to type checking, and the exact rules of the type discipline saying which terms have which types are the lowest level proof rules. If the typing rules are too permissive, we can prove propositions that should be unprovable, and if the typing rules are too restrictive, we cannot proof enough.

Here are the type checking rules as we know them so far:

- A lambda abstraction $\lambda x : u.s$ has type $u \rightarrow v$ if u is not a universe and s has type v in a context where x has type u.
- A lambda abstraction $\lambda x : u.s$ has type $\forall x : u.v$ if u is a universe (i.e., $\mathbb{T}$ or $\mathbb{P}$) and s has type v in a context where x has type u.
- An application st has type v if s has type $u \rightarrow v$ and t has type u.
- An application st has type v^x_t if s has type $\forall x : \mathbb{P}.v$ and t has type $\mathbb{P}$.
- A term MATCH s $[\cdots]$ has type u if s is has an inductive type v, the match has a rule for every constructor of v, and every rule of the match yields a result of type u.

3.9 Final Remarks

In this section we have seen that lambda abstractions and matches are essential proof constructs. Without lambda abstractions and matches most of the propositions in Figure 3.1 would be unprovable. We have seen that matches provide for the description of functions that cannot be described otherwise, and that the functions describable with matches often inhabit function types that otherwise would be uninhabited (e.g., $X \wedge Y \to Y \wedge X$).

Note that the functions we can describe with abstractions and matches are controlled by type checking, and that the details of this type checking are important in that they prevent a proof of falsity.

Nowhere in this chapter the reductions coming with lambda abstractions and matches were used. We may say that the proof discipline introduced in this chapter uses the typing discipline of the computational system introduced in Chapter 2 without making use of its computation rules. This will change in the next chapter, where we extend the typing discipline with dependent function types integrating computational equality with type checking.

4 Dependent Function Types

We now generalize polymorphic function types so that one can quantify over every type. The thus obtained dependent function types provide universal quantification for propositions and also subsume simple function types. Dependent function types are accompanied by the conversion law, which relaxes type checking so that computationally equal types become interchangeable.

We also introduce the hierarchy of universes.

With dependent function types and the conversion law we have arrived at an expressive type theory. We will see in later chapters that propositional equality and existential quantification can be defined, and that the proof rules for boolean case analysis and structural induction on numbers can be derived.

The generalisation of function types to dependent function types is the key feature of modern type theories. One often speaks of *dependent type theories* to acknowledge the presence of dependent function types.

4.1 Generalization of Polymorphic Function Types

Consider the types of the proof constructors for conjunctions and disjunctions:

$$\begin{aligned}\mathsf{C}:&\ \forall X^{\mathbb{P}}.\forall Y^{\mathbb{P}}.\ X \to Y \to X \wedge Y\\ \mathsf{L}:&\ \forall X^{\mathbb{P}}.\forall Y^{\mathbb{P}}.\ X \to X \vee Y\\ \mathsf{R}:&\ \forall X^{\mathbb{P}}.\forall Y^{\mathbb{P}}.\ Y \to X \vee Y\end{aligned}$$

These polymorphic function types are in fact propositions. The type of the proof constructor R, for instance, may be read as saying “for all propositions X and Y and every proof of Y there is a proof of $X \vee Y$”. As the notation ’$\forall$’ suggests, propositional polymorphic function types are understood as universal quantifications. Note that the constructors serve as canonical proofs of the propositions given as their types.

Technically, it is straightforward to generalize polymorphic function types to **dependent function types**

$$\forall x\!:\!s.t$$

that can quantify over all types s, not just the two universes $\mathbb{P}$ and $\mathbb{T}$. As with polymorphic types, the inhabitants of a general dependent function type $\forall x\!:\!s.t$

are functions taking arguments of type s and returning results of type t, where t may dependent on the argument x.

If t is a proposition, then every dependent function type $\forall x{:}s.\,t$ is a proposition. As the notation suggests, propositional dependent function types $\forall x{:}s.\,t$ serve as **universal quantifications** $\forall x{:}s.\,t$. Since s can be any type, we can quantify over every type. The propositions as types semantics for universal quantification is just fine since it captures the proofs of a proposition $\forall x{:}s.\,t$ as functions that given a value x yield a proof of the proposition t.

Dependent function types not only subsume polymorphic function types, but also subsume simple function types $s \to t$. In fact, an **simple function type**

$$s \to t$$

is just a dependent function type $\forall x{:}s.\,t$ where the variable x does not appear in t.

As with simple function types, the canonical terms for dependent function types are obtained with abstractions, constructors, and partial applications of constructors.

For dependent function types we use the notational conveniences we have seen before for polymorphic function types:

$$\begin{array}{lclcl} \forall x^s.\,t & \rightsquigarrow & \forall x{:}s.\,t & & \\ \forall xy.\,s & \rightsquigarrow & \forall x\forall y.\,s & \rightsquigarrow & \forall x.\forall y.\,s \end{array}$$

4.2 Impredicative Characterizations

It turns out that quantification over propositions has amazing expressivity. Given two propositional variables X and Y, we can prove the equivalences

$$\begin{array}{rcl} \bot & \leftrightarrow & \forall Z^{\mathbb{P}}.\ Z \\ X \wedge Y & \leftrightarrow & \forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z \\ X \vee Y & \leftrightarrow & \forall Z^{\mathbb{P}}.\ (X \to Z) \to (Y \to Z) \to Z \end{array}$$

which specify $\bot$, $X \wedge Y$, and $X \vee Y$ using polymorphic and simple function types. The equivalences are known as **impredicative characterizations** of falsity, conjunction, and disjunction. Figure 4.1 gives normal proofs for the equivalences. The term impredicative refers to the fact that quantification over all propositions is used.

The equivalences demonstrate that falsity, conjunction, and disjunction can be defined only using dependent function types.

Exercise 4.2.1 Give proof diagrams for the impredicative characterizations.

Exercise 4.2.2 Find an impredicative characterisation for $\top$.

$$\bot \;\leftrightarrow\; \forall Z^{\mathbb{P}}.\, Z$$
$$\mathsf{C}\ (\lambda h.\, \textsc{match}\ h\ [\,])\ (\lambda f.\, f\bot)$$

$$X \wedge Y \;\leftrightarrow\; \forall Z^{\mathbb{P}}.\, (X \to Y \to Z) \to Z$$
$$\mathsf{C}\ (\lambda hZf.\, \textsc{match}\ h\ [\,\mathsf{C}xy \Rightarrow fxy\,])\ (\lambda f.\, f(X \wedge Y)\mathsf{C}_{XY})$$

$$X \vee Y \;\leftrightarrow\; \forall Z^{\mathbb{P}}.\, (X \to Z) \to (Y \to Z) \to Z$$
$$\mathsf{C}\ (\lambda hZfg.\, \textsc{match}\ h\ [\,\mathsf{L}x \Rightarrow fx \mid \mathsf{R}y \Rightarrow gy\,])\ (\lambda f.\, f(X \vee Y)\,\mathsf{L}_{XY}\,\mathsf{R}_{XY})$$

The subscripts give the implicit arguments of C, L, and R.

Figure 4.1: Normal proofs for impredicative characterizations

4.3 Predicates

A **predicate** is a function that after taking enough arguments yields a proposition. Constructors that are predicates are called **inductive predicates**. The constructors '$\wedge$' and '$\vee$' for conjunctions and disjunctions are examples for inductive predicates. Note that the proof constructors for conjunctions and disjunctions are not predicates since the yield proofs rather than propositions.

Let X and Y be types and $p : X \to Y \to \mathbb{P}$ be a predicate. We can prove the equivalence

$$(\forall x \forall y.\, pxy) \;\leftrightarrow\; (\forall y \forall x.\, pxy)$$

formulating a swap law for universal quantifiers with the normal proof

$$\mathsf{C}\ (\lambda fyx.fxy)\ (\lambda fxy.fyx)$$

Using universal quantification, we can internalize the types X and Y and the predicate p:

$$\forall X^{\mathbb{T}} \forall Y^{\mathbb{T}} \forall p^{X \to Y \to \mathbb{P}}.\ (\forall x \forall y.\, pxy) \;\leftrightarrow\; (\forall y \forall x.\, pxy)$$

A normal proof now looks as follows:

$$\lambda XYp.\, \mathsf{C}\ (\lambda fyx.fxy)\ (\lambda fxy.fyx)$$

In fact, this proof is canonical since it is a closed and normal term.

Figure 4.2 shows a proof diagram for a double negation law for the universal quantifier. We remark that the converse of the law cannot be shown.

Figure 4.3 shows a proof diagram for a quantifier law where a **destructuring action** is used to obtain the right-to-left direction of an equivalence proof. This is the first time a destructuring action is used in a proof diagram.

	$\forall X^{\mathbb{T}} \forall p^{X\to\mathbb{P}}.\ \neg\neg(\forall x.\,px) \to \forall x.\,\neg\neg px$	intros
$X:\mathbb{T},\ \ p:X\to\mathbb{P}$		
$f:\neg\neg(\forall x.\,px)$		
$x:X,\ \ g:\neg px$	$\bot$	apply f
	$\neg(\forall x.\,px)$	intros
$f':\forall x.\,px$	$\bot$	$g(f'x)$

Proof term: $\lambda Xpfxg.f(\lambda f'.g(f'x))$

Figure 4.2: Proof diagram for a double negation law

	$\forall X^{\mathbb{T}} \forall p^{X\to\mathbb{P}} \forall q^{X\to\mathbb{P}}.$ $(\forall x.\,px \leftrightarrow qx) \to (\forall x.\,qx) \to \forall x.\,px$	intros
$X:\mathbb{T},\ p:X\to\mathbb{P},\ q:X\to\mathbb{P}$		
$f:\forall x.\,px \leftrightarrow qx$		
$g:\forall x.\,qx$		
$x:X$	px	destruct fx
$h:qx\to px$		$h(gx)$

Proof term: $\lambda Xpqfgx.\,\text{MATCH}\ fx\ [\,\mathsf{C}\_h \Rightarrow h(gx)\,]$

Figure 4.3: Proof diagram using a destructuring action

Exercise 4.3.1 Give a proof diagram and a canonical proof for the distribution law $\forall X^{\mathbb{T}} \forall p^{X\to\mathbb{P}} \forall q^{X\to\mathbb{P}}.\ (\forall x.\ px \wedge qx) \leftrightarrow (\forall x.\,px) \wedge (\forall x.\,qx)$.

Exercise 4.3.2 Find out which direction of the equivalence $\forall X^{\mathbb{T}} \forall Z^{\mathbb{P}}.\ (\forall x^X.\,Z) \leftrightarrow Z$ cannot be proved.

Exercise 4.3.3 Prove $\forall X^{\mathbb{T}} \forall p^{X\to\mathbb{P}} \forall Z^{\mathbb{P}}.\ (\forall x.px) \to Z \to \forall x.\ px \wedge Z$.

Exercise 4.3.4 Give a proof of the proposition in Figure 4.3 using a projection rather than a destructuring action.

4.4 Conversion Law

Recall computational equality of terms (Section 2.4). Computationally equal terms describe the same value. In particular, computationally equal terms that describe types describe the same type. This design is accommodated in the typing discipline by a rule saying that a typing $s : t$ is admitted if t is a term describing a type and there is some computationally equal term t' such that the typing $s : t'$ is admitted. We refer to this basic principle of the typing discipline as the **conversion law**.

Using formal notation, we may write the conversion law as follows:

$$\frac{\vdash s : t' \qquad t \approx t' \qquad \vdash t : \mathbb{P} \ \text{ or } \ \vdash t : \mathbb{T}}{\vdash s : t}$$

The notation $\vdash s : t$ says that the typing $s : t$ is admitted, and the notation $t \approx t'$ says that the terms t and t' are computationally equal.

The conversion law is of particular importance for propositional types since it ensures that provability interacts with computational equality as we would expect it from the examples in Chapter 1. If we search for a proof of a proposition, the conversion law makes it possible to switch to any computationally equal proposition. Several such **conversion steps** can be found in the proof diagrams of Chapter 1, where propositions take the form of equations.

The statements $\vdash s : t$ (typing) and $s \approx t$ (computational equality) appearing in the above rules are called **judgements**. Judgements are used to set up the governing type theory with its term-based notions of well-typedness and computational equality. Judgements appear at the outside of the type theory and are different from propositions appearing as propositional types inside the type theory.

We will see many examples for the use of the conversion law once we have introduced propositional equality. As our first example, however, we consider a proposition known as **Leibniz symmetry** not yet involving propositional equality. Leibniz symmetry for a type X and two inhabitants $x : X$ and $y : X$ is the proposition

$$(\forall p.\ px \to py) \to (\forall p.\ py \to px)$$

quantifying over predicates $p : X \to \mathbb{P}$. Informally, Leibniz symmetry says that whenever a value y satisfies every property a value x satisfies, x also satisfies every property y satisfies.

Figure 4.4 show a proof diagram for Leibniz symmetry involving two conversion steps:

$$\begin{aligned} py \to px \ &\approx\ (\lambda z.\ pz \to px)\,y \\ (\lambda z.\ pz \to px)\,x \ &\approx\ px \to px \end{aligned}$$

The proof term constructed is

$$\lambda fp.\, f(\lambda z.\, pz \to px)(\lambda h.h)$$

The two conversions are implicit in the proof term since they are admitted by the conversion law of the typing discipline.

$X:\mathbb{T},\ x:X,\ y:X$	$(\forall p.\ px \to py) \to (\forall p.\ py \to px)$	intros
$f:\forall p.\ px \to py$		
$p:X \to \mathbb{P}$	$py \to px$	conversion
	$(\lambda z.\ pz \to px)\, y$	apply f
	$(\lambda z.\ pz \to px)\, x$	conversion
	$px \to px$	$\lambda h.h$

Proof term: $\lambda f p.\ f(\lambda z.\ pz \to px)(\lambda h.h)$

Figure 4.4: Proof diagram for Leibniz symmetry

4.5 Negation and Equivalence as Defined Constants

In Chapter 3, we have accommodated negation and equivalence as notations:

$$\begin{aligned} \neg s \; &:= \; s \to \bot \\ s \leftrightarrow t \; &:= \; (s \to t) \land (t \to s) \end{aligned}$$

Now that we have the conversion law, we may also accommodate negation and equivalence as defined constants:

$$\begin{aligned} \neg \; : \; \mathbb{P} \to \mathbb{P} \; &:= \; \lambda X.\, X \to \bot \\ \leftrightarrow \; : \; \mathbb{P} \to \mathbb{P} \to \mathbb{P} \; &:= \; \lambda XY.\, (X \to Y) \land (Y \to X) \end{aligned}$$

If we accommodate negation and equivalence as defined constants, as it is done by Coq, it takes conversion steps to switch between $\neg s$ and $s \to \bot$ or $s \leftrightarrow t$ and $(s \to t) \land (t \to s)$. The conversions steps will involve δ- and β-reductions. Since conversion steps do not show up in proof terms, the proof terms stay unchanged when we switch between the two representations of negation and equivalence.

4.6 Hierarchy of Universes

We have seen the universes $\mathbb{P}$ and $\mathbb{T}$ so far. Universes are types whose inhabitants are types. The universe $\mathbb{P}$ of propositions is accommodated as a subuniverse of the universe of types $\mathbb{T}$, a design written as

$$\mathbb{P} \subseteq \mathbb{T}$$

and being realized with the typing rule

$$\frac{\vdash t : \mathbb{P}}{\vdash t : \mathbb{T}}$$

Types are first class objects in Coq's type theory and first class objects always have a type. So what are the types of $\mathbb{P}$ and $\mathbb{T}$? Giving $\mathbb{T}$ the type $\mathbb{T}$ does not work since this yields a proof of falsity (a nontrivial result). What works, however, is an infinite cumulative hierarchy of universes:

$$\mathbb{T}_1 : \mathbb{T}_2 : \mathbb{T}_3 : \cdots$$
$$\mathbb{P} \subseteq \mathbb{T}_1 \subseteq \mathbb{T}_2 \subseteq \mathbb{T}_3 \subseteq \cdots$$
$$\mathbb{P} : \mathbb{T}_2$$

For dependent function types we have two **closure rules**

$$\frac{s : \mathbb{T}_i \qquad t : \mathbb{P}}{\forall x{:}s.t : \mathbb{P}} \qquad\qquad \frac{s : \mathbb{T}_i \qquad t : \mathbb{T}_1}{\forall x{:}s.t : \mathbb{T}_i}$$

The rule for $\mathbb{P}$ says that the universe of propositions is closed under all quantifications including **big quantifications** quantifying over the types of a universe. In contrast, a dependent function type $\forall x{:}\mathbb{T}_i.t$ where t is not a proposition will not be an inhabitant of the universe $\mathbb{T}_i$ it quantifies over.

The universe $\mathbb{P}$ is called **impredicative** since it is closed under big quantifications. The impredicative characterizations we have seen for falsity, conjunction, disjunctions, and equality exploit this fact.

It is common practice to not give the **universe level** and just write $\mathbb{T}$ for all $\mathbb{T}_i$ as we did so far. This is justified by the fact that the exact universe levels don't matter as long as they can be assigned consistently. Coq ensures during type checking that universe levels can be assigned consistently.

Ordinary inductive types like B, N, N × N, and N → N are placed in the lowest type universe $\mathbb{T}_1$, which is called Set in Coq (a historical name, not related to mathematical sets).

4.7 Type Checking Rules Revisited

Since both simple function types and polymorphic function types are special cases of dependent function types, we can simplify the type checking rules for abstractions and applications given in Section 3.8.

- A lambda abstraction $\lambda x{:}u.s$ has type $\forall x{:}u.v$ if u is a type and s has type v in a context where x has type u.

$$\frac{\vdash u : \mathbb{T} \qquad x{:}u \vdash s : v}{\vdash \lambda x{:}u.s \;:\; \forall x{:}u.v}$$

- An application st has type v^x_t if s has type $\forall x\!:\!u.v$ and t has type u.

$$\frac{\vdash s : \forall x\!:\!u.v \qquad \vdash t : u}{\vdash st \ : \ v^x_t}$$

The type checking rule for matches we have used in this chapter is the one given in Section 3.8. In the next chapter we will see a radical generalization of the typing rule for matches making it possible to derive the rules for structural case analysis and structural induction.

5 Leibniz Equality

We will now see that propositional equality can be defined following a scheme known as Leibniz equality. It turns out that three typed constants suffice: One constant accommodating equations $s = t$ as propositions, one constant providing canonical proofs for trivial equations $s = s$, and one constant providing for rewriting. It suffices to provide the constants as *declared constants* hiding their definitions.

This chapter and the previous chapter introduce much of the technical essence of dependent type theory. Students will need time to understand the material. On the technical side, we see dependent function types, the conversion law, and abstraction by means of declared constants. On the applied side, we see the treatment of propositional equality with declared constants and the concomitant Leibniz definition. There is much elegance and surprise in this chapter.

5.1 Propositional Equality with Three Constants

With dependent function types and the conversion law at our disposal, we can now show how the propositions as types approach can accommodate propositional equality. It turns out that all we need are three typed constants:

$$\begin{aligned}
\mathsf{eq} &: \forall X^{\mathbb{T}}.\ X \to X \to \mathbb{P} \\
\mathsf{Q} &: \forall X^{\mathbb{T}} \forall x.\ \mathsf{eq}\, X\, x\, x \\
\mathsf{R} &: \forall X^{\mathbb{T}} \forall x y \forall p^{X \to \mathbb{P}}.\ \mathsf{eq}\, X x y \to p x \to p y
\end{aligned}$$

The constant eq allows us to write equations as propositional types. We treat X as an implicit argument and write $s = t$ for $\mathsf{eq}\, s\, t$. The constants Q and R provide two basic proof rules for equations. With Q we can prove every trivial equation $s = s$. Given the conversion law, we can also prove with Q every equation $s = t$ where s and t are convertible. In other words, Q provides for proofs by computational equality.

The constant R provides for equational rewriting: Given a proof of an equation $s = t$, we can rewrite a claim pt to a claim ps. Moreover, we can get from an assumption ps an additional assumption pt by asserting pt and rewriting to ps.

We refer to R as **rewriting law**, and to the argument p of R as **rewriting predicate**. Moreover, we refer to the predicate eq as **propositional equality** or just

$\top \neq \bot$	propositional disjointness
$\mathsf{T} \neq \mathsf{F}$	boolean disjointness
$\forall x^{\mathsf{N}}.\ 0 \neq \mathsf{S}x$	disjointness of 0 and S
$\forall x^{\mathsf{N}}\,y^{\mathsf{N}}.\ \mathsf{S}x = \mathsf{S}y \to x = y$	injectivity of successor
$\forall X^{\mathbb{T}}\,Y^{\mathbb{T}}\,f^{X \to Y}\,x\,y.\ x = y \to fx = fy$	applicative closure
$\forall X^{\mathbb{T}}\,x^{X}\,y^{X}.\ x = y \to y = x$	symmetry
$\forall X^{\mathbb{T}}\,x^{X}\,y^{X}\,z^{X}.\ x = y \to y = z \to x = z$	transitivity

Figure 5.1: Basic equational facts

equality. We will treat X, x and y as implicit arguments of R and X as implicit argument of eq and Q.

Exercise 5.1.1 Give a canonical proof for $!\mathsf{T} = \mathsf{F}$. Make all implicit arguments explicit and explain which type checking rules are needed to establish that your proof term has type $!\mathsf{T} = \mathsf{F}$. Explain why the same proof term also proves $\mathsf{F} = !!\mathsf{F}$.

Exercise 5.1.2 Give a term where R is applied to 7 arguments. In fact, for every number n there is a term that applies R to exactly n arguments.

Exercise 5.1.3 Suppose we want to rewrite a subterm u in a proposition t using the rewriting law R. Then we need a rewrite predicate $\lambda x.s$ such that t and $(\lambda x.s)u$ are convertible and s is obtained from t by replacing the occurrence of u with the variable x. Let t be the proposition $x + y + x = y$.

a) Give a predicate for rewriting the first occurrence of x in t.

b) Give a predicate for rewriting the second occurrence of y in t.

c) Give a predicate for rewriting all occurrences of y in t.

d) Give a predicate for rewriting the term $x + y$ in t.

e) Explain why the term $y + x$ cannot be rewritten in t.

5.2 Basic Equational Facts

The constants Q and R give us straightforward proofs for many equational facts. Figure 5.1 shows a collection of basic equational facts, and Figure 5.2 gives proof diagrams and the resulting proof terms for some of them.

Note that the proof diagrams in Figure 5.2 all follow the same scheme: First comes a step introducing assumptions, then a conversion step making the rewriting

	$\top \neq \bot$	intros
$H : \top = \bot$	$\bot$	conversion
	$(\lambda X^{\mathbb{P}}.X)\,\bot$	apply $\mathsf{R}\,\_\,H$
	$(\lambda X^{\mathbb{P}}.X)\,\top$	conversion
	$\top$	I

Proof term: $\lambda H.\ \mathsf{R}_{(\lambda X^{\mathbb{P}}.X)}\ H\ \mathsf{I}$

	$\mathsf{T} \neq \mathsf{F}$	intros
$H : \mathsf{T} = \mathsf{F}$	$\bot$	conversion
	$(\lambda x^{\mathsf{B}}.\ \text{MATCH}\ x\ [\,\mathsf{T} \Rightarrow \top \mid \mathsf{F} \Rightarrow \bot\,])\ \mathsf{F}$	apply $\mathsf{R}\,\_\,H$
	$(\lambda x^{\mathsf{B}}.\ \text{MATCH}\ x\ [\,\mathsf{T} \Rightarrow \top \mid \mathsf{F} \Rightarrow \bot\,])\ \mathsf{T}$	conversion
	$\top$	I

Proof term: $\lambda H.\ \mathsf{R}_{(\lambda x^{\mathsf{B}}.\ \text{MATCH}\ x\ [\mathsf{T}\Rightarrow\top\mid\mathsf{F}\Rightarrow\bot])}\ H\ \mathsf{I}$

$x : \mathsf{N},\ y : \mathsf{N}$	$\mathsf{S}x = \mathsf{S}y \to x = y$	intros
$H : \mathsf{S}x = \mathsf{S}y$	$x = y$	conversion
	$(\lambda z.\ x = \text{MATCH}\ z\ [\,0 \Rightarrow 0 \mid \mathsf{S}z' \Rightarrow z'\,])\ (\mathsf{S}y)$	apply $\mathsf{R}\,\_\,H$
	$(\lambda z.\ x = \text{MATCH}\ z\ [\,0 \Rightarrow 0 \mid \mathsf{S}z' \Rightarrow z'\,])\ (\mathsf{S}x)$	conversion
	$x = x$	$\mathsf{Q}x$

Proof term: $\lambda xyH.\ \mathsf{R}_{(\lambda z.\ x=\text{MATCH}\ z\ [0\Rightarrow 0\mid \mathsf{S}z'\Rightarrow z'])}\ H\ (\mathsf{Q}x)$

$X : \mathbb{T},\ x : X,\ y : X$	$x = y \to y = z \to x = z$	intros
$H : x = y$	$y = z \to x = z$	conversion
	$(\lambda a.\ a = z \to x = z)\ y$	apply $\mathsf{R}\,\_\,H$
	$(\lambda a.\ a = z \to x = z)\ x$	conversion
	$x = z \to x = z$	$\lambda h.h$

Proof term: $\lambda xyH.\ \mathsf{R}_{(\lambda a.\ a=z\to x=z)}\ H\ (\lambda h.h)$

Figure 5.2: Proofs of basic equational facts

predicate explicit, then the rewriting step as application of R, then a conversion step simplifying the claim, and then the final step proving the simplified claim.

We now understand how the basic proof steps "rewriting" and "proof by computational equality" used in the diagrams in Chapter 1 are realized in the propositions as types approach.

Exercise 5.2.1 Give proof diagrams and proof terms for the following propositions:

a) $\forall x^{\mathsf{N}}.\ 0 \neq \mathsf{S}x$

b) $\forall X^{\mathbb{T}}\, Y^{\mathbb{T}}\, f^{X \to y}\, x\, y.\ x = y \to fx = fy$

c) $\forall X^{\mathbb{T}}\, x^{X}\, y^{X}.\ x = y \to y = x$

d) $\forall X^{\mathbb{T}}\, Y^{\mathbb{T}}\, f^{X \to Y}\, g^{X \to Y}\, x.\ f = g \to fx = gx$

Exercise 5.2.2 Prove that the pair constructor is injective:
$\mathsf{pair}\, x\, y = \mathsf{pair}\, x'\, y' \to x = x' \wedge y = y'$.

Exercise 5.2.3 Prove the **converse rewriting law**
$\forall X^{\mathbb{T}} \forall x y \forall p^{X \to \mathbb{P}}.\ \mathsf{eq}\, X x y \to p y \to p x$.

Exercise 5.2.4 Verify the **impredicative characterization of equality**:

$$x = y \leftrightarrow \forall p^{X \to \mathbb{P}}.\ px \to py$$

Using Leibniz symmetry from Section 4.4, we may rewrite the equivalence to the equivalence

$$x = y \leftrightarrow \forall p^{X \to \mathbb{P}}.\ px \leftrightarrow py$$

known as **Leibniz characterization of equality**. Leibniz's characterization of equality may be phrased as saying that two objects are equal if and only if they satisfy the same properties.

The impredicative characterizations matter since they specify conjunction, disjunction, falsity, truth, and propositional equality prior to their definition. The impredicative characterizations may or may not be taken as definitions. Coq chooses inductive definitions since in each case the inductive definition provides additional benefits.

5.3 Declared Constants

To accommodate propositional equality, we assumed three constants eq, Q, and R. Assuming constants without justification is something one does not do in type theory. For instance, if we assume a constant of type $\perp$, we can prove everything (ex falso quodlibet) and our carefully constructed logical system collapses.

One solid justification we can have for a constant is that it was introduced as a constructor by an inductive definition. Inductive definitions can only be formed observing certain conditions ensuring that nothing bad can happen (i.e., a proof of falsity).

Another solid justification we can have for a group of constants is that we can define the constants with plain definitions. This way we know that any proof using the constants can also be done without the constants. Thus no proof of falsity can be introduced by the constants.

Here are plain definitions justifying the constants for propositional equality:

$$
\begin{aligned}
\mathsf{eq} &: \forall X^{\mathbb{T}}.\ X \to X \to \mathbb{P} \\
&:= \lambda Xxy.\ \forall p^{X \to \mathbb{P}}.\ px \to py \\
\mathsf{Q} &: \forall X^{\mathbb{T}} \forall x.\ \mathsf{eq}\, X\, x\, x \\
&:= \lambda Xxph.h \\
\mathsf{R} &: \forall X^{\mathbb{T}} \forall xy \forall p^{X \to \mathbb{P}}.\ \mathsf{eq}\, Xxy \to px \to py \\
&:= \lambda Xxypf.\, fp
\end{aligned}
$$

The definitions are amazingly simply. Check them by hand and with Coq. The idea for the definitions comes from the Leibniz characterization of equality we have seen in Exercise 5.2.4.

The above definition of equality is known as **Leibniz equality**. Coq uses another definition of equality based on an inductive definition following a scheme we will introduce later.

Note that for the equational reasoning done so far we completely ignored the definitions of the typed constants eq, Q, and R. This demonstrates an abstractness property of logical reasoning that appears as a general phenomenon.

It will often be useful to declare typed constants and hide their justifications. We speak of **declared constants**. In particular all lemmas and theorems[1] will be accommodated as declared constants. This makes explicit that when we use a lemma we don't need its proof but just its representation as a typed constant.

Conjunctions and disjunctions can also be accommodated with declared constants. Figure shows the constants needed for conjunctions and disjunctions. We distinguish between **constructors** and **eliminators**. The constructors are obtained directly with the inductive definitions we have seen for conjunction and disjunctions. The eliminators can be defined with matches for the respective inductive

[1] Whether we say theorem, lemma, corollary, or fact is a matter of style and doesn't make a formal difference. We shall use theorem as generic name (as in interactive theorem proving). As it comes to style, a lemma is a technical theorem needed for proving other theorems, a corollary is a consequence of a major theorem, and a fact is a straightforward theorem to be used tacitly in further proofs. If we call a result theorem, we want to emphasize its importance.

$$\begin{aligned}
\wedge &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\
\mathsf{C} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ X \to Y \to X \wedge Y \\
\mathsf{E}_\wedge &: \forall X^{\mathbb{P}} Y^{\mathbb{P}} Z^{\mathbb{P}}.\ X \wedge Y \to (X \to Y \to Z) \to Z \\[1ex]
\vee &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\
\mathsf{L} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ X \to X \vee Y \\
\mathsf{R} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ Y \to X \vee Y \\
\mathsf{E}_\vee &: \forall X^{\mathbb{P}} Y^{\mathbb{P}} Z^{\mathbb{P}}.\ X \vee Y \to (X \to Z) \to (Y \to Z) \to Z
\end{aligned}$$

Figure 5.3: Constructors and eliminators for conjunctions and disjunctions

predicates. As it comes to proofs, it suffices to have the eliminators as declared constants. As declared constants, the eliminators provide the constructions coming with matches but hide the accompanying reductions.

Note that the types of the eliminators $\mathsf{E}_\wedge$ and $\mathsf{E}_\vee$ are closely related to the impredicative characterizations of conjunction and disjunction (Section 4.2).

If we look at the constants for equality, we can identify eq and Q as constructors and R as eliminator.

Exercise 5.3.1 Define the eliminators for conjunction and disjunction based on the inductive definitions of conjunction and disjunction.

Exercise 5.3.2 Define the constructors and eliminators for conjunction and disjunction using their impredicative definitions. Do not use the inductive definitions.

Exercise 5.3.3 Prove commutativity of conjunction and disjunction just using the constructors and eliminators.

Exercise 5.3.4 Assume two sets $\wedge$, C, $\mathsf{E}_\wedge$ and $\wedge'$, C', $\mathsf{E}_{\wedge'}$ of constants for conjunctions. Prove $X \wedge Y \leftrightarrow X \wedge' Y$. Do the same for disjunction and propositional equality. We may say that the constructors and eliminators for a propositional construct characterize the propositional construct up to logical equivalence.

6 Inductive Elimination

Dependent function types and the conversion law make it possible to derive the proof rules for structural case analysis and structural induction we have used in Chapter 1. The surprisingly straightforward derivations are the final step in bootstrapping the abstractions used in Chapter 1 from a type-theoretic kernel language.

For inductive types we can define functions called eliminators that through their types provide the proof rules for case analysis and induction, and that through their defining equations provide expressive schemes for defining functions on the underlying inductive types. The eliminators have dependent function types and their definitions use matches with return type functions.

We will touch upon the elim restriction that constrains matches for propositional types by disallowing non-propositional return types.

6.1 Eliminators for Booleans

Recall the inductive definition of booleans:

$$\begin{aligned} \mathsf{B} &: \mathbb{T} \\ \mathsf{T} &: \mathsf{B} \\ \mathsf{F} &: \mathsf{B} \end{aligned}$$

In order to prove the equation

$$!!x = x$$

for boolean negation (see Section 1.1), we need structural case analysis on the boolean variable x. Following the design of the eliminator R for equality, we may accommodate boolean case analysis with a **boolean eliminator**

$$\mathsf{E_B} \; : \; \forall p^{\mathsf{B} \to \mathbb{P}}.\; p\,\mathsf{T} \to p\,\mathsf{F} \to \forall x.px$$

A justification for a declared constant $\mathsf{E_B}$ can be obtained with a boolean match:

$$\lambda pabx.\ \text{MATCH}\ x\ [\,\mathsf{T} \Rightarrow a \mid \mathsf{F} \Rightarrow b\,]$$

Figure 6.1 shows a proof diagram for $!!x = x$ using the eliminator $\mathsf{E_B}$.

$x : \mathsf{B}$	$!!x = x$	conversion
	$(\lambda x.\ !!x = x)x$	apply $\mathsf{E_B}$
1	$(\lambda x.\ !!x = x)\mathsf{T}$	$\mathsf{Q}\,\mathsf{T}$
2	$(\lambda x.\ !!x = x)\mathsf{F}$	$\mathsf{Q}\,\mathsf{F}$

Figure 6.1: Boolean case analysis with $\mathsf{E_B}$.

We define a **full boolean eliminator**

$$\begin{aligned} \hat{\mathsf{E}}_\mathsf{B} &: \forall p^{\mathsf{B}\to\mathbb{T}}.\ p\,\mathsf{T} \to p\,\mathsf{F} \to \forall x.px \\ &:= \lambda pabx.\ \text{MATCH}\ x\ [\,\mathsf{T} \Rightarrow a \mid \mathsf{F} \Rightarrow b\,] \end{aligned}$$

representing boolean matches in full generality. Using $\hat{\mathsf{E}}_\mathsf{B}$, we can define boolean negation as follows:

$$! := \hat{\mathsf{E}}_\mathsf{B}\,(\lambda x.\mathsf{B})\,\mathsf{F}\,\mathsf{T}$$

In fact, $\hat{\mathsf{E}}_\mathsf{B}\,(\lambda x.\mathsf{B})\,\mathsf{F}\,\mathsf{T}$ and $\lambda x.$ MATCH x $[\,\mathsf{T} \Rightarrow \mathsf{F} \mid \mathsf{F} \Rightarrow \mathsf{T}\,]$ are computationally equal.

We can also define the propositional eliminator with the full eliminator:

$$\mathsf{E_B} := \lambda p^{\mathsf{B}\to\mathbb{P}}.\ \hat{\mathsf{E}}_\mathsf{B}\,p$$

This works since Coq's type theory sees the function type $\mathsf{B} \to \mathbb{P}$ as a **subtype** of $\mathsf{B} \to \mathbb{T}$. The subtyping is justified since $\mathbb{P}$ is a subuniverse of $\mathbb{T}$. Using notation familiar from sets, we may write $\mathbb{P} \subseteq \mathbb{T}$ and $(\mathsf{B} \to \mathbb{P}) \subseteq (\mathsf{B} \to \mathbb{T})$ for the relevant subtypings.

We will use the eliminator $\mathsf{E_B}$ only as a declared constant. Following this policy, we will refer to $\mathsf{E_B}$ as the **elimination lemma for B**.

Exercise 6.1.1 For each of the following propositions give two normal proofs, one with a match and one with the eliminator $\mathsf{E_B}$.

a) $\forall x.\ x = \mathsf{T} \lor x = \mathsf{F}$.

b) $\forall p^{\mathsf{B}\to\mathbb{P}}.\ (\forall xy.\ y = x \to px) \to \forall x.px$.

c) $\forall p^{\mathsf{B}\to\mathbb{P}} \forall x^{\mathsf{B}}.\ (x = \mathsf{T} \to p\mathsf{T}) \to (x = \mathsf{F} \to p\mathsf{F}) \to px$.

Exercise 6.1.2 Define boolean conjunction with the full eliminator $\hat{\mathsf{E}}_\mathsf{B}$ such that the defining term is computationally equal to $\lambda xy.$ MATCH x $[\,\mathsf{T} \Rightarrow y \mid \mathsf{F} \Rightarrow \mathsf{F}\,]$.

Exercise 6.1.3 Prove $x\ \&\ y = \mathsf{T} \leftrightarrow x = \mathsf{T} \land y = \mathsf{T}$ and $x \mid y = \mathsf{F} \leftrightarrow x = \mathsf{F} \land y = \mathsf{F}$.

6.2 Matches with Return Type Functions

In the previous section we have tacitly used a typing rule for boolean matches that generalizes the typing rule for matches given in Section 3.8. With the old rule, a boolean match has type t if both rules yield type t. With the new rule, a boolean match on x has type px if the rule for T yields $p\mathsf{T}$ and the rule for F yields $p\mathsf{F}$:

$$\frac{x : \mathsf{B} \qquad p : \mathsf{B} \to \mathbb{T} \qquad s : p\mathsf{T} \qquad t : p\mathsf{F}}{\textsc{match}\ x\ [\,\mathsf{T} \Rightarrow s \mid \mathsf{F} \Rightarrow t\,] : px}$$

We refer to p as **return type function** for the match. We say that matches with return type functions are **dependently typed** to acknowledge the fact that the return type px depends on the value x being analyzed by the match.

Things become clearer once we look at the equational definition of the full boolean eliminator, giving a dependently typed functional account of the boolean match:

$$\begin{aligned}
&\hat{\mathsf{E}}_{\mathsf{B}} : \forall p^{\mathsf{B}\to\mathbb{T}}.\ p\,\mathsf{T} \to p\,\mathsf{F} \to \forall x.px \\
&\hat{\mathsf{E}}_{\mathsf{B}}\, p\, a\, b\, \mathsf{T} \ := \ a \\
&\hat{\mathsf{E}}_{\mathsf{B}}\, p\, a\, b\, \mathsf{F} \ := \ b
\end{aligned}$$

The equational definition explains the type checking for boolean matches using dependent function types. Given the variables $p : \mathsf{B} \to \mathbb{T}$, $a : p\,\mathsf{T}$, and $b : p\,\mathsf{F}$, we have the typings

$$\begin{aligned}
\hat{\mathsf{E}}_{\mathsf{B}}\, p\, a\, b\, \mathsf{T} \ &: \ p\,\mathsf{T} \\
\hat{\mathsf{E}}_{\mathsf{B}}\, p\, a\, b\, \mathsf{F} \ &: \ p\,\mathsf{F}
\end{aligned}$$

determining the right hand sides for the defining equations.

We will write dependently typed matches without stating the return type function explicitly, assuming that the return type function can be determined from the context. In Coq, the return type function of a match can be stated explicitly.

Simple matches without return type function can be understood as matches whose return type function is constant (i.e., $\lambda\_.t$).

6.3 Eliminators for Numbers

Recall the inductive definition of numbers:

$$\begin{aligned}
\mathsf{N} &: \ \mathbb{T} \\
0 &: \ \mathsf{N} \\
\mathsf{S} &: \ \mathsf{N} \to \mathsf{N}
\end{aligned}$$

In order to prove the equation (see Section 1.3)

$$x + 0 = x$$

we may use natural induction on the variable x. Following the design of the eliminator for booleans, we realize natural induction with a typed constant

$$\mathsf{E_N} : \forall p^{\mathsf{N} \to \mathbb{P}}.\ p0 \to (\forall x.\ px \to p(\mathsf{S}x)) \to \forall x.px$$

providing an inductive eliminator for numbers. Note how the eliminator obtains the subgoals for the base case and the successor case we have seen in the diagrams in Chapter 1, and how in the successor case the **inductive hypothesis** is obtained using an implication (make sure you can identify the inductive hypothesis). We refer to $\mathsf{E_N}$ as **induction lemma** for N.

It is important to understand how induction in proof digrams is justified by applications of the induction lemma

$$\mathsf{E_N} : \forall p^{\mathsf{N} \to \mathbb{P}}.\ p0 \to (\forall x.\ px \to p(\mathsf{S}x)) \to \forall x.px$$

and by introductions. Note that the variable x is quantified separately in the target and in the successor case of the eliminator. When we apply the induction lemma in proof diagrams or with Coq, it is usually convenient to reuse the name of x from the target in the successor case, but logically any other name can be used.

Figure 6.2 shows a detailed proof diagram for the equation $x + 0 = x$ using the induction lemma $\mathsf{E_N}$. The figure also shows the constructed proof term, which uses the variable h to identify the inductive hypothesis (called IH in the diagram), and the converse rewriting law

$$\mathsf{R}' : \forall X^{\mathbb{T}} \forall xy \forall p^{X \to \mathbb{P}}.\ \mathsf{eq}\,Xxy \to py \to px$$

to obtain left-to-right rewriting with the inductive hypothesis.

Although the proof term is small, there is a lot of detail and complexity in the proof shown in Figure 6.2. To understand what is going on, it is best to explore the details of the example in interaction with Coq.

Clearly, humans cannot routinely come up on paper with proof terms like the one shown in Figure 6.2. To get the details right, tedious type verification is needed, something where humans are not good at. However, the situation changes if one works with Coq, where the proof term is generated automatically from the proof script one constructs interactively with Coq following a trial and error strategy.

The reason for giving complex proof terms on paper in this chapter is that we are explaining the principles behind the reduction to the logical kernel language.

Exercise 6.3.1 Prove the following propositions with proof diagrams using the induction lemma $\mathsf{E_N}$. You will also need lemmas for the disjointness of 0 and S and the injectivity of S.

		$x + 0 = x$	conversion
		$(\lambda x.\, x + 0 = x)\, x$	apply $\mathsf{E_N}$
1		$(\lambda x.\, x + 0 = x)\, 0$	conversion
		$0 = 0$	comp. equality
2		$\forall x.\, (\lambda x.\, x + 0 = x)\, x \to (\lambda x.\, x + 0 = x)(\mathsf{S}x)$	conversion
		$\forall x.\, x + 0 = x \to \mathsf{S}x + 0 = \mathsf{S}x$	intros
	IH: $x + 0 = x$	$\mathsf{S}x + 0 = \mathsf{S}x$	conversion
		$\mathsf{S}(x + 0) = \mathsf{S}x$	rewrite IH
		$\mathsf{S}x = \mathsf{S}x$	comp. equality

proof term: $\mathsf{E_N}\, (\lambda x.x + 0 = x)\, (\mathsf{Q}\, 0)\, (\lambda x h.\, \mathsf{R}'\, (\lambda z.\mathsf{S}z = \mathsf{S}x)\, h\, (\mathsf{Q}(\mathsf{S}x)))\, x$

Figure 6.2: Detailed proof diagram for $x + 0 = x$

a) $\mathsf{S}n \neq n$.

b) $n + \mathsf{S}k \neq n$.

Do the proofs also with Coq's induction tactic to see how details can be omitted. Write high-level proof diagrams in the style of Chapter 1.

Exercise 6.3.2 Prove that addition is injective in its second argument: $x + y = x + z \to y = z$.

Justification of the eliminator

To justify the eliminator $\mathsf{E_N}$, we recursively define a **full eliminator** $\hat{\mathsf{E}}_\mathsf{N}$ for numbers (following the definition of the full eliminator for booleans):

$$\begin{aligned} \hat{\mathsf{E}}_\mathsf{N} :\ & \forall p^{\mathsf{N}\to\mathbb{T}}.\ p\,0 \to (\forall x.\, px \to p(\mathsf{S}x)) \to \forall x.px \\ \hat{\mathsf{E}}_\mathsf{N}\, p\, a\, \varphi\ 0\ &:=\ a \\ \hat{\mathsf{E}}_\mathsf{N}\, p\, a\, \varphi\ (\mathsf{S}x)\ &:=\ \varphi x(\hat{\mathsf{E}}_\mathsf{N}\, p\, a\, \varphi\, x) \end{aligned}$$

The eliminator obtains inductive proofs as recursive proofs. Given the functional interpretation of implications and universal quantifications, the recursive definition of the eliminator follows computational intuitions and is strongly guided by the typing discipline. Given the variables

$$\begin{aligned} p &:\ \mathsf{N} \to \mathbb{T} \\ a &:\ p\,0 \\ \varphi &:\ \forall x.\, px \to p(\mathsf{S}x) \\ x &:\ \mathsf{N} \end{aligned}$$

we have the typings

$$\begin{aligned} \hat{\mathsf{E}}_{\mathsf{N}}\, p\, a\, \varphi\ 0 &: p\,0 \\ \hat{\mathsf{E}}_{\mathsf{N}}\, p\, a\, \varphi\ (\mathsf{S}x) &: p(\mathsf{S}x) \end{aligned}$$

determining the right hand sides of the defining equations.

Exercise 6.3.3 Define $\hat{\mathsf{E}}_{\mathsf{N}}$ with FIX and MATCH.

6.4 Full Eliminator as Recursor

With the full eliminator $\hat{\mathsf{E}}_{\mathsf{N}}$ we can describe recursive functions on numbers. For instance, the terms

$$\hat{\mathsf{E}}_{\mathsf{N}}\,(\lambda\_.\mathsf{N})\,0\,(\lambda\_a.\mathsf{S}(\mathsf{S}a))$$

and

$$\text{FIX } fx.\ \text{MATCH } x\ [\,0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(fx'))\,]$$

are computationally equal and describe a function $\mathsf{N} \to \mathsf{N}$ doubling its argument. Moreover, the terms

$$\lambda x.\ \hat{\mathsf{E}}_{\mathsf{N}}\,(\lambda\_.\mathsf{N})\,x\,(\lambda\_.\mathsf{S})$$

and

$$\lambda x.\ \text{FIX } fy.\ \text{MATCH } y\ [\,0 \Rightarrow x \mid \mathsf{S}y' \Rightarrow \mathsf{S}(fy')\,]$$

are computationally equal and describe a function $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ adding two numbers. Note that the recursion is on the second argument of the function.

Since the full eliminator $\hat{\mathsf{E}}_{\mathsf{N}}$ represents a scheme for defining recursive functions, it is also called **recursor**.

Exercise 6.4.1 Describe a function $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ for truncating subtraction using $\hat{\mathsf{E}}_{\mathsf{N}}$ but not using match or fix. Tricky, use Coq to get it right.

6.5 A Quantified Inductive Hypothesis

Figure 6.3 shows a proof diagram for an inductive proof of the proposition

$$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \lor x \neq y$$

where it is essential that the inductive hypothesis quantifies over y. Without the quantification of y the destructuring of y at the beginning of subgoal 2 would affect the inductive hypothesis, making it impossible to close subgoals 2.2.1 and 2.2.2. Explore the details with Coq.

		$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \lor x \neq y$	apply $\mathsf{E_N}$, intros
1		$0 = y \lor 0 \neq y$	destruct y
1.1		$0 = 0 \lor 0 \neq 0$	trivial
1.2		$0 = \mathsf{S}y \lor 0 \neq \mathsf{S}y$	trivial
2	IH: $\forall y^{\mathsf{N}}.\ x = y \lor x \neq y$	$\mathsf{S}x = y \lor \mathsf{S}x = y$	destruct y
2.1		$\mathsf{S}x = 0 \lor \mathsf{S}x \neq 0$	trivial
2.2		$\mathsf{S}x = \mathsf{S}y \lor \mathsf{S}x \neq \mathsf{S}y$	destruct (IH y)
2.2.1	H: $x = y$	$\mathsf{S}x = \mathsf{S}y$	rewrite H, trivial
2.2.2	H: $x \neq y$	$\mathsf{S}x \neq \mathsf{S}y$	intros, apply H
	H_1: $\mathsf{S}x = \mathsf{S}y$	$x = y$	injectivity

Figure 6.3: Proof diagram with a quantified inductive hypothesis

6.6 Abstract Target Types

The type of the eliminator $\hat{\mathsf{E}}_{\mathsf{N}}$

$$\forall p^{\mathsf{N}\to\mathbb{T}}.\ p\,0 \to (\forall x.\ px \to p(\mathsf{S}x)) \to \forall x.px$$

is a function type that doesn't fully specify the number of arguments the function takes. For instance,

$$\hat{\mathsf{E}}_{\mathsf{N}}\,(\lambda\_.\mathsf{N})\ :\ \mathsf{N} \to (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N}$$

takes 3 arguments while

$$\hat{\mathsf{E}}_{\mathsf{N}}\,(\lambda\_.\mathsf{N} \to \mathsf{N})\ :\ (\mathsf{N} \to \mathsf{N}) \to (\mathsf{N} \to (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$

takes 4 arguments. We may say that $\hat{\mathsf{E}}_{\mathsf{N}}$ is polymorphic in the number of arguments it takes. This form of polymorphism becomes possible through the **abstract target type** px obtained with the **target type function** $p : \mathsf{N} \to \mathbb{T}$ taken as first argument.

The first eliminator with a dependent target type we saw was the rewrite law R for equality, where the target function was a predicate called rewriting predicate. The next use of an abstract target type was with the eliminators for booleans. A weaker form abstract target types appeared with the eliminators for conjunction and disjunction in Figure 5.3, where the entire target type Z is taken as an argument.

6.7 Eliminator for Pairs

Following the scheme we have seen for booleans and numbers, we can declare an eliminator for pairs (Section 1.5):

$$\begin{aligned}\mathsf{E}_\times :\ & \forall X^{\mathbb{T}} Y^{\mathbb{T}} \forall p^{X\times Y\to\mathbb{T}}.\ (\forall xy.\ p(x,y)) \to \forall a.pa\\ & \lambda XYpfa.\ \text{MATCH}\ a\ [\,(x,y) \Rightarrow fxy\,]\end{aligned}$$

Exercise 6.7.1 Prove the following facts for pairs $a : X \times Y$ using the eliminator $\mathsf{E}_\times$:

a) $(\pi_1 a, \pi_2 a) = a$.

b) $\mathsf{swap}(\mathsf{swap}\ a)$.

Exercise 6.7.2 Define a full eliminator for pairs and use it to write terms that are computationally equal to π_1, π_2, and swap (see Section 1.5).

6.8 Eliminator for Falsity

A simple but important inductive proposition is falsity:

$$\bot : \mathbb{P}$$

Since $\bot$ is defined without a proof constructor, $\bot$ has no proof. We may also say that $\bot$ is an empty type that is registered as a proposition. If $\bot$ is used in a context where its status as proposition does not matter, it is often called *void*. The full eliminator for $\bot$ is

$$\begin{aligned}\hat{\mathsf{E}}_\bot :\ & \forall Z^{\mathbb{T}}.\ \bot \to Z\\ :=\ & \lambda Zh.\ \text{MATCH}\ h\ [\,]\end{aligned}$$

The eliminator $\hat{\mathsf{E}}_\bot$ is a function that for every type Z and every proof of $\bot$ yields an inhabitant of Z. Logically, the constant $\hat{\mathsf{E}}_\bot$ provides the ex falso quodlibet rule, which says that from a proof of falsity we can get a proof of everything.

Look again at the definition of the full eliminator for falsity:

$$\begin{aligned}\hat{\mathsf{E}}_\bot :\ & \forall Z^{\mathbb{T}}.\ \bot \to Z\\ :=\ & \lambda Z^{\mathbb{T}} h^{\bot}.\ \text{MATCH}\ h^{\bot}\ [\,]^{Z}\end{aligned}$$

This time we have annotated the match with the **match type** ($\bot$) and the **return type** (Z) for clarity. The match type must be an inductive type and requires that the match comes with one rule for every value constructor of the match type. The return type requires that every rule of the match yields a value of the return type. Since in our case the match does not have rules, the return type requirement is satisfied vacuously.

6.9 Eliminator for Truth

Truth is an inductive proposition with exactly one proof:

$$\begin{aligned} \top &: \mathbb{P} \\ \mathsf{I} &: \top \end{aligned}$$

We may also say that $\top$ is a type with exactly one element that is registered as a proposition. If $\top$ is used in a context where its status as proposition does not matter, it is often called **unit**.

The full eliminator for $\top$ is

$$\begin{aligned} \hat{\mathsf{E}}_\top &: \forall p^{\top\to\mathbb{T}}.\ p\,\mathsf{I} \to \forall h^\top.\ ph \\ &:= \lambda pHh.\ \textsc{match}\ h^\top\ [\,\mathsf{I} \Rightarrow H\,]^{ph} \end{aligned}$$

Note that the return type of the match is ph, where p is the return type function and h is the variable being matched on. In the rule of the match h can be assumed to be I and thus it suffices that the rule yields a result of type $p\,\mathsf{I}$.

Things become clearer if we define $\hat{\mathsf{E}}_\top$ equationally:

$$\begin{aligned} &\hat{\mathsf{E}}_\top : \forall p^{\top\to\mathbb{T}}.\ p\,\mathsf{I} \to \forall h^\top.\ ph \\ &\hat{\mathsf{E}}_\top\, p\, H\, \mathsf{I} := H \end{aligned}$$

Note that the equational definition is exhaustive, disjoint, and non-recursive. The dependency of the return type on h is now handled by the typing rule for dependent function types. We may say that the equational specification of the eliminator $\hat{\mathsf{E}}_\top$ suggests the dependent return type rule for matches for $\top$.

Logically, the type of the eliminator for $\top$ says that a predicate holds for every proof of $\top$ if it holds for the canonical proof I. With the eliminator for $\top$ we can in fact show that all proofs of $\top$ are equal to I:

$$\begin{aligned} &\forall h{:}\top.\ h = \mathsf{I} \\ &\hat{\mathsf{E}}_\top\ (\lambda h.\ h = \mathsf{I})\ (\mathsf{Q}\,\mathsf{I}) \end{aligned}$$

Exercise 6.9.1 Prove $\forall p^{\top\to\mathbb{P}}\forall h.\ ph \leftrightarrow p\,\mathsf{I}$ using $\hat{\mathsf{E}}_\top$.

6.10 Elim Restriction

There is an important typing restriction on matches for inductive propositions and inductive predicates: If a match analyses a proof it must return a proof. We call the restriction **elim restriction**. There are two exceptions to the elim restriction:

1. Inductive propositions and inductive predicates with no proof constructors are exempted from the elim restriction.
2. Inductive propositions and inductive predicates with a single proof constructor are exempted from the elim restriction if every non-parametric argument of the proof constructor is a proof.

Note that $\bot$, $\top$ and $\wedge$ are exempted from the elim restriction:

- $\bot$ has no proof constructor.
- $\top$ has a single proof constructor not taking arguments.
- The predicate for conjunctions has a single proof constructor

$$\mathsf{C} : \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\, X \to Y \to X \wedge Y$$

  where X and Y are parametric arguments and the third and fourth argument are proofs.

If the target type of an elimination is not propositional, we speak of a **computational elimination**. The elim restriction is a device that disallows computational eliminations for most inductive propositions. We say that an inductive type is **computational** if it is not affected by the elim restriction.

The elim restriction is the price we have to pay for the impredicativity of the universe $\mathbb{P}$ of propositions. Without the elim restriction one could in fact construct a proof of falsity (a nontrivial result). On the positive side, the elim restriction makes it possible to assume the law of excluded middle $\forall X^{\mathbb{P}}.\, X \vee \neg X$ without enabling a proof of falsity.

If the elim restriction does not apply to a match on a proof, one speaks of a **singleton elimination**. Informally, we may say that singleton eliminations are possible since they don't leak equality of proofs to equality of non-proofs. It will turn out that singleton elimination plays an important role in Coq's type theory.

6.11 Final Remarks

This chapter is the first time matches with dependent return types (for $\top$, B, and N) are needed. We think that the typing rules for matches with return type functions are best explained through the equational definition of the concomitant eliminators.

Coq generates the inductive eliminators we have seen in this chapter automatically. That is, once Coq accepts an inductive definition, it will automatically define a collection of eliminators following a fixed scheme. The automatically defined eliminators are all accommodated as defined constants.

7 Existential Quantification

An existential quantification $\exists x : X.\ px$ says that there is a value of type X satisfying the predicate p. As normal proofs of $\exists x : X.\ px$ we take all pairs consisting of a term s of type X and a proof of the proposition ps. This design can be captured with an inductive predicate

$$\mathsf{ex} :\ \forall X^{\mathbb{T}}.\ (X \to \mathbb{P}) \to \mathbb{P}$$

and the notation

$$\exists x : t.s \ :=\ \mathsf{ex}\ t\ (\lambda x^t.\ s)$$

The normal proofs of existential quantifications are obtained with a single proof constructor

$$\mathsf{E} :\ \forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}} \forall x^X.\ px \to \mathsf{ex}\, X\, p$$

In this chapter we will prove two basic logical facts involving existential quantification known as Barber theorem (a non-existence theorem) and Lawvere's fixed point theorem (an existence theorem). From Lawvere's theorem we will obtain a type-theoretic variant of Cantor's theorem (no surjection from a set to its power set).

Given the type theoretic foundation built up so far, the representation of existential quantifications with an inductive predicate is straightforward. Essential ingredients are dependent function types, the conversion law, and lambda abstractions.

7.1 Inductive Definition and Basic Facts

Following the design laid out above, we introduce the constructors ex and E with the inductive definition

$$\begin{aligned} \mathsf{ex} &:\ \forall X^{\mathbb{T}}.\ (X \to \mathbb{P}) \to \mathbb{P} \\ \mathsf{E} &:\ \forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}} \forall x^X.\ px \to \mathsf{ex}\, X\, p \end{aligned}$$

We treat X as implicit argument of ex and X and p as implicit arguments of E, and use the familiar notation

$$\exists x.s \ :=\ \mathsf{ex}\,(\lambda x.s)$$

	$X:\mathbb{T},\ p:X\to\mathbb{P}$	$\neg(\exists x.px) \leftrightarrow \forall x.\neg px$	apply C
1		$\neg(\exists x.px) \to \forall x.\neg px$	intros
	$f:\neg(\exists x.px),\ x:X,\ a:px$	$\bot$	apply f
		$\exists x.px$	η-conversion
		$\mathsf{ex}\,p$	$\mathsf{E}\,xa$
2		$(\forall x.\neg px) \to \neg(\exists x.px)$	intros
	$f:\forall x.\neg px,\ x:X,\ a:px$	$\bot$	fxa

Proof term: $\mathsf{C}\ (\lambda fxa.f(\mathsf{E}_p xa))\ (\lambda fh.\,\textsc{match}\ h\ [\,\mathsf{E}\,xa \Rightarrow fxa\,])$

Figure 7.1: Proof of existential de Morgan law

where the abstraction $\lambda x.s$ ensures that x is a local variable visible only in the term s. Given a proof $\mathsf{E}\,su$, we call s the **witness** of the proof.

Figure 7.1 shows a proof diagram and the constructed proof term for a de Morgan law for existential quantification. Note the use of an η-conversion step $(\lambda x.px) \approx p$ in the direction from left to right. Also note that the proof constructor E is used for construction in the left-to-right direction and for elimination in the right-to-left direction (in the pattern of a match).

Exercise 7.1.1 Prove the following propositions with proof diagrams and give the resulting proof terms. Make all conversion steps explicit in the proof diagram.

a) $(\exists x \exists y.\ pxy) \to \exists y \exists x.\ pxy$.

b) $(\exists x.\ px \vee qx) \;\leftrightarrow\; (\exists x.px) \vee (\exists x.qx)$.

c) $(\exists x.px) \to \neg\forall x.\neg px$.

d) $((\exists x.px) \to Z) \;\leftrightarrow\; \forall x.\ px \to Z$.

e) $\neg\neg(\exists x.px) \;\leftrightarrow\; \neg\forall x.\neg px$.

f) $(\exists x.\neg\neg px) \;\to\; \neg\neg\exists x.px$.

Exercise 7.1.2 Prove $\forall X^{\mathbb{P}}.\ X \leftrightarrow \exists x:X.\top$.

Exercise 7.1.3 Verify the following existential characterization of disequality:

$$x \neq y \leftrightarrow \exists p.\ px \wedge \neg py$$

Exercise 7.1.4 Verify the impredicative characterization of existential quantification:

$$(\exists x.px) \leftrightarrow \forall Z^{\mathbb{P}}.\ (\forall x.\ px \to Z) \to Z$$

Exercise 7.1.5 Declare an eliminator $\mathsf{E}_\exists$ for existential quantification that can replace the use of existential matches in proofs. Note that the type of the eliminator is essentially the left-to-right direction of the impredicative characterization.

Exercise 7.1.6 Universal and existential quantification are compatible with propositional equivalence. Prove the following compatibility laws:

$$(\forall x.\ px \leftrightarrow qx) \to (\forall x.px) \leftrightarrow (\forall x.qx)$$
$$(\forall x.\ px \leftrightarrow qx) \to (\exists x.px) \leftrightarrow (\exists x.qx)$$

Exercise 7.1.7 Prove $\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}} \forall Z^{\mathbb{P}}.\ (\exists x.px) \land Z \leftrightarrow \exists x.\ px \land Z$.

7.2 Barber Theorem

Proofs of nonexistence are sometimes mystified and then attract a lot of attention. Here are two famous examples:

1. Russell: There is no set containing exactly those sets that do not contain themselves: $\neg\exists x\, \forall y.\ y \in x \leftrightarrow y \notin y$.
2. Turing: There is no Turing machine that halts exactly on the codes of those Turing machines that don't halt on their own code: $\neg\exists x\, \forall y.\ Hxy \leftrightarrow \neg Hyy$. Here H is a predicate that applies to codes of Turing machines such that Hxy says that Turing machine x halts on Turing machine y.

It turns out that both results are trivial consequences of a straightforward logical fact known as barber theorem.

Fact 7.2.1 (Barber Theorem) Let X be a type and p be a binary predicate on X. Then $\neg\exists x\, \forall y.\ pxy \leftrightarrow \neg pyy$.

Proof Suppose there is an x such that $\forall y.\ pxy \leftrightarrow \neg pyy$. Then $pxx \leftrightarrow \neg pxx$. Contradiction by Russell's law $\neg(X \leftrightarrow \neg X)$ shown in Section 3.6. ■

The barber theorem is related to a logical puzzle known as barber paradox. Search the web to find out more.

Exercise 7.2.2 Give a proof diagram and a proof term for the barber theorem. Construct a detailed proof with Coq.

7.3 Lawvere's Fixed Point Theorem

Another famous non-existence theorem is Cantor's theorem. Cantor's theorem says that there is no surjection from a set into its power set. If we analyse the situation in type theory, we find a proof that for no type X there is a surjective function $X \to (X \to \mathsf{B})$. If for X we take the type of numbers, the result says that the function type $\mathsf{N} \to \mathsf{B}$ is uncountable. It turns out that in type theory facts like these are best obtained as consequences of a general logical fact known as Lawvere's fixed point theorem.

A **fixed point** of a function $f : X \to X$ is an x such that $fx = x$.

Fact 7.3.1 Boolean negation has no fixed point.

Proof Consider $!x = x$ and derive a contradiction with boolean case analysis on x. ■

Fact 7.3.2 Propositional negation $\lambda P.\neg P$ has no fixed point.

Proof Suppose $\neg P = P$. Then $\neg P \leftrightarrow P$. Contradiction with Russell's law. ■

A function $f : X \to Y$ is **surjective** if $\forall y \exists x.\ fx = y$.

Theorem 7.3.3 (Lawvere) Suppose there exists a surjective function $X \to (X \to Y)$. Then every function $Y \to Y$ has a fixed point.

Proof Let $f : X \to (X \to Y)$ be surjective and $g : Y \to Y$. Then $fa = \lambda x.g(fxx)$ for some a. We have $faa = g(faa)$ by rewriting and conversion. ■

Corollary 7.3.4 There is no surjective function $X \to (X \to \mathsf{B})$.

Proof Boolean negation doesn't have a fixed point. ■

Corollary 7.3.5 There is no surjective function $X \to (X \to \mathbb{P})$.

Proof Propositional negation doesn't have a fixed point. ■

We remark that Corollaries 7.3.4 and 7.3.5 may be seen as variants of Cantor's theorem.

Exercise 7.3.6 Construct with Coq detailed proofs of the results in this section.

Exercise 7.3.7 For each of the following types

$$Y \;=\; \bot,\ \mathsf{B},\ \mathsf{B} \times \mathsf{B},\ \mathsf{N},\ \mathbb{P},\ \mathbb{T}$$

give a function $Y \to Y$ that has no fixed point.

Exercise 7.3.8 Show that every function $\top \to \top$ has a fixed point.

Exercise 7.3.9 With Lawvere's theorem we can give another proof of Fact 7.3.2 (propositional negation has no fixed point). In contrast to the proof given with Fact 7.3.2, the proof with Lawvere's theorem uses mostly equational reasoning.

The argument goes as follows. Suppose $(\neg X) = X$. Since the identity is a surjection $X \to X$, the assumption gives us a surjection $X \to (X \to \bot)$. Lawvere's theorem now gives us a fixed point of the identity on $\bot \to \bot$. Contradiction since the fixed point is a proof of falsity.

Do the proof with Coq.

8 Prominent Proofs

We have now arrived at a stage where we can prove interesting mathematical facts and where on paper we want to switch from proof terms and proof diagrams to mathematical proof outlines written for humans. This does not mean that we give up on machine-checked proofs, but rather that we delegate the generation of machine-checkable proofs to Coq. This way we can combine mathematical proof outlines designed for humans with the rigor of machine-checked proofs.

We consider four problems every reader should know:

- Proving that two types are not equal using cardinality arguments. Our lead example is $\mathsf{N} \neq \mathsf{B}$.
- Proving that equational specifications are unique up to functional extensionality. Our lead example is the specification of the Ackermann function.
- Doing case analysis with witnessing equations. Our lead example is Kaminski's equation $f(f(fx)) = fx$ for boolean functions $f^{\mathsf{B}\to\mathsf{B}}$.
- Equality deciders for data types.

The problems provide for the demonstration of basic proof techniques we have not discussed so far:

- Nested induction.
- Case analysis with witnessing equations.
- Quantified inductive hypotheses.
- Functional extensionality.

8.1 Disequality of Types

Informally, the types N and B of booleans and numbers are different since they have different cardinality: While there are infinitely many numbers, there are only two booleans. But can we show in the logical system we have arrived at that the types N and B are not equal?

Since B and N both have type $\mathbb{T}_1$, we can write the propositions $\mathsf{N} = \mathsf{B}$ and $\mathsf{N} \neq \mathsf{B}$. So the question is whether we can prove $\mathsf{N} \neq \mathsf{B}$. From Exercise 7.1.3 we know (using symmetry of equality) that it suffices to give a predicate p such that we can prove

$p\,\mathsf{B}$ and $\neg p\,\mathsf{N}$. We choose the predicate

$$\lambda X^{\mathbb{T}}.\ \forall x^X y^X z^X.\ \ x = y \lor x = z \lor y = z$$

saying that a type has at most two elements. With boolean case analysis on the variables x, y, z we can show that the property holds for B. Moreover, with $x = 0$, $y = 1$, and $z = 2$ we get the proposition

$$0 = 1 \lor 0 = 2 \lor 1 = 2$$

which can be disproved using the basic techniques for equality we have seen in Section 5.2.

Fact 8.1.1 $\mathsf{N} \neq \mathsf{B}$.

On paper, it doesn't make sense to work out the proof in more detail since this involves a lot of writing and routine verification. With Coq, however, doing the complete proof is quite rewarding since the writing and the tedious details are taken care of by the system. When we do the proof with Coq we can see that the techniques introduced so far smoothly scale to more involved proofs.

Exercise 8.1.2 Proof $\mathsf{B} \neq \top$ and $\mathsf{B} \neq \mathsf{B} \times \mathsf{B}$.

Exercise 8.1.3 Note that one cannot prove $\mathsf{B} \neq \mathsf{B} \times \top$ since one cannot give a predicate that distinguishes the two types. Neither can one prove $\mathsf{B} = \mathsf{B} \times \top$.

8.2 Unique Specification of Functions

We may ask whether there is a function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ satisfying the equations

$$f 0 y = y$$
$$f(\mathsf{S}x)y = \mathsf{S}(fxy)$$

for all numbers x and y. Since the equations qualify for an equational definition of a function (exhaustive, disjoint, structurally recursive in first argument), we know that there is a function satisfying the equations. In fact, the function realizes the addition operation for numbers and we used the equations in Section 1.2 to define the addition operation for numbers.

We may also ask whether the equations are satisfied by functions that are different from addition. The answer is no and, in fact, we can prove this. To do so, we define a predicate on functions

$$\begin{aligned} \mathsf{Add} :\ & (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathbb{P} \\ :=\ & \lambda f.(\forall y.\ f0y = y) \land (\forall xy.\ f(\mathsf{S}x)y = \mathsf{S}(fxy)) \end{aligned}$$

formalizing the equational specification and prove that two functions satisfying the specification agree on all arguments:

$$\forall fg.\ \mathsf{Add}\, f \to \mathsf{Add}\, g \to \forall xy.\ fxy = gxy \tag{8.1}$$

Proving this claim by induction on x is straightforward.

We may ask whether we can also prove the proposition

$$\forall fg.\ \mathsf{Add}\, f \to \mathsf{Add}\, g \to f = g \tag{8.2}$$

which strengthens (8.1) by asserting that the two functions are equal, which is stronger than asserting that they yield the same result for all arguments. It turns out that (8.2) can only be shown under a logical assumption known as functional extensionality. **Functional extensionality** says that two functions of the same type are equal if for every argument both functions yield the same result:

$$\forall X^{\mathbb{T}} Y^{\mathbb{T}} \forall f^{X\to Y} g^{X\to Y}.\ (\forall x.\ fx = gx) \to f = g$$

We summarize our findings about the specification of the addition function with a fact.

Fact 8.2.1 The equations

$$\begin{aligned} f0y &= y \\ f(\mathsf{S}x)y &= \mathsf{S}(fxy) \end{aligned}$$

specify the addition function for numbers up to functional extensionality.

Recall the specification of the Ackermann function discussed in Section 1.4.

Fact 8.2.2 The equations

$$\begin{aligned} f0y &= \mathsf{S}y \\ f(\mathsf{S}x)0 &= fx1 \\ f(\mathsf{S}x)(\mathsf{S}y) &= fx(f(\mathsf{S}x)y) \end{aligned}$$

have a solution $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ that is unique up to functional extensionality.

Proof That there is a function satisfying the equations was shown in Section 1.4. That the specification is unique up to functional extensionality

$$\forall fg.\ \mathsf{Ack}\, f \to \mathsf{Ack}\, g \to \forall xy.\ fxy = gxy$$

can be shown by induction on x and a nested induction on y in the successor case $\mathsf{S}x$. We ask the reader to study the details of the nested induction using the Coq script. Here we only look at the case where both the inductive hypothesis for x and the inductive hypothesis y are available:

$$\begin{array}{lll}
\mathsf{IHx} : \forall y.\ fxy = gxy & & \\
\mathsf{IHy} : f(\mathsf{S}x)y = g(\mathsf{S}x)y & f(\mathsf{S}x)(\mathsf{S}y) = g(\mathsf{S}x)(\mathsf{S}y) & \text{rewrite } \mathsf{Ack}\ f,\ \mathsf{Ack}\ g \\
 & fx(f(\mathsf{S}x)y) = gx(g(\mathsf{S}x)y) & \text{rewrite } \mathsf{IHy} \\
 & fx(g(\mathsf{S}x)y) = gx(g(\mathsf{S}x)y) & \mathsf{IHx}\ (g(\mathsf{S}x)y)
\end{array}$$

Note that the inductive hypothesis IHx quantifies y and that this quantification is needed for the proof to go through. To obtain the quantified inductive hypothesis, the claim must quantify y when the induction is started. ■

The proof uses two ideas we have not seen before: **Nested induction** and a **quantified inductive hypothesis**. Note that this is the first time we do not give the full details of a proof using new ideas here on paper. The reason is that stepping through the proof with Coq using the Coq script we provide is more instructive than reading a description of the detailled proof on paper.

Exercise 8.2.3 Prove that the equations

$$\begin{aligned} f0y &= y \\ f(\mathsf{S}x)y &= fx(\mathsf{S}y) \end{aligned}$$

specify the addition function for numbers up to functional extensionality.

Exercise 8.2.4 (Hardt's Identity) Prove that the equations

$$\begin{aligned} f0 &= 0 \\ f(\mathsf{S}x) &= \mathsf{S}(f(fx)) \end{aligned}$$

specify the identity function for numbers up to functional extensionality.

The pattern of the specification can be varied and then uniqueness may not be obvious. For instance, what happens if the second equation is changed to $f(\mathsf{S}x) = f(\mathsf{S}(fx))$?

Exercise 8.2.5 Define a maximum function $M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ and prove $Mxy = Myx$ and $M(x+y)x = x+y$.

Exercise 8.2.6 Consider the specification

$$\begin{aligned} p &: (\bot \to \mathsf{N}) \to \mathbb{P} \\ pf &:= \forall x.\ fx = 5 \end{aligned}$$

Show that there is a function satisfying the specification, and that two functions satisfying the specification are equal up to functional equality.

8.3 Kaminski's Equation

We will show that the equation[1]

$$f(f(fx)) = fx$$

holds for every boolean function $f : \mathsf{B} \to \mathsf{B}$ and every boolean $x : \mathsf{B}$. The equation says that applying a boolean function three times yields the same result as applying the function once. Here is a straightforward proof: We do boolean case analysis on x, $f\,\mathsf{T}$, and $f\,\mathsf{F}$. This gives us $8 = 2^3$ cases. In each case rewriting with the equations witnessing the boolean case analyses proves the equations. For instance, given the equations

$$x = \mathsf{T}, \quad f\,\mathsf{T} = \mathsf{F}, \quad f\,\mathsf{F} = \mathsf{T}$$

we have

$$f(f(fx)) = f(f(f\,\mathsf{T})) = f(f\,\mathsf{F}) = f\,\mathsf{T} = fx$$

When we do the proof with the boolean eliminator $\mathsf{E_B}$ from Section 6.1, we run into the difficulty that we don't get the **witnessing equations** for $f\,\mathsf{T}$ ($f\,\mathsf{T} = \mathsf{T}$ or $f\,\mathsf{T} = \mathsf{F}$) and $f\,\mathsf{F}$ ($f\,\mathsf{F} = \mathsf{T}$ or $f\,\mathsf{F} = \mathsf{F}$). This can be cured by using the lemma

$$\forall p^{\mathsf{B}\to\mathbb{P}} \forall x^{\mathsf{B}}.\ (x = \mathsf{T} \to p\,\mathsf{T}) \to (x = \mathsf{F} \to p\,\mathsf{F}) \to px$$

providing the witnessing equations in the subgoals. The lemma has a straightforward proof using boolean case analysis. We remark that Coq's case analysis tactic *destruct* can provide the witnessing equations (modifier *eqn*).

8.4 Equality Decider for Numbers

An **equality decider** for a type X is a boolean function $f : X \to X \to \mathsf{B}$ such that

$$\forall x^X y^X.\ x = y \ \leftrightarrow\ fxy = \mathsf{T}$$

Types that have an equality decider are called **discrete**.

It is straightforward to define an equality decider for the type of numbers:

$$\begin{aligned}
&\mathsf{eqb} : \ \mathsf{N} \to \mathsf{N} \to \mathsf{B} \\
&\qquad \mathsf{eqb}\,0\,0 \ := \ \mathsf{T} \\
&\qquad \mathsf{eqb}\,0\,(\mathsf{S}\_) \ := \ \mathsf{F} \\
&\qquad \mathsf{eqb}\,(\mathsf{S}\_)\,0 \ := \ \mathsf{F} \\
&\qquad \mathsf{eqb}\,(\mathsf{S}x)(\mathsf{S}y) \ := \ \mathsf{eqb}\,x\,y
\end{aligned}$$

[1] The equation was brought up as a proof challenge by Mark Kaminski in 2005 when he wrote his Bachelor's thesis on a calculus for classical higher-order logic.

Note that the decider recurses on the first argument and in both cases does a case analysis on the second argument. We have seen this structure before in Section 6.5 with the proof of $\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \vee x \neq y$. The correctness proof for eqb also follows this structure.

Fact 8.4.1 $\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \ \leftrightarrow\ \mathsf{eqb}\, x\, y = \mathsf{T}$.

Proof By induction on x and case analysis on y. It is essential that y is universally quantified in the inductive hypothesis. ■

Fact 8.4.2 Let X be a discrete type. Then $\forall x^X y^X.\ x = y \vee x \neq y$.

Proof Follows with boolean case analysis and an equality decider for X. ■

Ordinary mathematics considers the above fact trivial since ordinary mathematics assumes $P \vee \neg P$ for every proposition P. In our proof system, however, this logical assumption is not built in. We will discuss this issue in Chapter 9.

Datatypes like B, N, and $\mathsf{N} \times \mathsf{B}$ all have boolean equality deciders. In fact, discrete types are closed under products. On the other hand, we cannot construct boolean equality deciders for functional types like $\mathsf{N} \to \mathsf{B}$ or $\mathsf{N} \to \mathsf{N}$.

Exercise 8.4.3 Show that B is discrete.

Exercise 8.4.4 Show that a product type $X \times Y$ is discrete if X and Y are discrete.

Exercise 8.4.5 Show that the propositional types $\bot$ and $\top$ are discrete.

Exercise 8.4.6 Define a comparison function $\mathsf{leb} : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$ that tests $x \leq y$. Prove

$$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ \mathsf{leb}\, x\, y = \mathsf{T} \ \leftrightarrow\ \exists k.\ x + k = y$$

Note that we have not defined comparisons $x \leq y$ yet. A fine definition could be $x \leq y := \exists k.\ x + k = y$. The final definition will be an inductive definition appearing in Chapter 14.

Exercise 8.4.7 Assume functional extensionality. Prove the following:

a) $\top \to \top$ has exactly 1 element.

b) $\top \to \top$ is discrete.

c) $\mathsf{B} \to \mathsf{B}$ has exactly 4 elements.

d) $\mathsf{B} \to \mathsf{B}$ is discrete.

9 Axiomatic Freedom

A proposition X is independent in a given proof system if neither X nor $\neg X$ is provable. There are many interesting propositions that are independent in Coq's type theory. Two examples are functional extensionality and excluded middle.

Often it is interesting to explore the consequences of mathematical assumptions and obtain results that depend on certain assumptions. For this purpose, a proof system that has only basic logical assumptions built in is preferable over a proof system that has many mathematical assumptions built in since the basic proof system provides finer distinctions and grants more axiomatic freedom. There is in fact a recent foundational system (homotopy type theory) that conflicts with excluded middle but respects most of Coq's logical commitments.[1]

9.1 Metatheorems

Given a proposition, we may prove or disprove the proposition with Coq's proof system. Recall that a disproof of a proposition is a proof of its negation. We call a proposition *independent* if we can neither prove nor disprove it. That a proposition is independent cannot be shown with Coq's proof system since unprovability of a proposition cannot be stated as a proposition.

Given a proof system, results that cannot be shown within the system are called **metatheorems**, and properties that cannot be stated within the system are called **metaproperties**. Independence of a proposition is a metaproperty and a result saying that a certain proposition is independent in Coq's proof system is a metatheorem. Note that we can use Coq to show for many propositions that they are not independent by proving or disproving the proposition with Coq.

An important metaproperty for any proof system is **consistency** saying that there is no proof of falsity. There is a general result (Gödel's incompleteness theorem) that says that no sufficiently strong proof system can prove its own consistency.

[1] There is a conflict with Coq's impredicative universe of propositions.

9.2 Abstract Provability Predicates

There is a way to prove certain properties of provability within Coq. The trick is to assume an abstract provability predicate

$$\mathsf{provable} : \mathbb{P} \to \mathbb{P}$$

satisfying certain properties. For our purposes the following properties suffice:

$$\begin{aligned}
\mathsf{PA}: &\ \forall XY.\ \mathsf{provable}\,(X \to Y) \to \mathsf{provable}\ X \to \mathsf{provable}\ Y\\
\mathsf{PI}: &\ \forall X.\ \mathsf{provable}\,(X \to X)\\
\mathsf{PK}: &\ \forall XY.\ \mathsf{provable}\ Y \to \mathsf{provable}\,(X \to Y)\\
\mathsf{PN}: &\ \forall XY.\ \mathsf{provable}\,(X \to Y) \to \mathsf{provable}\,(\neg Y \to \neg X)
\end{aligned}$$

Since Coq's provability predicate satisfies these properties, we can expect that properties we can show for abstract provability predicates also hold for Coq's provability predicate.

We identify three prominent properties based on provability:

- A proposition X is **contradictory** if $\neg X$ is provable.[2]

$$\mathsf{contradictory}\ X \ := \ \mathsf{provable}\,(\neg X)$$

- A proposition is **consistent** if it is not contradictory.

$$\mathsf{consistent}\ X \ := \ \neg\mathsf{contradictory}\,X$$

- A proposition is **independent** if it is unprovable and consistent.

$$\mathsf{independent}\ X \ := \ \neg\mathsf{provable}\,X \land \mathsf{consistent}\,X$$

The assumption PA known as *modus ponens* says that provability transports through implication. Unprovability thus transports in the reverse direction. With PN we then obtain that consistency transports through implications the same way provability does.

Fact 9.2.1 (Transport)

1. $\mathsf{provable}(X \to Y) \ \to \ \neg\,\mathsf{provable}\,Y \ \to \ \neg\,\mathsf{provable}\,X.$
2. $\mathsf{provable}(X \to Y) \ \to \ \mathsf{consistent}\,X \ \to \ \mathsf{consistent}\,Y.$

Proof Claim 1 follows with PA. Claim 2 follows with PN and (1). ■

[2] Note that a proposition is contradictory iff we can disprove it.

Corollary 9.2.2 Provability, unprovability, consistence, and independence all **transport** through propositional equivalence. Formally, if $X \to Y$ and $Y \to X$ are provable, then:

1. If X is provable, then Y is provable.
2. If X is unprovable, then Y is unprovable.
3. If X is consistent, then Y is consistent.
4. If X is independent, then Y is independent.

From the transport properties it follows that a proposition is independent if it can be sandwiched between a consistent and an unprovable proposition.

Theorem 9.2.3 (Sandwich) A proposition Y is independent if there exist propositions X and Z such that:

1. $X \to Y$ and $Y \to Z$ are provable.
2. X is consistent and Z is unprovable.

Proof Follows with Fact 9.2.1. ∎

A key property of provability is **consistency** saying that there is no proof of falsity. It turns out that consistency has interesting equivalent characterizations that can be established for abstract proof predicates.

Fact 9.2.4 (Consistency) The following propositions are equivalent:

1. $\neg\,\mathsf{provable}\,\bot$.
2. $\mathsf{consistent}\,(\neg\bot)$.
3. There is a consistent proposition.
4. Every provable proposition is consistent.

Proof $1 \to 2$. We assume $\mathsf{provable}(\neg\neg\bot)$ and show $\mathsf{provable}\,\bot$. By PA it suffices to show $\mathsf{provable}(\neg\bot)$, which holds by PI.

$2 \to 3$. Trivial.

$3 \to 1$. Suppose X is consistent. We assume $\mathsf{provable}\,\bot$ and show $\mathsf{provable}\,(\neg X)$. Follows by PK.

$1 \to 4$. We assume that $\bot$ is unprovable, X is provable, and $\neg X$ is provable. By PA we have $\mathsf{provable}\,\bot$. Contradiction.

$4 \to 1$. We assume that $\bot$ is provable and derive a contradiction. By the primary assumption it follows that $\neg\bot$ is unprovable. Contradiction since $\neg\bot$ is provable by PI. ∎

From Fact 9.2.4 we learn that a provability predicate is consistent if there are consistent propositions.

Exercise 9.2.5 Show that the functions $\lambda X^{\mathbb{P}}.X$ and $\lambda X^{\mathbb{P}}.\top$ are abstract provability predicates satisfying PA, PI, PK, and PN.

Exercise 9.2.6 Let $X \to Y$ be provable. Show that X and Y are both independent if X is consistent and Y is unprovable.

Exercise 9.2.7 Assume a provability predicate satisfying PA, PI, PK, PN, and also

$$\mathsf{PE}: \ \forall X^{\mathbb{P}}.\ \mathsf{provable}\ \bot \to \mathsf{provable}\ X$$

Prove $\neg\mathsf{provable}\ \bot \leftrightarrow \neg\forall X^{\mathbb{P}}.\ \mathsf{provable}\ X$.

Exercise 9.2.8 We may consider more abstract provability predicates

$$\mathsf{provable}: \ \mathsf{prop} \to \mathbb{P}$$

where prop is an assumed type of propositions with two assumed constants

$$\begin{aligned} \mathsf{falsity}&: \ \mathsf{prop} \\ \mathsf{impl}&: \ \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop} \end{aligned}$$

Show all results of this section for such abstract proof systems.

9.3 Prominent Independent Propositions

Figure 9.1 lists prominent propositions that are independent in Coq. Here are informal readings of the propositions.

- **Truth value semantics** (TVS) says that every proposition equals either $\top$ or $\bot$.
- **Excluded middle** (XM) says that every proposition is either provable or disprovable.
- **Limited propositional omniscience** (LPO) says that tests on numbers are either satisfiable or unsatisfiable.
- **Markov's principle** (Markov) says that a test on numbers that is not constantly false is true for some number. Markov's principle may be seen as a specialized de Morgan law.
- **Propositional extensionality** (PE) says that equivalent propositions are equal.
- **Proof irrelevance** (PI) says that propositions have at most one proof.
- **Functional extensionality** (FE) says that functions are equal if they agree on all arguments.

$$\begin{array}{rcl}
\mathsf{TVS} & := & \forall X^{\mathbb{P}}.\ X = \top \lor X = \bot \\
\mathsf{XM} & := & \forall X^{\mathbb{P}}.\ X \lor \neg X \\
\mathsf{LPO} & := & \forall f^{\mathsf{N}\to\mathsf{B}}.\ (\exists n.\, fn = \mathsf{T}) \lor \neg(\exists n.\, fn = \mathsf{T}) \\
\mathsf{Markov} & := & \forall f^{\mathsf{N}\to\mathsf{B}}.\ \neg(\forall n.\, fn = \mathsf{F}) \to (\exists n.\, fn = \mathsf{T}) \\
\mathsf{PE} & := & \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ X \leftrightarrow Y \to X = Y \\
\mathsf{PI} & := & \forall X^{\mathbb{P}} \forall a^X b^X.\ a = b \\
\mathsf{FE} & := & \forall X^{\mathbb{T}} Y^{\mathbb{T}} \forall f^{X\to Y} g^{X\to Y}.\ (\forall x.\, fx = gx) \to f = g
\end{array}$$

Figure 9.1: Independent propositions

TVS → XM	TVS → PE	PE → PI
XM → LPO	XM → PE → TVS	
LPO → Markov		

Figure 9.2: Provable implications

Note that LPO and Markov talk about tests on numbers and quantify only over types in $\mathbb{T}_1$. The other propositions in Figure 9.1 do not involve data types but quantify over universes.

Using the sandwich theorem, we will be able to show that all propositions in Figure 9.1 are independent provided we are given two nontrivial metatheorems.

Theorem 9.3.1 (Meta) TVS $\land$ FE is consistent.

Theorem 9.3.2 (Meta) Neither Markov nor PI nor FE is provable.

From Theorem 9.3.1 we obtain consistency as a corollary.

Corollary 9.3.3 (Meta) $\bot$ is unprovable.

Proof Follows with Fact 9.2.4 from Theorem 9.3.1. ∎

We will now prove the implications shown in Figure 9.2. The implications suffice so that with the sandwich theorem 9.2.3 and the metatheorems 9.3.1 and 9.3.2 we know that all propositions of Figure 9.1 are independent.

Fact 9.3.4 XM $\leftrightarrow \forall X^{\mathbb{P}}.\, (X \leftrightarrow \top) \lor (X \leftrightarrow \bot)$.

Proof Straightforward. ∎

Fact 9.3.5 TVS ↔ XM $\wedge$ PE.

Proof Straightforward using Fact 9.3.4. ■

Fact 9.3.6 XM → LPO and LPO → Markov.

Proof The first claim is obvious. For the second claim assume LPO and f such that $H : \neg\forall n.\ fn = \mathsf{F}$. By LPO we assume $H_1 : \neg\exists n.\ fn = \mathsf{T}$ and prove falsity. By H we prove $fn = \mathsf{F}$ for some n. By boolean case analysis we assume $fn = \mathsf{T}$ and prove falsity. The proof closes with H_1. ■

We call a proposition **pure** if it has at most one proof:

$$\begin{aligned}&\mathsf{pure} : \mathbb{P} \to \mathbb{P}\\ &\mathsf{pure}\ X \ := \ \forall a^X b^X.\ a = b\end{aligned}$$

Note that PI says that all propositions are pure.

Fact 9.3.7 $\bot$ and $\top$ are pure.

Proof Follows with the eliminators for $\bot$ and $\top$. ■

Fact 9.3.8 PE → PI.

Proof Assume PE and let a and b be two proofs of a proposition X. We show $a = b$. Since $X \leftrightarrow \top$, we have $X = \top$ by PE. Hence X is pure since $\top$ is pure. The claim follows. ■

Theorem 9.3.9 (Meta) All propositions in Figure 9.1 are independent.

Proof By the sandwich theorem 9.2.3 and Facts 9.3.5, 9.3.8, and 9.3.6 it suffices to show that TVS and FE are consistent and that PI, Markov, and FE are unprovable. This is exactly what Theorems 9.3.1 and 9.3.2 say. ■

Exercise 9.3.10 Prove TVS $\leftrightarrow\ \forall XYZ{:}\mathbb{P}.\ X = Y \vee X = Z \vee Y = Z$. Note that the equivalence characterizes TVS without using $\top$ and $\bot$.

Exercise 9.3.11 Prove TVS $\leftrightarrow\ \forall p^{\mathbb{P}\to\mathbb{P}}.\ p\top \to p\bot \to \forall X.pX$. Note that the equivalence characterizes TVS without using propositional equality.

Exercise 9.3.12 Prove that $\forall X^{\mathbb{T}}.\ X = \top \vee X = \bot$ is contradictory.

Exercise 9.3.13 We define **implicational excluded middle** as

$$\mathsf{IXM} \ := \ \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ (X \to Y) \vee (Y \to X)$$

a) Prove $\mathsf{XM} \to \mathsf{IXM}$.

b) Prove that IXM is consistent.

We remark that neither IXM nor $\mathsf{IXM} \to \mathsf{XM}$ is provable in Coq's type theory.

Exercise 9.3.14 We define **weak excluded middle** as

$$\mathsf{WXM} := \forall X^{\mathbb{P}}.\ \neg X \vee \neg\neg X$$

a) Prove $\mathsf{WXM} \leftrightarrow \forall X^{\mathbb{P}}.\ \neg\neg X \vee \neg\neg\neg X$.

b) Prove $\mathsf{WXM} \leftrightarrow \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ \neg(X \wedge Y) \to \neg X \vee \neg Y$.

c) Prove that WXM is consistent.

Note that (b) says that WXM is equivalent to the de Morgan law for conjunction. We remark that neither WXM nor $\mathsf{WXM} \to \mathsf{XM}$ is provable in Coq's type theory.

Exercise 9.3.15 Prove $\mathsf{FE} \to \mathsf{pure}\,(\top \to \top)$.

Exercise 9.3.16 Prove $\mathsf{FE} \to \mathsf{B} \neq (\top \to \top)$.

Exercise 9.3.17 Suppose there is a function $f : (\exists x^{\mathsf{B}}.\top) \to \mathsf{B}$ such that $f(\mathsf{E}\,x\,\mathsf{I}) = x$ for all x. Prove $\neg\,\mathsf{PI}$. Convince yourself that without the elim restriction you could define a function f as assumed.

Exercise 9.3.18 Suppose there is a function $f : (\top \vee \top) \to \mathsf{B}$ such that $f(\mathsf{L}\,\mathsf{I}) = \mathsf{T}$ and $f(\mathsf{R}\,\mathsf{I}) = \mathsf{F}$. Prove $\neg\,\mathsf{PI}$. Convince yourself that without the elim restriction you could define a function f as assumed.

Exercise 9.3.19 Functional extensionality can be formulated more generally for dependently typed functions:

$$\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{T}} \forall f^{\forall x.px} \forall g^{\forall x.px}.\ (\forall x.\ fx = gx) \to f = g$$

Convince yourself that the dependently typed version implies the simply typed version FE. We remark that the dependently typed version is consistent in conjunction with TVS.

9.4 Sets

Given FE and PE, predicates over a type X correspond exactly to sets whose elements are taken from X. We may define membership as $x \in p := px$. In particular, we obtain that two sets (represented as predicates) are equal if they have the same elements (set extensionality). Moreover, we can define the usual set operations:

$$\begin{array}{rcll} \emptyset & := & \lambda x^X.\bot & \text{empty set} \\ p \cap q & := & \lambda x^X.px \wedge qx & \text{intersection} \\ p \cup q & := & \lambda x^X.px \vee qx & \text{union} \\ p - q & := & \lambda x^X.px \wedge \neg qx & \text{difference} \end{array}$$

Exercise 9.4.1 Prove $x \in (p - q) \leftrightarrow x \in p \wedge x \notin q$. Check that the equation $(x \in (p - q)) = (x \in p \wedge x \notin q)$ holds by computational equality.

Exercise 9.4.2 Assume FE and PE and prove the following:

1. $(\forall x.\, x \in p \leftrightarrow x \in q) \to p = q$.
2. $p - (q \cup r) = (p - q) \cap (p - r)$.

9.5 No Computational Omniscience

Coq's type theory is carefully designed such that every definable function is computable. In fact, computability of definable functions is preserved if we assume TVS and FE.[3] On the other hand, using existential quantification, we can ask for the existence of functions having properties no computable function can have. Here is a proposition we call **computational omniscience**:

$$\mathsf{CO} := \exists F^{(\mathsf{N}\to\mathsf{B})\to\mathsf{B}}\,\forall f^{\mathsf{N}\to\mathsf{B}}.\ Ff = \mathsf{T} \leftrightarrow \exists n.\, fn = \mathsf{T}$$

CO states the existence of a boolean function F deciding whether tests on numbers are satisfiable. Computationally, we can apply a test f only finitely often, but after finitely many negative outcomes it is still possible that f tests positively the next number we try. In short, a computable satisfiability decider for tests on numbers cannot exist because there are infinitely many numbers.

Fact 9.5.1 $\mathsf{CO} \to \mathsf{LPO}$.

Proof Straightforward. ∎

As pointed out in the discussion above, we expect that ¬CO is consistent in the presence of TVS and FE.

Conjecture 9.5.2 (Meta) TVS ∧ FE ∧ ¬CO is consistent.

It turns out that assuming CO is also consistent.

[3] The results in the literature and experience support this claim, but there is no full proof yet.

Theorem 9.5.3 (Meta) TVS ∧ FE ∧ CO is consistent.

From the conjecture and the theorem it follows that CO is independent, even in the presence of TVS and FE. So technically, we could go either way. In this text, we want a type theory where all definable functions are computable, and hence will not admit assumptions conflicting with ¬CO.

9.6 Discussion

Basic Intuitionistic Reasoning

In mathematical practice TVS and FE are tacitly assumed. What is surprising at first is that *basic intuitionistic reasoning* (the reasoning directly obtained with the propositions as types principle) can prove so many interesting theorems. Basic intuitionistic reasoning is valuable in that it provides a basis for studying tacit assumptions used in mathematical reasoning.

Markov versus LPO

Markov's principle is weaker than LPO but still not provable with basic intuitionistic reasoning. If we look at Markov's principle

$$\forall f^{\mathsf{N}\to\mathsf{B}}.\ \neg(\forall n.\, fn = \mathsf{F}) \to \exists n.\, fn = \mathsf{T}$$

we see that a proof of the principle is a function that given a proof that a boolean test for numbers is not constantly negative returns a number where the test is positive. Such a function can be realized (not in Coq so) with an algorithm that starting from $n = 0$ checks the test until it finds a number testing positively. In contrast, such an algorithmic realization does not exist for a function proving LPO.

There is a philosophical direction called intuitionism that will only accept intuitionistic reasoning, which is basic intuitionistic reasoning plus assumptions whose proof functions can be realized algorithmically. While the use of Markov's principle is fine for intuitionists, the use of LPO is not. The results in the literature suggest that LPO does not imply XM.

Consistency of Proof Irrelevance

Given the setup of Coq's proof system, where the structure of canonical proofs is crucial for reasoning with proofs of inductive propositions, the consistency of proof irrelevance is surprising, in particular, as it comes to disjunctions. There is the important result that proof irrelevance is already implied by excluded middle (we will see a proof in a later chapter).

10 Excluded Middle and Double Negation

We consider propositionally equivalent characterizations of excluded middle, including Peirce's law, the double negation law, and the counterexample law. We show for several examples that the double negation of a quantification-free proposition can be shown even if the proposition itself can only be shown with excluded middle. We also consider definiteness and stability of propositions, two interesting properties that trivially hold under excluded middle.

10.1 Characterizations of Excluded Middle

Recall that excluded middle

$$\mathsf{XM} \;:=\; \forall X^{\mathbb{P}}.\; X \lor \neg X$$

is independent in Coq's type theory. There are several propositionally equivalent characterizations of excluded middle. Most amazing is Peirce's law that formulates excluded middle with just implication.

Fact 10.1.1 The following propositions are equivalent. That is, if we can prove one of them, we can prove all of them.

1. $\forall X^{\mathbb{P}}.\; X \lor \neg X$ *excluded middle*
2. $\forall X^{\mathbb{P}}.\; \neg\neg X \to X$ *double negation*
3. $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\; (\neg X \to \neg Y) \to Y \to X$ *contraposition*
4. $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\; ((X \to Y) \to X) \to X$ *Peirce's law*
5. $\forall X^{\mathbb{P}}.\, (X \leftrightarrow \top) \lor (X \leftrightarrow \bot)$

Proof Since (5) is a minor reformulation of (1), proving the implications $1 \to 5$ and $5 \to 1$ is easy. It remains to prove the implications $1 \to 2 \to 3 \to 4 \to 1$.

$1 \to 2$. Assume $\neg\neg X$ and show X. By (1) we have either X or $\neg X$. Both cases are easy.

$2 \to 3$. Assume $\neg X \to \neg Y$ and Y and show X. By (2) it suffices to show $\neg\neg X$. We assume $\neg X$ and show X. Follows by ex falso quodlibet since we have Y and $\neg Y$.

$3 \to 4$. By (3) it suffices to show $\neg X \to \neg((X \to Y) \to X))$. Straightforward.

$4 \to 1$. By (4) with $X \mapsto (X \lor \neg X)$ and $Y \mapsto \bot$ we can assume $\neg(X \lor \neg X)$ and prove $X \lor \neg X$. We assume X and prove $\bot$. Straightforward since we have $\neg(X \lor \neg X)$. ■

$\neg(X \wedge Y)$	$\leftrightarrow$	$\neg X \vee \neg Y$	de Morgan
$\neg(\forall a.pa)$	$\leftrightarrow$	$\exists a.\neg pa$	de Morgan
$(\neg X \to \neg Y)$	$\leftrightarrow$	$(Y \to X)$	contraposition
$(X \to Y)$	$\leftrightarrow$	$\neg X \vee Y$	classical implication

Figure 10.1: Prominent equivalences only provable with XM

There is another characterization of excluded middle asserting existence of counterexamples, often used as tacit assumption in mathematical arguments.

Fact 10.1.2 (Counterexample) XM $\leftrightarrow$ $\forall A^{\mathbb{T}} \forall p^{A \to \mathbb{P}}.\ (\forall a.pa) \vee \exists a.\neg pa.$

Proof Assume XM and $p^{A \to \mathbb{P}}$. By XM we assume $\neg\exists a.\neg pa$ and prove $\forall a.pa$. By the de Morgan law for existential quantification we have $\forall a.\neg\neg pa$. The claim follows since XM implies the double negation law.

Now assume the right hand side and let X be a proposition. We prove $X \vee \neg X$. We choose $p := \lambda a^{\top}.X$. By the right hand side and conversion we have either $\forall a^{\top}.X$ or $\exists a^{\top}.\neg X$. In each case the claim follows. Note that choosing an inhabited type for A is essential. ■

Another common tacit use of XM in Mathematics is **proof by contradiction**: To prove s, we assume $\neg s$ and derive a contradiction. The lemma justifying proof by contradiction is double negation:

$$\mathsf{XM} \to (\neg X \to \bot) \to X$$

Figure 10.1 shows prominent equivalences whose left-to-right directions are only provable with XM. Note the de Morgan laws for conjunction and universal quantification. Recall that the de Morgan laws for disjunction and existential quantification

$\neg(X \vee Y)$	$\leftrightarrow$	$\neg X \wedge \neg Y$	de Morgan
$\neg(\exists a.pa)$	$\leftrightarrow$	$\forall a.\neg pa$	de Morgan

have constructive proofs.

Exercise 10.1.3

a) Prove the right-to-left directions of the equivalences in Figure 10.1.

b) Prove the left-to-right directions of the equivalences in Figure 10.1 using XM.

Exercise 10.1.4 Prove the following equivalences possibly using XM. In each case find out which direction needs XM.

$$\begin{aligned}\neg(\exists a.\neg pa) &\leftrightarrow \forall a.pa\\ \neg(\exists a.\neg pa) &\leftrightarrow \neg\neg\forall a.pa\\ \neg\neg(\exists a.pa) &\leftrightarrow \neg\forall a.\neg pa\end{aligned}$$

Exercise 10.1.5 Prove that the left-to-right direction of the de Morgan law for universal quantification implies XM:

$$(\forall X^{\mathbb{T}} \forall p^{X\to\mathbb{P}}.\ \neg(\forall x.px) \to (\exists x.\,\neg px)) \to \mathsf{XM}$$

Hint: Instantiate the de Morgan law with $X \vee \neg X$ and $\lambda\_.\bot$.

Exercise 10.1.6 Make sure you can prove the de Morgan laws for disjunction and existential quantification (not using XM).

Exercise 10.1.7 Explain why Peirce's law and the double negation law are independent in Coq's type theory.

Exercise 10.1.8 (Drinker Paradox) Consider a bar populated by at least one person. Using excluded middle, one can argue that one can pick some person in the bar such that everyone in the bar drinks Whiskey if this person drinks Whiskey.

We assume an inhabited type X representing the persons in the bar and a predicate $p^{X\to\mathbb{P}}$ identifying the persons who drink Whiskey. The job is now to prove the proposition $\exists x.\ px \to \forall x.px$. Do the proof in detail and point out where XM and inhabitation of X are needed. A nice proof can be done with the counterexample law Fact 10.1.2.

10.2 Double Negation

Given a proposition X, we call $\neg\neg X$ the **double negation** of X. It turns out that the double negation of a quantifier-free proposition is provable even if the proposition by itself is only provable with XM. For instance,

$$\forall X^{\mathbb{P}}.\ \neg\neg(X \vee \neg X)$$

is provable. This metaproperty cannot be proved in Coq. However, for every instance a proof can be given in Coq.

There is a useful proof technique for working with double negation: If we have a double negated assumption and need to derive a proof of falsity, we can **drop the double negation**. The lemma behind this is simply identity:

$$\neg\neg X \to (X \to \bot) \to \bot$$

With excluded middle, double negation distributes over all connectives and quantifiers. Without excluded middle, we can still prove that double negation distributes over implication and conjunction.

Fact 10.2.1 The following **distribution laws** for double negation are provable:

$$\begin{aligned}\neg\neg(X \to Y) &\leftrightarrow (\neg\neg X \to \neg\neg Y)\\ \neg\neg(X \land Y) &\leftrightarrow \neg\neg X \land \neg\neg Y\\ \neg\neg\top &\leftrightarrow \top\\ \neg\neg\bot &\leftrightarrow \bot\end{aligned}$$

Exercise 10.2.2 Prove the equivalences of Fact 10.2.1.

Exercise 10.2.3 Prove the following propositions:

$$\begin{aligned}\neg(X \land Y) &\leftrightarrow \neg\neg(\neg X \lor \neg Y)\\ (\neg X \to \neg Y) &\leftrightarrow \neg\neg(Y \to X)\\ (\neg X \to \neg Y) &\leftrightarrow (Y \to \neg\neg X)\\ (X \to Y) &\to \neg\neg(\neg X \lor Y)\end{aligned}$$

Exercise 10.2.4 Prove $\neg(\forall a.\neg pa) \leftrightarrow \neg\neg\exists a.pa$.

Exercise 10.2.5 Prove the following implications:

$$\begin{aligned}\neg\neg X \lor \neg\neg Y &\to \neg\neg(X \lor Y)\\ (\exists a.\neg\neg pa) &\to \neg\neg\exists a.pa\\ \neg\neg(\forall a.pa) &\to \forall a.\neg\neg pa\end{aligned}$$

Convince yourself that the converse directions are not provable without excluded middle.

Exercise 10.2.6 Make sure you can prove the double negations of the following propositions:

$$\begin{gathered}X \lor \neg X\\ \neg\neg X \to X\\ \neg(X \land Y) \to \neg X \lor \neg Y\\ (\neg X \to \neg Y) \to Y \to X\\ ((X \to Y) \to X) \to X\\ (X \to Y) \to \neg X \lor Y\\ (X \to Y) \lor (Y \to X)\end{gathered}$$

10.3 Definiteness and Stability

We define definiteness and stability of propositions as follows:

$$\begin{aligned} \mathsf{definite}\ X^{\mathbb{P}} &:= X \lor \neg X \\ \mathsf{stable}\ X^{\mathbb{P}} &:= \neg\neg X \to X \end{aligned}$$

Fact 10.3.1

1. Every definite proposition is stable.
2. Every negated proposition is stable.
3. $\top$ and $\bot$ are definite and stable.
4. Definiteness and stability are transported by propositional equivalence.
5. Under XM, all propositions are definite and stable.

Fact 10.3.2 Implication, conjunction, disjunction, and negation preserve definiteness:

1. $\mathsf{definite}\ X \to \mathsf{definite}\ Y \to \mathsf{definite}\,(X \to Y)$.
2. $\mathsf{definite}\ X \to \mathsf{definite}\ Y \to \mathsf{definite}\,(X \land Y)$.
3. $\mathsf{definite}\ X \to \mathsf{definite}\ Y \to \mathsf{definite}\,(X \lor Y)$.
4. $\mathsf{definite}\ X \to \mathsf{definite}\,(\neg X)$.

Fact 10.3.3 (Definite de Morgan) $\mathsf{definite}\ X \lor \mathsf{definite}\ Y \to \neg(X \land Y) \leftrightarrow \neg X \lor \neg Y$.

Fact 10.3.4 Implication, conjunction, and universal quantification preserve stability:

1. $\mathsf{stable}\ Y \to \mathsf{stable}\,(X \to Y)$.
2. $\mathsf{stable}\ X \to \mathsf{stable}\ Y \to \mathsf{stable}\,(X \land Y)$.
3. $(\forall a.\ \mathsf{stable}\,(pa)) \to \mathsf{stable}\,(\forall a.pa)$.

Exercise 10.3.5 Prove the above facts.

Exercise 10.3.6 Prove $\mathsf{Markov} \leftrightarrow \forall f^{\mathsf{N}\to\mathsf{B}}.\ \mathsf{stable}(\exists n.\ fn = \mathsf{T})$. The equivalence says that Markov is equivalent to stability of satisfiability of tests on numbers.

Exercise 10.3.7 Prove $(\forall a.\ \mathsf{stable}\,(pa)) \to \neg(\forall a.pa) \leftrightarrow \neg\neg\exists a.\neg pa$.

Exercise 10.3.8 We define **classical variants** of conjunction, disjunction, and existential quantification:

$$\begin{aligned} X \land_c Y &:= (X \to Y \to \bot) \to \bot && \neg(X \to \neg Y) \\ X \lor_c Y &:= (X \to \bot) \to (Y \to \bot) \to \bot && \neg X \to \neg\neg Y \\ \exists_c a.pa &:= (\forall a.pa \to \bot) \to \bot && \neg(\forall a.\neg pa) \end{aligned}$$

The definitions are obtained from the impredicative characterisations by replacing the quantified target proposition Z with $\bot$. At the right we give computationally equal variants using negation. The classical variants are implied by the originals and are equivalent to the double negations of the originals. Under excluded middle, the classical variants thus agree with the originals. Prove the following propositions.

a) $X \land Y \to X \land_c Y$ and $X \land_c Y \leftrightarrow \neg\neg(X \land Y)$.

b) $X \lor Y \to X \lor_c Y$ and $X \lor_c Y \leftrightarrow \neg\neg(X \lor Y)$.

c) $(\exists a.pa) \to \exists_c a.pa$ and $(\exists_c a.pa) \leftrightarrow \neg\neg(\exists a.pa)$.

d) $X \lor_c \neg X$.

e) $\neg(X \land_c Y) \leftrightarrow \neg X \lor_c \neg Y$.

f) $(\forall a.\ \mathsf{stable}\,(pa)) \to \neg(\forall a.pa) \leftrightarrow \exists_c a.\neg pa$.

g) $X \land_c Y$, $X \lor_c Y$, and $\exists_c a.\, pa$ are stable.

11 Informative Types

So far, we have used the propositions-as-types principle to transport intuitions from types to propositions. In this chapter, we will go the other way and transport intuitions from propositions to types, relying on the fact that the logical connectives and quantifiers all have meaningful computational versions. The situation can be depicted as follows:

$$\begin{array}{lcccccccl} \mathbb{T} & \forall & \to & \times & \Leftrightarrow & + & \Sigma & & \text{computational types} \\ \hline \mathbb{P} & \forall & \to & \land & \leftrightarrow & \lor & \exists & & \text{propositional types} \end{array}$$

The function types $\forall$ and $\to$ exist natively at both levels, where $\to$ is just a special form of $\forall$. The inductive definitions of $\land$, $\lor$, and $\exists$ are lifted from $\mathbb{P}$ to $\mathbb{T}$ by declaring the constructors with $\mathbb{T}$ in place of $\mathbb{P}$. The computational version of conjunctions are the familiar product types. The computational version $\Leftrightarrow$ of propositional equivalence $\leftrightarrow$ is defined with product $\times$ in place of conjunction $\land$. The computational versions of disjunctions and existential quantifications are known as **sum types** and **sigma types**. We will refer to the computational variants of propositional types as **informative types**.

A main motivation for using the computational versions $+$ and Σ of $\lor$ and $\exists$ is the fact that they are not affected by the elim restriction and thus can be used freely for defining functions.[1]

The correspondence between propositional types and their computational variants is so close that speaking of proofs of the respective computational types is meaningful and technically helpful.

The other main topic of this chapter are (computationally) decidable predicates. Given that all functions definable in Coq are computable, we can identify decidable predicates as predicates that are decidable with boolean functions. The study of decidable predicates profits much from the presence of sigma types and sum types. Sum types lead to decision types, which in turn lead to certifying deciders.

11.1 Boolean Deciders

A **boolean decider** for a predicate $p^{X \to \mathbb{P}}$ is a function $f^{X \to \mathsf{B}}$ such that

$$\forall x.\ px \leftrightarrow fx = \mathsf{T}$$

[1] Recall that $\land$ is excepted from the elim restriction (Section 6.10).

A predicate $p^{X\to\mathbb{P}}$ is **decidable** if it has a boolean decider. Formally, we define a predicate as follows:

$$\begin{aligned}\mathsf{bdec} :\ & \forall X.\ (X \to \mathbb{P}) \to (X \to \mathsf{B}) \to \mathbb{P}\\ \mathsf{bdec}\,Xpf\ :=\ & \forall x.\ px \leftrightarrow fx = \mathsf{T}\end{aligned}$$

We treat the first argument of bdec as implicit argument. Given a predicate $p^{X\to\mathbb{P}}$, the proposition $\mathsf{ex}(\mathsf{bdec}\,p)$ formally says that p is decidable.

Recall the discussion of computational omniscience in Section 9.5. There we agreed that the predicate

$$\begin{aligned}\mathsf{tsat} :\ & (\mathsf{N} \to \mathsf{B}) \to \mathbb{P}\\ \mathsf{tsat}\,f\ :=\ & \exists n.\ fn = \mathsf{T}\end{aligned}$$

is computationally undecidable. In fact, we have $\mathsf{CO} \approx \mathsf{ex}(\mathsf{bdec}\ \mathsf{tsat})$.

Fact 11.1.1 Decidable predicates $X \to \mathbb{P}$ are closed under implication, negation, conjunction, and disjunction. That is, if p and q are decidable predicates $X \to \mathbb{P}$, then so are $\lambda x.(px \to qx)$, $\lambda x.\neg px$, $\lambda x.(px \wedge qx)$, and $\lambda x.(px \vee qx)$.

Proof Straightforward. ■

Fact 11.1.2 Decidability of predicates transports through propositional equivalence. That is, $\forall X^{\mathbb{T}} p^{X\to\mathbb{P}} q^{X\to\mathbb{P}} f^{X\to\mathsf{B}}.\ (\forall x.\ px \leftrightarrow qx) \to \mathsf{bdec}\,pf \to \mathsf{bdec}\,qf$.

Proof Straightforward. ■

Exercise 11.1.3 Prove $\mathsf{ex}(\mathsf{bdec}\,p) \to \forall x.\ px \vee \neg px$.

Exercise 11.1.4 Prove $\mathsf{bdec}\,pf \to \mathsf{bdec}\,qf \to \forall x.\ px \leftrightarrow qx$.

Exercise 11.1.5 Prove the following equivalences:

$$\begin{aligned}\mathsf{CO}\ &\leftrightarrow\ \exists f.\ \mathsf{bdec}\,\mathsf{tsat}\,f\\ \mathsf{LPO}\ &\leftrightarrow\ \forall f.\ \mathsf{tsat}\,f \vee \neg\mathsf{tsat}\,f\\ \mathsf{Markov}\ &\leftrightarrow\ \forall f.\ \neg\neg\mathsf{tsat}\,f \to \mathsf{tsat}\,f\end{aligned}$$

Note that CO says that $\mathsf{tsat}\,f$ is decidable, that LPO says that $\mathsf{tsat}\,f$ is definite, and that Markov says that $\mathsf{tsat}\,f$ is stable (in each case for all tests f). Thus we may write the above equivalences as follows:

$$\begin{aligned}\mathsf{CO}\ &\leftrightarrow\ \mathsf{ex}\,(\mathsf{bdec}\ \mathsf{tsat})\\ \mathsf{LPO}\ &\leftrightarrow\ \forall f.\ \mathsf{definite}\,(\mathsf{tsat}\,f)\\ \mathsf{Markov}\ &\leftrightarrow\ \forall f.\ \mathsf{stable}\,(\mathsf{tsat}\,f)\end{aligned}$$

11.2 Certifying Deciders and Sum Types

Certifying deciders refine boolean deciders in that they yield certified decisions, carrying proofs of their correctness. Certifying deciders are based on sum types $X + Y$, which generalize disjunctions $X \vee Y$ from propositions to all types thus avoiding the elim restriction.

Sum types $X + Y$ are inductively defined as follows:

$$\begin{aligned} + &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{L} &: \forall XY.\, X \to X + Y \\ \mathsf{R} &: \forall XY.\, Y \to X + Y \end{aligned}$$

Using sum types, we define **decision types $\mathcal{D}(X)$** as follows:

$$\begin{aligned} &\mathcal{D} : \mathbb{P} \to \mathbb{T} \\ &\mathcal{D}(X) \;:=\; X + \neg X \end{aligned}$$

A value of a decision type $\mathcal{D}(X)$ is a **decision** carrying either a proof of X or a proof of $\neg X$. We say that a proposition is **decided** if its decision type is inhabited. Note that decided propositions are definite.

Fact 11.2.1 (Propagation) There are propagation functions

$$\begin{aligned} \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\; \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \to Y) \\ \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\; \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \wedge Y) \\ \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\; \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \vee Y) \\ \forall X^{\mathbb{P}}.\; \mathcal{D}(X) \to \mathcal{D}(\neg X) \end{aligned}$$

Moreover, there are decisions for the propositions $\bot$ and $\top$.

Proof Straightforward. ■

Fact 11.2.2 (Transport) Decisions transport through propositional equivalence. That is, there is a function $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\; (X \leftrightarrow Y) \to \mathcal{D}(X) \to \mathcal{D}(Y)$.

Proof Straightforward. ■

A **certifying decider** for a predicate $p^{X \to \mathbb{P}}$ is a function $\forall x^X.\, \mathcal{D}(px)$. From a certifying decider p we can obtain a boolean decider for p by forgetting the proofs. Vice versa, we can construct from a boolean decider and its correctness proof a certifying decider.

Fact 11.2.3 There is a function $\forall X^{\mathbb{T}} f^{X \to \mathsf{B}} x^X.\, \mathcal{D}(fx = \mathsf{T})$.

Proof Boolean case analysis on fx reduces the claim to $\mathcal{D}(\mathsf{T} = \mathsf{T})$ and $\mathcal{D}(\mathsf{F} = \mathsf{T})$. ∎

Fact 11.2.4 There is a function $\forall X^{\mathbb{T}} p^{X \to \mathbb{P}} f^{X \to \mathsf{B}}.\ \mathsf{bdec}\, p f \to \forall x. \mathcal{D}(px)$ translating boolean deciders into certifying deciders.

Proof Fact 11.2.2 and Fact 11.2.3. ∎

Fact 11.2.5 There is a function $\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ (\forall x. \mathcal{D}(px)) \to \mathsf{ex}(\mathsf{bdec}\, p)$ translating certifying deciders into boolean deciders.

Proof For $h^{\forall x. \mathcal{D}(px)}$ define $fx := \text{MATCH } hx\ [\,\mathsf{L}\,\_ \Rightarrow \mathsf{T} \mid \mathsf{R}\,\_ \Rightarrow \mathsf{F}\,]$. ∎

The translation function provided by Fact 11.2.5 has the problem that the boolean decider in the proof of $\mathsf{ex}(\mathsf{bdec}\, p)$ cannot be accessd computationally (because of the elim restriction). We will fix the problem with a computational version of existential quantification in a later section.

We can recover some of the lost symmetry between the translation functions with an **inhabitation predicate** for types:

$$\mathcal{I} : \mathbb{T} \to \mathbb{P}$$
$$\mathcal{I}(X) \ := \ \exists x^X. \top$$

Corollary 11.2.6 $\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ \mathsf{ex}(\mathsf{bdec}\, p) \ \leftrightarrow \ \mathcal{I}(\forall x. \mathcal{D}(px))$.

Proof Facts 11.2.5 and 11.2.4. ∎

Exercise 11.2.7 Prove $X + Y \to X \vee Y$ and $\mathcal{I}(X + Y) \leftrightarrow X \vee Y$.

Exercise 11.2.8 Prove $(\forall x.\ \mathcal{D}(px)) \to \forall x.\ px \vee \neg px$.

Exercise 11.2.9 Define a function $\forall b^{\mathsf{B}}.\ (b = \mathsf{T}) + (b = \mathsf{F})$.

Exercise 11.2.10 Prove
$\forall B^{\mathbb{T}} a^B b^B.\ (\forall p^{B \to \mathbb{T}}.\ pa \to pb \to \forall x.px) \Leftrightarrow (\forall x.\ (x = a) + (x = b))$.

Exercise 11.2.11 Show that a type X is discrete if and only if it has a certifying equality decider $\forall x^X y^X.\ \mathcal{D}(x = y)$. Discrete types are defined in Section 8.4.

Exercise 11.2.12 Show that a sum type $X+Y$ is discrete if both X and Y are discrete.

Exercise 11.2.13 Define a function $\forall X^{\mathbb{T}}.\ X \to \mathcal{I}(X)$.

Exercise 11.2.14 Define an inductive inhabitation predicate not using existential quantification and show that it is equivalent to the predicate $\mathcal{I}$ defined with existential quantification.

Exercise 11.2.15 Prove $x\ \&\ y = \mathsf{F} \Leftrightarrow (x = \mathsf{F}) + (y = \mathsf{F})$ and
$x \mid y = \mathsf{T} \Leftrightarrow (x = \mathsf{T}) + (y = \mathsf{T})$.

11.3 Sigma Types

Recall that a proof of an existential quantification $\exists x.px$ can be thought of as a pair (x, h) consisting of a witness x and a proof h of px. Because of the elim restriction, the witness cannot be accessed computationally. We now define a computational version $\Sigma x.px$ of existential quantification residing in $\mathbb{T}$ that is not affected by the elim restriction.

We define **Σ-types** inductively as follows:

$$\begin{aligned} \mathsf{sig} : \;& \forall X^{\mathbb{T}}.\,(X \to \mathbb{T}) \to \mathbb{T} \\ \mathsf{E} : \;& \forall X^{\mathbb{T}} q^{X \to \mathbb{T}} x^X.\; qx \to \mathsf{sig}\, X q \end{aligned}$$

We treat the first argument of sigand the first two arguments of E as implicit arguments and write $\Sigma x.s$ for $\mathsf{sig}(\lambda x.s)$. The elements $\mathsf{E}\,xa$ of Σ-types are called **dependent pairs**. This speak forgets about the parameters X and q and emphasizes the fact the type of the second component $a : qx$ depends on the first component x. Σ-types are also called **dependent pair types**.

With Σ-types we can specify the translation functions between boolean and certifying deciders concisely and symmetrically.

Fact 11.3.1 (Translation) There are functions as follows:

$$\begin{aligned} &\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\quad (\Sigma f.\,\mathsf{bdec}\,pf) \to (\forall x.\,\mathcal{D}(px)) \\ &\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\quad (\forall x.\,\mathcal{D}(px)) \to (\Sigma f.\,\mathsf{bdec}\,pf) \end{aligned}$$

Proof Follows with Facts 11.2.4, 11.2.3, and 11.2.2. ■

We go one step further and define **propositional equivalence $X \Leftrightarrow Y$** for types:

$$\begin{aligned} \Leftrightarrow \,:\; & \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ X \Leftrightarrow Y \;:=\; & (X \to Y) \times (Y \to X) \end{aligned}$$

Note that a proof of an equivalence $X \Leftrightarrow Y$ is a pair (f, g) of functions $f : X \to Y$ and $g : Y \to X$.

Corollary 11.3.2 $\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\; (\Sigma f.\,\mathsf{bdec}\,pf) \Leftrightarrow (\forall x.\,\mathcal{D}(px))$.

Exercise 11.3.3 Prove $(\Sigma x.px) \to (\exists x.px)$ and $\mathcal{I}(\Sigma x.px) \leftrightarrow (\exists x.px)$.

Exercise 11.3.4 Prove $\forall X^{\mathbb{P}}.\; \mathcal{D}(X) \Leftrightarrow \Sigma b.\, X \leftrightarrow b = \mathsf{T}$.

Exercise 11.3.5 Define a function $\forall n^{\mathsf{N}}.\,(n = 0) + (\Sigma k.\, n = \mathsf{S}k)$.

Exercise 11.3.6 Prove a computational version of Lawvere's theorem:
$\forall X^{\mathbb{T}} Y^{\mathbb{T}} f^{X \to X \to Y}.\ (\forall g\, \Sigma x.\ fx = g) \to (\forall g\, \Sigma y^{Y}.\ gy = y)$.

Exercise 11.3.7 The elimination lemma for booleans suffices for computational boolean case analysis. To study this issue we consider an abstract type of booleans.

a) Prove $\forall B^{\mathbb{T}} \forall a^B b^B.\ (\forall x.\ (x = a) + (x = b)) \Leftrightarrow (\forall p^{B \to \mathbb{T}}.\ pa \to pb \to \forall x.px)$.

b) Prove $\forall B^{\mathbb{T}} \forall a^B b^B \forall d^{\forall x.\ (x=a)+(x=b)}.\ a \neq b \to \Sigma f^{B \to B} \forall x.\ fx \neq x$.

Both claims have straightforward proofs, in particular when written in Coq. Note that the second claim says that one can define a function $B \to B$ not having a fixed point if one assumes that the two members of B are distinct.

Exercise 11.3.8 Define projections

$$\pi_1 :\ \forall X^{\mathbb{T}} p^{X \to \mathbb{T}}.\ \mathsf{sig}\, p \to X$$
$$\pi_2 :\ \forall X^{\mathbb{T}} p^{X \to \mathbb{T}} h^{\mathsf{sig}\, p}.\ p(\pi_1 h)$$

and do the following for $X^{\mathbb{T}}$ and $p^{X \to \mathbb{T}}$:

a) Prove the η-law $h = \mathsf{E}\, q(\pi_1 h)(\pi_2 h)$.

b) Prove injectivity of π_1: $\forall h^{\mathsf{sig}\, p}\, h'^{\mathsf{sig}\, p}.\ h = h' \to \pi_1 h = \pi_1 h'$.

c) Explain why $\forall h^{\mathsf{sig}\, p}\, h'^{\mathsf{sig}\, p}.\ h = h' \to \pi_2 h = \pi_2 h'$ doesn't type check.

d) Convince yourself that you cannot prove $\forall x^X a^{px} a'^{px}.\ \mathsf{E} xa = \mathsf{E} xa' \to a = a'$.

It turns out that the injectivity claim (d) can be shown assuming PI (proof irrelevance). In fact, a weaker version of PI known as Axiom K suffices. You'll find a proof in the Coq library EqdepFacts.

11.4 Constructing Functions in Proof Mode

Since certifying deciders have **informative types**,[2] they can be constructed like proofs using proof diagrams and tactics. For instance, a **certifying equality decider**

$$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ (x = y) + (x \neq y)$$

can be constructed with essentially the same diagram that is used in Section 6.5 for a proof of

$$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \vee x \neq y$$

The reasons for this coincidence are that the elimination lemma for numbers can be used computationally, and that sums are a computational version of disjunctions. As it comes to the Coq script, exactly the same script can be used for both proofs.

[2]By an informative type we mean a type specifying its inhabitants up to inessential details.

We define **comparisons** for numbers as follows:

$$\begin{aligned} x \le y &:= \exists k.\ x + k = y \\ x < y &:= \mathsf{S}x \le y \\ x \ge y &:= y \le x \\ x > y &:= y < x \end{aligned}$$

We can go one step further and construct a function

$$F:\ \forall x^{\mathsf{N}} y^{\mathsf{N}}.\ \ (\Sigma k.\ x + \mathsf{S}k = y) + (x = y) + (\Sigma k.\ x = y + \mathsf{S}k)$$

Given two numbers x and y, the function decides whether $x < y$ or $x = y$ or $y < x$. In addition, using dependent pairs, the function yields the differences $y - \mathsf{S}x$ and $x - \mathsf{S}y$ for the cases $x < y$ and $y < x$.

Given the informative type specifying the function F, the function can be constructed in proof mode, using induction on x followed by case analysis on y. As in the examples mentioned before, it is essential that y is quantified in the inductive hypothesis.

When we construct a term with a script in Coq, we can bind the constructed term to a constant, where we can choose between a **transparent binding** and an **opaque binding**. Choosing an opaque binding gives us a declared constant hiding its definition, and choosing a transparent binding gives us a defined constant where the term constructed with the script is taken as the term defining the constant. Using transparent bindings, we may construct reducible functions using scripts.

Defining the above constant F as a transparent constant is attractive, since from a transparent F we can easily get a reducible symmetric subtraction function

$$\mathsf{ssub}:\ \mathsf{N} \to \mathsf{N} \to \mathsf{B} \times \mathsf{N}$$

that yields $(\mathsf{T}, x - y)$ if $x \ge y$ and $(\mathsf{F}, y - x)$ if $y > x$.

Exercise 11.4.1 Construct a reducible symmetric subtraction function using proof diagrams and proof scripts.

Exercise 11.4.2 Construct certifying deciders as specified below using proof diagrams and scripts.

a) $\forall x^{\mathsf{B}} y^{\mathsf{B}}.\ \mathcal{D}(x = y)$.

b) $\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ \mathcal{D}(x = y)$.

c) $\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ (x < y) + (y \le x)$.

Exercise 11.4.3 Construct a dependent pair

$$\Sigma f^{\mathsf{N} \to \mathsf{N} \to \mathsf{B}}.\ \forall x y.\ f x y = \mathsf{T} \leftrightarrow x = y$$

using the certifying equality decider from Exercise 11.4.2. Make sure that the function f of the pair in fact computes (i.e., $f\,2\,3$ should reduce to F).

11.5 Discussion

So far, we have used the propositions-as-types principle to transport intuitions from types to propositions. In this chapter we have seen that transporting intuitions from propositions to types is also beneficial. At the type level, implication and universal quantification are native, and computational versions of conjunction, disjunction, propositional equivalence, and existential quantification are easily defined. It now makes sense to see the construction of inhabitants of informative types as proofs and to use the language coming with proofs for these constructions. In fact, proof diagrams and Coq's tactic language apply readily to the construction of inhabitants of types.

Technical Summary

$$\begin{aligned} \mathcal{D}(X) &:= X + \neg X \\ \mathsf{tsat}\, f &:= \exists n^{\mathsf{N}}.\, fn = \mathsf{T} \\ \mathsf{bdec}\, pg &:= \forall x^X.\, px \leftrightarrow gx = \mathsf{T} \\ \mathsf{CO} &:= \mathsf{ex}\,(\mathsf{bdec}\ \mathsf{tsat}) \end{aligned}$$

$$\begin{gathered} (\forall x.\, \mathcal{D}(px)) \Leftrightarrow \mathsf{sig}\,(\mathsf{bdec}\, p) \\ (\forall f.\, \mathcal{D}(\mathsf{tsat}\, f)) \to \mathsf{CO} \end{gathered}$$

Decisions travel through $\leftrightarrow$ and propagate through $\to$, $\neg$, $\wedge$, $\vee$.

12 Witness Operators

In this chapter we define a witness operator

$$\forall f^{\mathsf{N}\to\mathsf{B}}.\ (\exists n.\ fn = \mathsf{T}) \to (\Sigma n.\ fn = \mathsf{T})$$

That one can define a witness operator comes as a surprise. It seems that the operator bypasses the elim restriction for existential quantification. Technically, this is not the case. What happens is that the witness in the dependent pair is computed by checking fn for $n = 0, 1, 2, \dots$ until $fn = \mathsf{T}$ holds. We speak of *linear search* and *ascending recursion*. The witness in the satisfiability proof is used to enable the linear search through an inductive transfer predicate that is not affected by the elim restriction. The transfer predicate is defined with a single proof constructor using a higher-order form of recursion we call *guarded recursion*. The proof constructor is designed such that recursion on derivations supports the ascending recursion needed for linear search.

12.1 Transfer Predicate with Guarded Recursion

We fix a test $f : \mathsf{N} \to \mathsf{B}$ for the rest of the section.

We call a number n **guarded** if $f(n+k) = \mathsf{T}$ for some k:

$$\mathsf{guarded}\, n \ :=\ \exists k.\ f(n+k) = \mathsf{T}$$

Here are some straightforward facts about guardedness:

1. If $fn = \mathsf{T}$, then n is guarded.
2. If $\mathsf{S}n$ is guarded, then n is guarded.
3. If f is satisfiable, then 0 is guarded.
4. If n is guarded and $fn = \mathsf{F}$, then $\mathsf{S}n$ is guarded.

The trick is now to define an inductive predicate G with a single recursive proof constructor realizing facts (1) and (2) such that G holds exactly for the guarded numbers. The definition of the **transfer predicate** G is as follows:

$$\begin{aligned} \mathsf{G} &:\ \mathsf{N} \to \mathbb{P} \\ \mathsf{G_I} &:\ \forall n.\ (fn = \mathsf{F} \to \mathsf{G}\,(\mathsf{S}n)) \to \mathsf{G}\,n \end{aligned}$$

Note that the second argument of the proof constructor $\mathsf{G_I}$ is an implication

$$fn = \mathsf{F} \to \mathsf{G}\,(\mathsf{S}n)$$

The implication makes it possible to define the transfer predicate with a single proof constructor not invoking the elim restriction. Computationally, we say that the implication sets up a **guarded recursion** (where $fn = \mathsf{F}$ is the guard).

We may depict the proof constructor $\mathsf{G_I}$ as the proof rule

$$\frac{fn = \mathsf{F} \;\to\; \mathsf{G}\,(\mathsf{S}n)}{\mathsf{G}\,n}$$

which says that a derivation of $\mathsf{G}\,n$ is obtained from a function that given a proof of $fn = \mathsf{F}$ yields a derivation of $\mathsf{G}\,(\mathsf{S}n)$.

Fact 12.1.1

1. $fn = \mathsf{T} \to \mathsf{G}\,n$.
2. $\mathsf{G}\,(\mathsf{S}n) \to \mathsf{G}\,n$.
3. $\mathsf{G}\,n \to \mathsf{G}\,0$.
4. $(\exists n.\, fn = \mathsf{T}) \to \mathsf{G}\,0$.

Proof (1) follows with $\mathsf{G_I}$ since $fn = \mathsf{T}$ and $fn = \mathsf{F}$ are contradictory. (2) is a trivial consequence of $\mathsf{G_I}$. (3) follows by induction on n from (2). (4) follows from (1) and (3). ■

It remains to construct a recursive function

$$F : \;\forall n.\, \mathsf{G}\,n \to \Sigma k.\, fk = \mathsf{T}$$

We first observe that G is not subject to the elim restriction since all non-proof arguments of its single proof constructor appear in the target type of the proof constructor. Thus F can recurse on the derivation of $\mathsf{G}\,n$.

Fact 12.1.2 $\forall n.\, \mathsf{G}\,n \to \Sigma k.\, fk = \mathsf{T}$.

Proof We assume $\mathsf{G}\,n$ and prove $\Sigma k.\, fk = \mathsf{T}$ by induction on the derivation of $\mathsf{G}\,n$. Since $\mathsf{G_I}$ is the only proof constructor, we have $fn = \mathsf{F} \to \mathsf{G}\,(\mathsf{S}n)$ and

$$fn = \mathsf{F} \to \Sigma k.\, fk = \mathsf{T}$$

as inductive hypothesis. By boolean case analysis we have either $fn = \mathsf{T}$ or $fn = \mathsf{F}$. If $fn = \mathsf{T}$, the claim follows with $k = n$. If $fn = \mathsf{F}$, the claim follows with the inductive hypothesis. ■

We refer to the function provided by Fact 12.1.2 as *linear search function.*

The informal proof of Fact 12.1.2 deserves formal justification. To do so, we define an eliminator for G whose type justifies the induction we have used in the informal proof:[1]

$$\begin{aligned} &\mathsf{E_G} : \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ (\forall n.\ (fn = \mathsf{F} \to p(\mathsf{S}n)) \to pn) \to \forall n.\ \mathsf{G}\,n \to pn \\ &\mathsf{E_G}\,pgn\,(\mathsf{G_I}\,\_\,h) \ := \ gn(\lambda e.\ \mathsf{E_G}\,pg(\mathsf{S}n)(he)) \end{aligned}$$

Checking that the defining equation of $\mathsf{E_G}$ is well-typed is not difficult. The recursion on the derivation of $\mathsf{G}\,n$ we see in the defining equation of $\mathsf{E_G}$ counts as structurally recursive in Coq's type theory. Informally, we may say that every derivation he is smaller than the given derivation $\mathsf{G_I}\,nh$.

We now formalize the informal proof of Fact 12.1.2 with a proof diagram:

		$\forall n.\ \mathsf{G}\,n \to \Sigma k.\ fk = \mathsf{T}$	apply $\mathsf{E_G}$, intros
	$n : \mathsf{N}$		
	$IH : fn = \mathsf{F} \to \Sigma k.\ fk = \mathsf{T}$	$\Sigma k.\ fk = \mathsf{T}$	destruct fn
1	$H : fn = \mathsf{F}$		exact $IH\,H$
2	$H : fn = \mathsf{T}$		exists n, exact H

Theorem 12.1.3 $\forall f^{\mathsf{N}\to\mathsf{B}}.\ (\exists n.\ fn = \mathsf{T}) \to (\Sigma n.\ fn = \mathsf{T})$.

Proof Follows with Fact 12.1.2 and Fact 12.1.1 (4). Passing through $\mathsf{G}\,0$ is essential to not get in conflict with the elim restriction for the proof of $\exists n.\ fn = \mathsf{T}$. ■

Exercise 12.1.4 Prove $\mathsf{G}\,fn \leftrightarrow \exists k.\ f(n+k) = \mathsf{T}$.
Hint: For the direction $\leftarrow$ prove $\forall n.\ f(n+k) = \mathsf{T} \to \mathsf{G}\,fn$ by induction on k.

Exercise 12.1.5 Define a function $\forall fn.\ \mathsf{G}\,fn \to \Sigma k.\ f(n+k) = \mathsf{T}$.

Exercise 12.1.6 Show $\mathsf{tsat}\,f \leftrightarrow \mathsf{G}\,f\,0$.

Exercise 12.1.7 Define the eliminator $\mathsf{E_G}$ with FIX and MATCH. Note that FIX must be used with a leading argument so that the recursive function can receive the type $\forall n.\ \mathsf{G}\,fn \to pn$ needed for the recursive application.

Exercise 12.1.8 Prove $\forall f^{\mathsf{N}\to\mathsf{B}}.\ (\exists n.\ fn = \mathsf{T}) \Leftrightarrow (\Sigma n.\ fn = \mathsf{T})$.

Exercise 12.1.9 Construct a witness operator $\forall f^{\mathsf{B}\to\mathsf{B}}.\ (\exists x.\ fx = \mathsf{T}) \to (\Sigma x.\ fx = \mathsf{T})$ for tests on booleans.

[1] In fact, $\mathsf{E_G}$ gives us more than we need for Fact 12.1.2 since the claim $\Sigma k.\ fk = \mathsf{T}$ does not depend on n. Starting with Exercise 12.1.4, we will use $\mathsf{E_G}$ for claims that depend on n.

Exercise 12.1.10 Construct a witness operator $\forall p^{\mathsf{N} \to \mathbb{P}}.\ (\forall n.\ \mathcal{D}(pn)) \to \mathsf{ex}\, p \to \mathsf{sig}\, p$ for decidable predicates on numbers.

Exercise 12.1.11 Prove $\mathsf{tsat}\,(\lambda n.\, fn \mid gn) \Leftrightarrow \mathsf{tsat}\, f + \mathsf{tsat}\, g$.

Exercise 12.1.12 Prove the following impredicative characterizations for G.

a) $\mathsf{G}\, fn \leftrightarrow \forall p^{\mathsf{N} \to \mathbb{P}}.\ (\forall n.\ (fn = \mathsf{F} \to p(\mathsf{S}n)) \to pn) \to pn.$

b) $\mathsf{G}\, fn \leftrightarrow \forall p^{\mathsf{N} \to \mathbb{P}}.\ (\forall n.\ fn = \mathsf{T} \to pn) \to (\forall n.\ p(\mathsf{S}n) \to pn) \to pn.$

Note that (a) follows the inductive definition of G.

12.2 Discussion

With the transfer predicate G we have seen an inductive predicate that goes far beyond the inductive definitions we have seen so far.

1. G is defined with a recursive proof constructor.
2. The proof constructor of G employs guarded recursion, which greatly extends the power of structural recursion. Guarded recursion means that the recursive argument of a proof constructor is a function that for every argument yields a structurally smaller value. That this is the case and that guarded structural recursions always terminate is a basic fact (or assumption if you wish) that is build-in in Coq's type theory.
3. The parameter n of G is nonuniform. While G can be defined with the parameter f abstracted out (e.g., as a section variable in Coq), the parameter n cannot be abstracted out since the proof constructor employs $\mathsf{G}\, f\, (\mathsf{S}n)$ in its argument type.
4. Due to the nonuniform parameter and the guarded recursion, the recursive eliminator we defined for G has a very particular form we have not seen before.
5. The eliminator we have defined for G is not the strongest one. One can define a stronger eliminator where the target type depends on both n and the derivation $h : \mathsf{G}n$. This eliminator makes it possible to prove properties of a linear search function $\forall n.\ \mathsf{Gn} \to \mathsf{N}$ with a noninformative target type.
6. The construction of the witness operator crucially relies on two exemptions from the elim restriction: That one can freely match on proofs of $\bot$ and on proofs of $\mathsf{G}n$.

12.3 Unsuccessful Variations

We look at two variants of the inductive transfer predicate that support ascending recursion but fail to deliver computational elimination because of the elim restriction.

Exercise 12.3.1 One feature of the transfer predicate G is that it supports an ascending recursion $n = 0, 1, 2, \ldots$ until $fn = \mathsf{T}$ holds. Here is a more conventional inductive predicate supporting the ascending recursion:

$$\begin{aligned}
\mathsf{G}' &: \mathsf{N} \to \mathbb{P} \\
\mathsf{G}'_\mathsf{B} &: \forall n.\, fn = \mathsf{T} \to \mathsf{G}'\, n \\
\mathsf{G}'_\mathsf{S} &: \forall n.\, \mathsf{G}'(\mathsf{S}n) \to \mathsf{G}'\, n
\end{aligned}$$

Note that G' is subject to the elim restriction since it has two proof constructors. Thus the ascending recursion licensed by a derivation of $\mathsf{G}'\, n$ cannot be used computationally.

a) Prove $\mathsf{G}\, n \to \mathsf{G}'\, n$.

b) Define an eliminator for G'.

c) Prove $\mathsf{G}'\, n \to \mathsf{G}\, n$ using the eliminator from (b).

Exercise 12.3.2 Here is another candidate for a transfer predicate:

$$\begin{aligned}
\mathsf{G}'' &: \mathsf{N} \to \mathbb{P} \\
\mathsf{G}''_\mathsf{I} &: \forall n.\, (fn = \mathsf{T} \vee \mathsf{G}''(\mathsf{S}n)) \to \mathsf{G}''\, n
\end{aligned}$$

Note that G'' is not affected by the elim restriction. However, it seems impossible to define a computational eliminator since the proof of the disjunction appearing as argument of G''_I must be destructured.

a) Prove $\mathsf{G}\, n \to \mathsf{G}''\, n$.

b) Define an eliminator
$\forall p^{\mathsf{N}\to\mathbb{P}}.\, (\forall n.\, fn = \mathsf{F} \to pn) \to (\forall n.\, p(\mathsf{S}n) \to pn) \to \forall n.\, \mathsf{G}''\, n \to pn.$

c) Prove $\mathsf{G}''\, n \to \mathsf{G}\, n$ using the eliminator from (b).

Technical Summary

$$\begin{gathered}
(\exists n^{\mathsf{N}}.\, fn = \mathsf{T}) \Leftrightarrow (\Sigma n.\, fn = \mathsf{T}) \\
(\forall n^{\mathsf{N}}.\, \mathcal{D}(pn)) \to (\mathsf{ex}\, p \Leftrightarrow \mathsf{sig}\, p) \\
\mathsf{tsat}(\lambda n.\, fn \mid gn) \Leftrightarrow \mathsf{tsat}\, f + \mathsf{tsat}\, g
\end{gathered}$$

13 Semi-Decidability

The theory of computation distinguishes between decidable and semi-decidable predicates, where Post's theorem says that a predicate is decidable if and only if both the predicate and its complement are semi-decidable. There are important semi-decidable problems that are undecidable. It turns out that semi-decidability has an elegant definition in type theory using semi-decision types, and that Post's theorem is equivalent to Markov's principle, where the direction from Markov to Post needs a witness operator.

13.1 Semi-Deciders

A **semi-decider** for a predicate $p^{X \to \mathbb{P}}$ is a function $F^{X \to \mathsf{N} \to \mathsf{B}}$ such that

$$\forall x.\ px \leftrightarrow \mathsf{tsat}(Fx)$$

A predicate $p^{X \to \mathbb{P}}$ is **semi-decidable** if it has a semi-decider.

We offer two intuitions for semi-deciders. Let F be a semi-decider for p. This means we have $px \leftrightarrow \exists n.\ Fxn = \top$ for every x. The *fuel intuition* says that F confirms px if and only if px holds and F is given enough fuel n. The *proof intuition* says that the proof system F has a proof n of px if and only if px holds.

Fact 13.1.1 Decidable predicates are semi-decidable.

Fact 13.1.2 tsat is semi-decidable.

We define **semi-decision types** $\mathcal{S}(X)$ as follows:

$$\begin{aligned} &\mathcal{S} : \mathbb{P} \to \mathbb{T} \\ &\mathcal{S}(X) := \Sigma f^{\mathsf{N} \to \mathsf{B}}.\ X \leftrightarrow \mathsf{tsat}\, f \end{aligned}$$

Fact 13.1.3 $\forall X^{\mathbb{P}}.\ \mathcal{D}(X) \to \mathcal{S}(X)$.

Fact 13.1.4 (Transport) $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ (X \leftrightarrow Y) \to \mathcal{S}(X) \to \mathcal{S}(Y)$.

Fact 13.1.5 $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ \mathcal{S}(X) \to \mathcal{S}(Y) \to \mathcal{S}(X \vee Y)$.

Proof Let f be the test for X and g be the test for Y. Then $\lambda n.fn \mid gn$ is a test for $X \vee Y$. ■

Fact 13.1.6 $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ \mathcal{S}(X) \to \mathcal{S}(Y) \to \mathcal{S}(X \wedge Y)$.

Proof Let f be the test for X and g be the test for Y. We assume a pairing function for numbers. Let $F : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$, $\pi_1 : \mathsf{N} \to \mathsf{N}$ and $\pi_2 : \mathsf{N} \to \mathsf{N}$ such that $\pi_1(Fn_1n_2) = n_1$ and $\pi_2(Fn_1n_2) = n_2$. Then $\lambda n.f(\pi_1 n) \& g(\pi_2 n)$ is a test for $X \wedge Y$. ■

A **certifying semi-decider** for a predicate $p^{X \to \mathbb{P}}$ is a function $\forall x^X.\ \mathcal{S}(px)$. From a certifying semi-decider for p we can obtain a semi-decider for p by forgetting the proofs. Vice versa, we can construct from a semi-decider and its correctness proof a certifying semi-decider.

Fact 13.1.7 $\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ (\forall x. \mathcal{S}(px)) \Leftrightarrow (\Sigma F^{X \to \mathsf{N} \to \mathsf{B}}\, \forall x.\ px \leftrightarrow \mathsf{tsat}\,(Fx))$.

Proof Direction $\Rightarrow$. We assume $H : \forall x. \mathcal{S}(px)$ and $x : X$ and show $px \leftrightarrow \mathsf{tsat}(Fx)$ for $Fx := \pi_1(Hx)$. Straightforward.

Direction $\Leftarrow$. We assume $H : \forall x.\ px \leftrightarrow \mathsf{tsat}\,(Fx)$ and $x : X$ and show $\mathcal{S}(px)$. Trivial with Fx as test. ■

Corollary 13.1.8 A predicate is semi-decidable iff it has a certifying semi-decider.

Recall the discussion of computational omniscience in Section 9.5. It turns out that from a decider for tsat we can get a function translating semi-decisions into decisions, and vice versa.

Fact 13.1.9 $(\forall X^{\mathbb{P}}.\ \mathcal{S}(X) \to \mathcal{D}(X)) \Leftrightarrow (\forall f^{\mathsf{N} \to \mathsf{B}}.\ \mathcal{D}(\mathsf{tsat}\, f))$.

Proof Direction $\Rightarrow$ follows since f is a test for $\mathcal{S}(\mathsf{tsat}\, f)$. For direction $\Leftarrow$ we assume $X \leftrightarrow \mathsf{tsat}\, f$ and show $\mathcal{D}(X)$. By the primary assumption we have either $\mathsf{tsat}\, f$ or $\neg\mathsf{tsat}\, f$. Thus $\mathcal{D}(X)$. ■

13.2 Markov-Post Equivalence

Recall the definition of Markov's principle:

$$\mathsf{Markov} := \forall f^{\mathsf{N} \to \mathsf{B}}.\ \neg(\forall n.\ fn = \mathsf{F}) \to (\exists n.\ fn = \mathsf{T})$$

We also need the function type

$$\mathsf{Post} := \forall X^{\mathbb{P}}.\ \mathcal{S}(X) \to \mathcal{S}(\neg X) \to \mathcal{D}(X)$$

We will refer to functions of type Post as **Post operators**.[1]

[1] Post operators are named after Emil Post, who first showed that predicates are decidable if they are semi-decidable and co-semi-decidable.

Fact 13.2.1 Post $\to \forall X^{\mathbb{P}}.\ \mathcal{D}(X) \Leftrightarrow \mathcal{S}(X) \times \mathcal{S}(\neg X)$.

Proof Straightforward. ■

Fact 13.2.2 Markov $\to$ Post.

Proof Assume Markov. Let f be a test for X and g be a test for $\neg X$. We show $\mathcal{D}(X)$. Let $hn := fn \mid gn$. It suffices to show $\Sigma n.\ hn = \mathsf{T}$. Since we have a witness operator and Markow, it suffices to show $\neg\forall n.\ hn = \mathsf{F}$. We assume $H : \forall n.\ hn = \mathsf{F}$ and show $\neg X$ and $\neg\neg X$. Follows since H implies that f and g are constantly false. ■

Fact 13.2.3 Post $\to$ Markov.

Proof We assume Post and $H : \neg\neg\mathsf{tsat}\, f$ and prove $\mathsf{tsat}\, f$ (using Exercise 11.1.5). It suffices to show $\mathcal{D}(\mathsf{tsat}\, f)$. Using Post it suffices to show $\mathcal{S}(\mathsf{tsat}\, f)$ and $\mathcal{S}(\neg\mathsf{tsat}\, f)$. $\mathcal{S}(\mathsf{tsat}\, f)$ holds with f as test. $\mathcal{S}(\neg\mathsf{tsat}\, f)$ holds with $\lambda\_.\mathsf{F}$ as test. ■

Theorem 13.2.4 Markov $\Leftrightarrow$ Post.

Proof Facts 13.2.2 and 13.2.3. ■

We say that a predicate p is **co-semi-decidable** if its **complement** $\overline{p} := \lambda x.\neg px$ is semi-decidable.

Corollary 13.2.5 Given Markov, a predicate is decidable iff it is semi-decidable and co-semi-decidable.

Corollary 13.2.6 Given Markov and $\neg$CO, tsat is not co-semi-decidable.

Proof Suppose that tsat is co-semi-decidable. Since tsat is semi-decidable (Fact 13.1.2), tsat is decidable by Fact 13.2.2. Contradiction with $\neg$CO. ■

Exercise 13.2.7 Prove $\forall X^{\mathbb{P}}.\ (X \vee \neg X) \to \mathcal{S}(X) \to \mathcal{S}(\neg X) \to \mathcal{D}(X)$.

Exercise 13.2.8 Prove Markov $\to \forall X.\ \mathcal{D}(X) \Leftrightarrow \mathcal{S}(X) \times \mathcal{S}(\neg X)$.

Exercise 13.2.9 Prove Markov $\to (\forall X.\ \mathcal{S}(X) \to \mathcal{S}(\neg X)) \to$ CO. Note that this implies that semi-decisions don't propagate through implications and negations if Markov and $\neg$CO are assumed.

13.3 Reductions

The theory of computation employs so-called many-one reductions to transport undecidability results between problems. In our setting problems are predicates and many-one reductions can be easily defined since all functions are computable.

Given two predicates $p^{X \to \mathbb{P}}$ and $q^{Y \to \mathbb{P}}$, a **reduction from p to q** is a function $f^{X \to Y}$ such that $\forall x.\ px \leftrightarrow q(fx)$. Formally, we define a predicate as follows:

$$\begin{aligned}
&\mathsf{red} : \ \forall X^{\mathbb{T}} Y^{\mathbb{T}}.\ (X \to \mathbb{P}) \to (Y \to \mathbb{P}) \to (X \to Y) \to \mathbb{P} \\
&\mathsf{red}\ XYpqf \ := \ \forall x.\ p(x) \leftrightarrow q(fx)
\end{aligned}$$

We treat the polymorphic arguments of red as implicit arguments.

Fact 13.3.1 Decidability and undecidability propagate along reductions as follows.

$$\begin{gathered}
\mathsf{red}\, pqf \to (\forall y. \mathcal{D}(qy)) \to (\forall x. \mathcal{D}(px)) \\
\mathsf{red}\, pqf \to (\forall y. \mathcal{S}(qy)) \to (\forall x. \mathcal{S}(px)) \\
\mathsf{ex}\,(\mathsf{red}\, pq) \to \mathsf{ex}\,(\mathsf{bdec}\, q) \to \mathsf{ex}\,(\mathsf{bdec}\, p) \\
\mathsf{ex}\,(\mathsf{red}\, pq) \to \neg\mathsf{ex}\,(\mathsf{bdec}\, p) \to \neg\mathsf{ex}\,(\mathsf{bdec}\, q)
\end{gathered}$$

Proof Straightforward. ∎

Fact 13.3.2 A predicate is semi-decidable if and only if it reduces to tsat. Formally: $\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ (\forall x.\ \mathcal{S}(px)) \Leftrightarrow \mathsf{sig}\,(\mathsf{red}\, p\, \mathsf{tsat})$.

Proof Direction $\Rightarrow$ follows with the reduction mapping x to the test for $\mathsf{S}(px)$. Direction $\Leftarrow$ uses the test the reduction yields for x. ∎

Exercise 13.3.3 The reducibility relation between predicates is reflexive and transitive. Prove $\mathsf{red}\, pp(\lambda x.x)$ and $\mathsf{red}\, pqf \to \mathsf{red}\, qrg \to \mathsf{red}\, pr(\lambda x.g(fx))$ to establish this claim.

Exercise 13.3.4 Prove $\mathsf{red}\ p\ q\ f \to \mathsf{red}\ \overline{q}\ \overline{p}\ f$.

Technical Summary

$$\begin{aligned}
\mathcal{D}(X) &:= X + \neg X \\
\mathcal{S}(X) &:= \Sigma f.\, X \leftrightarrow \mathsf{tsat}\, f \\
\mathsf{tsat}\, f &:= \exists n^{\mathsf{N}}.\, fn = \mathsf{T} \\
\mathsf{bdec}\, pg &:= \forall x^{X}.\, px \leftrightarrow gx = \mathsf{T} \\
\mathsf{CO} &:= \mathsf{ex}\,(\mathsf{bdec}\ \mathsf{tsat}) \\
\mathsf{Markov} &:= \forall f.\, \neg(\forall n.\, fn = \mathsf{F}) \to \mathsf{tsat}\, f \\
\mathsf{Post} &:= \forall X.\, \mathcal{S}(X) \to \mathcal{S}(\neg X) \to \mathcal{D}(X) \\
\mathsf{red}\, pqf &:= \forall x.\, px \leftrightarrow q(fx)
\end{aligned}$$

$$\begin{gathered}
\mathsf{tsat}\, f \Leftrightarrow \Sigma n.\, fn = \mathsf{T} \\
\mathsf{tsat}(\lambda n.\, fn \mid gn) \Leftrightarrow \mathsf{tsat}\, f + \mathsf{tsat}\, g \\
(\forall n^{\mathsf{N}}.\, \mathcal{D}(pn)) \to \mathsf{ex}\, p \to \mathsf{sig}\, p \\
\mathsf{sig}\,(\mathsf{bdec}\, p) \Leftrightarrow \forall x.\, \mathcal{D}(px) \\
(\forall f.\, \mathcal{D}(\mathsf{tsat}\, f)) \to \mathsf{CO} \\
\mathcal{D}(X) \to \mathcal{S}(X) \times \mathcal{S}(\neg X) \\
\mathcal{S}(\mathsf{tsat}\, f) \\
(\forall f.\, \mathcal{D}(\mathsf{tsat}\, f)) \Leftrightarrow \forall X.\, \mathcal{S}(X) \to \mathcal{D}(X) \\
\mathsf{Markov} \Leftrightarrow \mathsf{Post} \\
\mathsf{Markov} \to (\mathcal{D}(X) \Leftrightarrow \mathcal{S}(X) \times \mathcal{S}(\neg X)) \\
\mathsf{Markov} \to (\forall f.\, \mathcal{S}(\neg \mathsf{tsat}\, f)) \to \mathsf{CO} \\
(\forall x.\, \mathcal{S}(px)) \Leftrightarrow \mathsf{sig}\,(\mathsf{red}\, p\ \mathsf{tsat})
\end{gathered}$$

- Decisions propagate through $\to$, $\neg$, $\wedge$, $\vee$.
- Semi-decisions propagate through $\wedge$, $\vee$. (conjunction needs pairing function)
- Decisions and semi-decisions travel through $\leftrightarrow$.
- Deciders and semi-deciders travel from target to source of reductions.

14 More on Numbers

Numbers are the most basic infinite data structure there is. We derive a few hand-picked results about numbers to show important ideas. There is much beauty in deriving results about numbers from first principles. The few results we cover include complete induction, the division theorem, and a bounded μ-operator. Complete induction gives us a computational principle providing for the definition of many functions that do not have natural definitions with structural recursion.

14.1 Addition

We have seen the inductive definition of the numbers with the constructors $\mathsf{N}^{\mathbb{T}}$, 0^{N}, and $\mathsf{S}^{\mathsf{N}\to\mathsf{N}}$ in Chapter 1. The inductive definition provides for recursive definitions of the basic operations for addition, subtraction, and multiplication, and, more generally, for the recursive definition of an eliminator providing for inductive proofs (Chapter 6). Figure 14.1 collects the basic definitions for numbers.

The two basic facts about addition are **associativity** and **commutativity**.

Fact 14.1.1 $(x + y) + z = x + (y + z)$ and $x + y = y + x$.

Proof Associativity follows by induction on x. Commutativity also follows by induction on x, where the lemmas $x + 0 = x$ and $x + \mathsf{S}y = \mathsf{S}x + y$ are needed. Both lemmas follow by induction on x. ■

We will use associativity and commutativity of addition tacitly in proofs. If we omit parentheses for convenience, they are inserted from the left: $x + y + z \rightsquigarrow (x + y) + z$.

Another important fact about numbers is injectivity, which comes in two flavors.

Fact 14.1.2 (Injectivity) $x + y = x + z \to y = z$ and $x + k = x \to k = 0$.

Proof Both claims follow by induction on x. ■

Exercise 14.1.3 Prove $x + y \le x + z \to y \le z$.

Exercise 14.1.4 Prove that multiplication is commutative.

Exercise 14.1.5 Prove $x \neq x + \mathsf{S}k$.

Inductive Definition

$$\mathsf{N}^{\mathbb{T}}, \quad 0^{\mathsf{N}}, \quad \mathsf{S}^{\mathsf{N}\to\mathsf{N}}$$

Addition, Subtraction, Multiplication

$$\begin{aligned} + : \mathsf{N} \to \mathsf{N} \to \mathsf{N} & \\ 0 + y &:= y \\ \mathsf{S}x + y &:= \mathsf{S}(x + y) \end{aligned} \qquad \begin{aligned} \cdot : \mathsf{N} \to \mathsf{N} \to \mathsf{N} & \\ 0 \cdot y &:= 0 \\ \mathsf{S}x \cdot y &:= y + x \cdot y \end{aligned}$$

$$\begin{aligned} - : \mathsf{N} \to \mathsf{N} \to \mathsf{N} & \\ 0 - y &:= 0 \\ \mathsf{S}x - 0 &:= \mathsf{S}x \\ \mathsf{S}x - \mathsf{S}y &:= x - y \end{aligned}$$

Eliminator

$$\begin{aligned} \mathsf{E_N} : \ & \forall p^{\mathsf{N}\to\mathbb{T}}.\ p\,0 \to (\forall x.\ px \to p(\mathsf{S}x)) \to \forall x.px \\ \mathsf{E_N}\, paf\, 0 &:= a \\ \mathsf{E_N}\, paf\, (\mathsf{S}x) &:= fx(\mathsf{E_N}\, pafx) \end{aligned}$$

Linear Order

$$x \le y := \exists k.\ x + k = y$$

Figure 14.1: Basic definitions for numbers

14.2 Order

We define the order relation on numbers as follows:

$$x \le y := \exists k.\ x + k = y$$

We also define the following notational variants:

$$\begin{aligned} x < y &:= \mathsf{S}x \le y \\ x \ge y &:= y \le x \\ x > y &:= y < x \end{aligned}$$

We list the basic facts about order we will use in the following (mostly tacitly).

Fact 14.2.1

1. $x \le x$ (reflexivity)
2. $x \le y \to y \le z \to x \le z$ (transitivity)
3. $x \le y \to y \le x \to x = y$ (antisymmetry)
4. $\mathsf{S}x \le \mathsf{S}y \leftrightarrow x \le y$ (shift)
5. $0 \le x$ (origin)
6. $x \le 0 \to x = 0$ (origin)
7. $\neg(x < 0)$ (origin)
8. $\neg(x < x)$ (strictness)
9. $\neg(x + k < x)$ (strictness)
10. $x \le x + y$
11. $x \le y \to x \le \mathsf{S}y$
12. $x \le y \to x \le y + k$

Proof All claims have straightforward proofs not using induction. In a few cases, commutativity and associativity of addition are needed. Antisymmetry and strictness need injectivity of addition. ■

14.3 Linearity

Linearity is an essential fact about numbers saying that for two numbers x and y we have always either $x \le y$ or $y < x$. We prove the computational version of linearity from which we derive that bounded quantification preserves decidability. Although linearity and its consequences do have the flavor of excluded middle, they do have straightforward constructive proofs.

Fact 14.3.1 (Linearity) $(x \le y) + (y < x)$.

Proof By induction on x with y quantified in the inductive hypothesis, followed by case analysis on y in the successor case. In the successor-successor case, the claim follows with the shift law from the inductive hypothesis. ■

Corollary 14.3.2 (Contraposition) $\neg(x < y) \to y \le x$.

Corollary 14.3.3 (Equality by Contradiction) $\neg(x < y) \to \neg(y < x) \to x = y$.

Proof Follows by contraposition and antisymmetry. ■

Fact 14.3.4 $x \le y \Leftrightarrow (x < y) + (x = y)$.

Proof Direction $\Leftarrow$ is straightforward. For $\Rightarrow$ we assume $x \leq y$. By linearity we have either $y \leq x$ or $x < y$. If $y \leq x$, antisymmetry gives us $x = y$. ■

Fact 14.3.5 **Bounded quantification** preserves decidability:

1. $(\forall x. \mathcal{D}(px)) \to \mathcal{D}(\forall x.\, x < k \to px)$.
2. $(\forall x. \mathcal{D}(px)) \to \mathcal{D}(\exists x.\, x < k \to px)$.

Proof By induction on k and Fact 14.3.4. ■

Exercise 14.3.6 We define **divisibility** and **primality** as follows:

$$\begin{aligned} k \mid x \ &:= \ k > 0 \land \exists n.\, x = n \cdot k \\ \mathsf{prime}\, x \ &:= \ x \geq 2 \land \forall k.\, k \mid x \to k = 1 \lor k = x \end{aligned}$$

Prove that both predicates are decidable. Hint: First prove

$$\begin{aligned} & x > 0 \to x = n \cdot k \to n \leq x \\ & x > 0 \to k \mid x \to k \leq x \end{aligned}$$

and then exploit that bounded quantification preserves decidability.

14.4 Truncating Subtraction

There is a strong connection between order and truncating subtraction.

Fact 14.4.1 $x - (x + y) = 0$.

Proof By induction on x. ■

Fact 14.4.2 $x \leq y \ \leftrightarrow \ x - y = 0$.

Proof Direction $\to$ follows with Fact 14.4.1. Direction $\leftarrow$ follows by induction on x with y quantified. ■

Corollary 14.4.3 $x \leq y$ is decidable.

Fact 14.4.4

1. $(x + y) - x = y$
2. $x < y \to y - \mathsf{S}x < y$
3. $x \leq y \to x + (y - x) = y$

Proof The first claim follows by induction on x. The second and third claim follow with the first claim. ■

Exercise 14.4.5 Prove $\mathcal{D}(x \leq y)$. Give two proofs, one using Fact 14.4.2 and one not using subtraction.

Exercise 14.4.6 Define a boolean decider for $x \leq y$ and prove its correctness.

Exercise 14.4.7 Prove that equality of numbers is decidable using antisymmetry and decidability of $x \leq y$.

Exercise 14.4.8 Define a witness operator $x \leq y \to \Sigma k.\, x + k = y$.

14.5 Complete Induction and Size Induction

Complete induction says that when we prove px for a number x we may assume py for every number $y < x$. This strengthens structural induction which give us p only for the predecessor of x. It turns out that complete induction can be obtained from structural induction with a straightforward proof.

Fact 14.5.1 (Complete Induction)
$\forall p^{\mathsf{N} \to \mathbb{T}}.\, (\forall x.\, (\forall y.\, y < x \to py) \to px) \to \forall x.\, px.$

Proof Assume $H : \forall x.\, (\forall y.\, y < x \to py) \to px$. By H it suffices to prove $\forall y.\, y < x \to py$, which we do by induction on x. The base case is trivial. For the successor case we have $IH : \forall y.\, y < x \to py$ and $H_1 : y < \mathsf{S}x$ and need to show py. With H we get $H_2 : z < y$ and need to show pz. By IH it suffices to show $z < x$. Follows from H_1 and H_2 with transitivity and shift. ■

We will see soon that the computational variant of complete induction we have shown provides for the definition of functions that cannot be naturally defined by structural recursion. We speak of a computational variant of complete induction since the target type function is not restricted to propositions.

There is a useful generalisation of complete induction that applies to any type with a size function.

Fact 14.5.2 (Size Induction)
$\forall X^{\mathbb{T}}\, f^{X \to \mathsf{N}}\, p^{X \to \mathbb{T}}.\, (\forall x.\, (\forall y.\, fy < fx \to py) \to px) \to \forall x.\, px.$

Proof Similar to the proof of Fact 14.5.1. ■

Exercise 14.5.3 Obtain complete induction as an instance of size induction.

14.6 Division Theorem

Given two numbers x and y, there exist unique numbers a and $b \leq y$ such that $x = a \cdot \mathsf{S}y + b$. One speaks of the *integer quotient* and the *remainder* of x and y. We will define functions (often called *div* and *mod* in programming languages) that for x and y yield a and b. The idea is to subtract $\mathsf{S}y$ from x as often as is possible without truncation. Then the number of subtractions is a and the remaining number is b. Note that we consider $x = a \cdot \mathsf{S}y + b$ rather than $x = a \cdot y + b$ to avoid the side condition $y > 0$.

Fact 14.6.1 (Uniqueness)
If $a \cdot \mathsf{S}y + b = a' \cdot \mathsf{S}y + b'$ and $b, b' \leq y$, then $a = a'$ and $b = b'$.

Proof By induction on a with a' quantified in the inductive hypothesis, followed in both cases by case analysis on a'. The case $a = a' = 0$ is trivial. The cases where a and a' are not both 0 or successors are contradictory since $\neg(\mathsf{S}y + k \leq y)$. The case where a and a' are both successors follows from the inductive hypothesis. ■

Fact 14.6.2 (Existence) $\forall xy\, \Sigma ab.\ x = a \cdot \mathsf{S}y + b \land b \leq y$.

Proof We fix y and prove the claim by complete induction on x and case analysis on $(x \leq y) + (y < x)$. If $x \leq y$, then $a = 0$ and $b = x$ satisfy the claim. Otherwise, we have $y < x$. Thus $x - \mathsf{S}y < x$ by Fact 14.4.4 The inductive hypothesis gives us a and $b \leq y$ such that $x - \mathsf{S}y = a \cdot \mathsf{S}y + b$. Using Fact 14.4.4, we have $x = \mathsf{S}y + (x - \mathsf{S}y) = \mathsf{S}y + a \cdot \mathsf{S}y + b = \mathsf{S}a \cdot \mathsf{S}y + b$. The claim follows.

Note that the proof works for Σ since complete induction and case analysis with $(x \leq y) + (y < x)$ are computational. ■

Corollary 14.6.3 There are functions $D, M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that

$$x = Dxy \cdot \mathsf{S}y + Mxy \quad \text{and} \quad Mxy \leq y$$

for all numbers x and y.

Proof Let F be the function provided by Fact 14.6.2. Then $Dxy := \pi_1(Fxy)$ and $Mxy := \pi_1(\pi_2(Fxy))$ are functions as required. ■

Exercise 14.6.4 Prove $\mathsf{S}(2 \cdot x) \neq 2 \cdot y$. Make sure you know the proof idea for each of the facts in the following row:

$$\mathsf{S}x \neq 0 \qquad \mathsf{S}x \neq x \qquad \mathsf{S}(2 \cdot x) \neq 2 \cdot y \qquad \mathsf{S}(x \cdot \mathsf{S}(\mathsf{S}k)) \neq y \cdot \mathsf{S}(\mathsf{S}k)$$

Exercise 14.6.5 Recall divisibility $k \mid x$ as defined in Exercise 14.3.6.

1. Use the division theorem to show that $k \mid x$ is decidable.
2. Show $\mathsf{S}k \mid x \leftrightarrow Mxk = 0$ for a modulo function M.

Exercise 14.6.6 Let $\mathsf{even}\, n := \exists k.\ n = k \cdot 2$. Prove the following:

a) $\mathcal{D}\,(\mathsf{even}\, n)$.

b) $\mathsf{even}\, n \;\rightarrow\; \neg\mathsf{even}\,(\mathsf{S}n)$.

c) $\neg\mathsf{even}\, n \;\rightarrow\; \mathsf{even}\,(\mathsf{S}n)$.

Exercise 14.6.7 Let D and M be functions $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that

$$x = Dxy \cdot \mathsf{S}y + Mxy \quad \text{and} \quad Mxy \le y$$

for all numbers x and y. Prove the following facts providing for the recursive computation of D and M.

a) D and M are unique up to functional extensionality.

b) $x \le y \to Dxy = 0 \wedge Mxy = x$.

c) $y < x \to Dxy = \mathsf{S}(D(x - \mathsf{S}y)y) \wedge Mxy = M(x - \mathsf{S}y)y$.

Hint: In each case use Fact 14.6.1.

Exercise 14.6.8 It is possible to define D and M with structural recursion on numbers where the recursion is on an extra argument z such that $x < z$. Here are equational definitions of the functions:

$$\begin{aligned}
&D : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\quad Dxy0 := 0 \\
&Dxy(\mathsf{S}z) := 0 && \text{if } x \le y \\
&Dxy(\mathsf{S}z) := \mathsf{S}(D(x - \mathsf{S}y)yz) && \text{if } x > y
\end{aligned}$$

$$\begin{aligned}
&M : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\quad Mxy0 := 0 \\
&Mxy(\mathsf{S}z) := x && \text{if } x \le y \\
&Mxy(\mathsf{S}z) := M(x - \mathsf{S}y)yz && \text{if } x > y
\end{aligned}$$

a) Give computational definitions for D and M.

b) Prove $x < z \to x = Dxyz \cdot \mathsf{S}y + Mxyz$.

c) Prove $x = Dxy(\mathsf{S}x) \cdot \mathsf{S}y + Mxy(\mathsf{S}x)$.

14.7 Bounded Mu Operator

A *bounded μ-operator* is a function $(\mathsf{N} \to \mathsf{B}) \to \mathsf{N} \to \mathsf{N}$ that for a test $f^{\mathsf{N}\to\mathsf{B}}$ and an upper bound n^{N} yields the first number $k \le n$ satisfying f. If no such number exists, the operator returns the upper bound n.

Defining a bounded μ-operator with structural recursion is interesting. Since structural recursion provides descending recursion, performing a linear search $k = 0, 1, 2, \dots$ until either $fk = \mathsf{T}$ or $k = n$ as in the informal specification is not directly possible. However, with structural recursion on the upper bound n, we can do the following: If $n = 0$, we return 0; if $n = \mathsf{S}n'$, we check $f(\mu f n')$. If we have T, we return $\mu f n'$, otherwise we return $\mathsf{S}n'$. Following this outline, we define a **bounded μ-operator** as follows:

$$\begin{aligned}\mu :\ & (\mathsf{N} \to \mathsf{B}) \to \mathsf{N} \to \mathsf{N}\\ \mu f\, 0 \ &:=\ 0\\ \mu f\, (\mathsf{S}n) \ &:=\ \text{LET } k = \mu f n \text{ IN IF } fk \text{ THEN } k \text{ ELSE } \mathsf{S}n\end{aligned}$$

We now have an operational specification for a bounded μ-operator. If we want to prove properties about μ, it is useful to have a more explicit declarative specification for μ.

Fact 14.7.1
$$\begin{aligned}&\text{IF } f(\mu f n) \text{ THEN } \mu f n \le n \land \forall k < \mu f n.\ fk = \mathsf{F}\\ &\qquad\qquad \text{ELSE } \mu f n = n \ \land\ \forall k \le n.\ fk = \mathsf{F}\end{aligned}$$

Proof By induction on n. For the base case we, do a case analysis on $f0$. For the successor case, we do a case analysis on $f(\mu f n)$, followed by a case analysis on $f(\mathsf{S}n)$ in the negative case. In the negative negative case, the proof closes with a case analysis $k = \mathsf{S}n \lor k \le n$. ■

Fact 14.7.1 and its proof deserve discussion. Already the formulation of the fact with a boolean conditional is uncommon from a traditional mathematical perspective. The formulation with the conditional is convenient in our computational setting since once we do case analysis on $f(\mu f n)$ only the relevant branch remains. Also note that the given proof outline just gives the strategy, the actual verification is left to the reader. The actual verification can be done with Coq or on paper, where doing the verification on paper requires some writing. Verifying and understanding the proof with Coq takes less time.

Figure 14.2 shows 3 possible specifications of a bounded μ-operator.

Fact 14.7.2 The three specifications of bounded μ-operators in Figure 14.2 are equivalent and unique up to functional extensionality.

Recursive Specification

$$\begin{aligned}&\mu f 0 = 0\\&f(\mu f n) = \mathsf{T} \to \mu f(\mathsf{S}n) = \mu f n\\&f(\mu f n) = \mathsf{F} \to \mu f(\mathsf{S}n) = \mathsf{S}n\end{aligned}$$

Explicit Specification

$$\begin{aligned}&f(\mu f n) = \mathsf{T} \to \mu f n \le n \wedge \forall k.\, k < \mu f n \to fk = \mathsf{F}\\&f(\mu f n) = \mathsf{F} \to \mu f n = n \wedge \forall k.\, k \le n \to fk = \mathsf{F}\end{aligned}$$

Liberal Specification

$$\begin{aligned}&\mu f n \le n\\&k < \mu f n \to fk = \mathsf{F}\\&\mu f n < n \to f(\mu f n) = \mathsf{T}\end{aligned}$$

Figure 14.2: Three equivalent specifications of bounded μ-operators

Proof The uniqueness of the recursive specification follows easily by induction on n. Note that all three specifications are unique if they are equivalent.

That the recursive specification entails the explicit specification follows with the proof of Fact 14.7.1. Proving that the explicit specification entails the three conditions of the liberal specification is straightforward.

It remains to show that the liberal specification entails the recursive specification. Proving the first condition of the recursive specification is straightforward. The second condition follows by contradiction. We assume $f(\mu f n) = \mathsf{T}$ and show that both $\mu f(\mathsf{S}n) < \mu f n$ and $\mu f n < \mu f(\mathsf{S}n)$ are contradictory, as is justified by Corollary 14.3.3.

To prove the third condition of the recursive specification, we first note that the liberal specification gives us

$$H : \forall n.\, \neg \mu f n < n \to \mu f n = n$$

by antisymmetry, the first liberal condition, and contraposition. We now assume $H_1 : f(\mu f n) = \mathsf{F}$. By H we asume $H_2 : \mu f(\mathsf{S}n) < \mathsf{S}n$ and derive a contradiction. By the third liberal condition we have $H_3 : f(\mu f(\mathsf{S}n)) = \mathsf{T}$. By H, the third liberal condition, and H_1 we have $H_4 : \mu f n = n$. Now we close the proof by case analysis on $\mu f(\mathsf{S}n) = n \vee \mu f(\mathsf{S}n) < n$. In the first case the contradiction follows with H_1,

H_3, and H_4. In the second case the contradiction follows with H_4, the second liberal precondition, and H_3. ■

The final part of the proof is unpleasantly tricky. Doing it with a proof assistant saves time and increases confidence. The trickiness seems to come from reasoning by contradiction (justified by Corollaries 14.3.2 and 14.3.3).

Exercise 14.7.3 (Least Solutions) We define least solutions of tests as follows:

$$\mathsf{least}\, f\, n \;:=\; fn = \mathsf{T} \wedge \forall k.\; k < n \to fk = \mathsf{F}$$

a) Prove that a test has at most one least solution.

b) Prove that every satisfiable test has a least solution.

c) Define a function $\forall f.\, \mathsf{tsat}\, f \to \mathsf{sig}\,(\mathsf{least}\, f)$ that yields the least solution of a satisfiable test.

Exercise 14.7.4 Show that the specifications in Figure 14.2 are equivalent and unique up to functional extensionality.

Exercise 14.7.5 (Optimized Bounded Mu Operator) Our recursive definition of the boundedμ-operator is such that $\mu f n$ always tests f exactly n-times. It is possible to write a bounded μ-operator $\mu' f n k$ with an extra argument k such that $\mu' f n 0 = \mu f n$ and $\mu' f n 0$ tests f only as often as necessary. The trick is to increment k with each recursion step. Define and verify such an optimized bounded μ-operator.

14.8 Recursive Specifications of Functions

The definition of the bounded μ-operator was the first time we defined a function with a boolean conditional. We may write the structurally recursive specification of the bounded μ-operator as follows:

$$\begin{aligned} \mu f 0 &= 0 & \\ \mu f(\mathsf{S}n) &= \mu f n & \text{if } f(\mu f n) = \mathsf{T} \\ &= \mathsf{S}n & \text{if } f(\mu f n) = \mathsf{F} \end{aligned}$$

From this specification a computational definition of μ can be derived automatically.

The existence proof for the division theorem implicitly constructs functions D and M using the complete induction operator. One can show that the functions satisfy the following recursive specifications:

$$Mxy = \begin{cases} x & \text{if } x \le y \\ M(x - \mathsf{S}y)y & \text{if } x > y \end{cases} \qquad Dxy = \begin{cases} 0 & \text{if } x \le y \\ \mathsf{S}(D(x - \mathsf{S}y)y) & \text{if } x > y \end{cases}$$

The specifications are recursive but not structurally recursive. Coq's equations package can automatically derive a definition of the functions D and M from the above specifications.

Exercise 14.8.1 We consider two specifications of two functions $D, M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$. The *liberal specification* is

$$\begin{aligned}&Mxy \le y\\ &x = Dxy \cdot \mathsf{S}y + Mxy\end{aligned}$$

The *recursive specification* is

$$Mxy = \begin{cases} x & \text{if } x \le y \\ M(x - \mathsf{S}y)y & \text{if } x > y \end{cases} \qquad\qquad Dxy = \begin{cases} 0 & \text{if } x \le y \\ \mathsf{S}(D(x - \mathsf{S}y)y) & \text{if } x > y \end{cases}$$

Show that the two specifications are equivalent and unique up to functional extensionality. Use results we have shown before.

15 Indexed Inductive Predicates

Inductive predicates are defined through systems of proof constructors. We have seen basic examples in Chapter 3 on propositions as types and an advanced example with guarded recursion in Chapter 12 on witness operators. We now explore a degree of freedom in choosing the proof constructors for an inductive predicate we have not seen before. This degree of freedom makes it possible to instantiate arguments of the inductive predicate in the target type of proof constructors. If this feature is used, we speak of index arguments and of indexed inductive predicates. Indexed inductive predicates furnish Coq's type theory with expressivity essential for some important applications.

We study a series of example predicates developing the accompanying elimination techniques. This way we get familiar with the parameter-index distinction and a new type-checking device for defining equations instantiating index variables. A prominent example is the inductive definition of propositional equality that is adopted by Coq.

We assume familiarity with the elimination techniques for inductive data types introduced in Chapter 6. Familiarity with the recursive transfer predicate from Chapter 12 will also be helpful but is not assumed.

Working with indexed inductive predicates requires a couple of new type-theoretic techniques, so working yourself through enough exercises is essential.

Sections 15.9 and 15.10 address advanced topics and may be skipped on first reading.

15.1 Zero

Our first indexed inductive predicate is

$$\begin{aligned}\mathsf{zero} &: \mathsf{N} \to \mathbb{P}\\ \mathsf{Z} &: \ \mathsf{zero}\ 0\end{aligned}$$

The single proof constructor Z provides a canonical proof for zero 0. Note that the constructor Z **instantiates** the argument of the target predicate zero with 0. We speak of an **instantiating proof constructor**.

Arguments of an inductive predicate that are instantiated by a proof constructor are called **indices** to distinguish them from arguments that are not instantiated.

Arguments of an inductive predicate that are not instantiated by a proof constructor are called **parameters**. So far we have only seen inductive predicates where all arguments are parameters. Inductive predicates with index arguments are called **indexed inductive predicates**. Note that every argument of an inductive predicate is either a **parameter** or and **index**, but not both.

Since zero has only a single proof constructor for 0, we should expect that we can prove

$$\mathsf{zero}\ x \to x = 0$$

This proof can indeed be obtained with the eliminator[1]

$$\begin{aligned} &\mathsf{E_{zero}} : \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to \forall x.\, \mathsf{zero}\ x \to px \\ &\mathsf{E_{zero}}\ pa\ \_\ \mathsf{Z} \ := \ a \qquad : p0 \end{aligned}$$

If we look at the defining equation of the eliminator, we see that a new type checking device is being used (we speak of **indexed typing**). In the left-hand side of the defining equation the argument x for the index of the inductive predicate appears as an underline. This indicates that the argument is determined by the index argument of the target type of the proof constructor Z following as next argument. We thus have the typings

$$\mathsf{E_{zero}}\ pa\ \_\ \mathsf{Z} : \ p\,0 \quad \text{and} \quad \mathsf{E_{zero}}\ pa\,0\,\mathsf{Z} : \ p\,0$$

validating the right hand side of the defining equation.

We call the variable x in the type of $\mathsf{E_{zero}}$ an **index variable** since it is determined as an index of an inductive predicate.

Following our convention for eliminators, we refer to $\mathsf{E_{zero}}$ as **elimination lemma** when we use it as a declared constant. Nowhere in this chapter will we exploit that $\mathsf{E_{zero}}$ is a defined function.

Exercise 15.1.1 Prove the following facts.

a) $\mathsf{zero}\ x \ \leftrightarrow \ x = 0$

b) $\neg\mathsf{zero}(\mathsf{S}x)$

c) $\neg\mathsf{zero}\,1$

d) $\mathcal{D}(\mathsf{zero}\,x)$

Exercise 15.1.2 Prove the following impredicative characterization for zero:

$$\mathsf{zero}\ x \ \leftrightarrow \ \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to px$$

Exercise 15.1.3 Convince yourself that $\mathsf{E_{zero}}(\lambda x.\ \text{IF}\ x\ \text{THEN}\ \top\ \text{ELSE}\ \bot)\,\mathsf{I}\,5$ is a proof of $\neg\mathsf{zero}\,5$.

[1] Note that zero is not affected by the elim restriction.

15.2 Inductive Propositional Equality

Recall the discussion of propositional equality in Chapter 5. There we justified the constants eq, Q, and R with the Leibniz scheme. We will now see an inductive definition of the constants, which in fact is the definition used for propositional equality in Coq.

Coq defines propositional equality as an indexed inductive predicate:

$$\begin{aligned} \mathsf{eq} &: \forall X^{\mathbb{T}}.\, X \to X \to \mathbb{P} \\ \mathsf{Q} &: \forall X^{\mathbb{T}}\, x^X.\, \mathsf{eq} X x x \end{aligned}$$

The inductive definition introduces the constants eq and Q as constructors. Both X and x are accommodated as parameters. Following the convention that parameters precede indices, we accommodate the third argument of eq as an index. Exploiting the index argument of eq, we define an eliminator for eq:[2]

$$\begin{aligned} &\mathsf{E_{eq}} : \forall X^{\mathbb{T}}\, x^X\, p^{X \to \mathbb{T}}.\; px \to \forall y.\; \mathsf{eq} X x y \to py \\ &\mathsf{E_{eq}}\, X x p a\, \_\, (\mathsf{Q}\_\,\_) := a \qquad : px \end{aligned}$$

The **flow of information during type checking** the left-hand side of the defining equation of $\mathsf{E_{eq}}$ is as follows: First the arguments of Q are determined as the parameters X and x, then the index variable y is determined as the index of the proposition $\mathsf{eq}\, X x x$ of $\mathsf{Q}\, X x$, which is x.

Using the eliminator $\mathsf{E_{eq}}$, we can now define the rewriting law:

$$\begin{aligned} \mathsf{R} &: \forall X^{\mathbb{T}}\, x^X\, y^X\, p^{X \to \mathbb{P}}.\; \mathsf{eq}\, X x y \to px \to py \\ &:= \lambda X x y p h a.\; \mathsf{E_{eq}}\, X x p a y h \end{aligned}$$

Exercise 15.2.1 Prove $\forall X^{\mathbb{T}}\, x^X\, y^X.\; \mathsf{eq} X x y \;\leftrightarrow\; \forall p^{X \to \mathbb{P}}.\, px \to py$.

15.3 Even

The even numbers can be obtained by starting at 0 and by adding 2 as often as one likes:

$$0,\ 2,\ 4,\ 6,\ \ldots$$

The idea can be captured with an inductive predicate

$$\begin{aligned} \mathsf{even} &: \mathsf{N} \to \mathbb{P} \\ \mathsf{even_B} &: \mathsf{even}\ 0 \\ \mathsf{even_S} &: \forall n.\; \mathsf{even}\ n \to \mathsf{even}(\mathsf{S}(\mathsf{S}n)) \end{aligned}$$

[2] Note that eq is not affected by the elim restriction.

with two proof constructors giving us proof terms for exactly the even numbers:

$\mathsf{even}\ 0$	$\mathsf{even_B}$
$\mathsf{even}\ 2$	$\mathsf{even_S}\ 2\ \mathsf{even_B}$
$\mathsf{even}\ 4$	$\mathsf{even_S}\ 4\ (\mathsf{even_S}\ 2\ \mathsf{even_B})$
...	...

More generally, we can prove that every multiple of 2 is an even number:

$$\mathsf{even}(k \cdot 2) \tag{15.1}$$

The proof is by induction on k. The base case is $\mathsf{even}\ 0$. In the successor case we have $\mathsf{even}(k \cdot 2)$ as inductive hypothesis and need to show $\mathsf{even}(\mathsf{S}k \cdot 2)$. Since $\mathsf{S}k \cdot 2 \approx \mathsf{S}(\mathsf{S}(k \cdot 2))$, the claim follows with the proof constructor $\mathsf{even_S}$.

The proof constructors for even may be depicted as the proof rules

$$\frac{}{\mathsf{even}\ 0} \qquad\qquad \frac{\mathsf{even}\ n}{\mathsf{even}\ (\mathsf{S}(\mathsf{S}n))}$$

where the premises appear above the rule and the conclusion appears below the rule.

To prove more results about even, we need an eliminator. Here is an eliminator that suffices for our purposes:

$$\begin{aligned}
\mathsf{E_{even}} :\ & \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\ \mathsf{even}\ n \to pn \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.\ \mathsf{even}\ n \to pn \\
\mathsf{E_{even}}\ p\,a\,f\ \_\ \mathsf{even_B} &:= a && : p0 \\
\mathsf{E_{even}}\ paf\ \_\ (\mathsf{even_S}\ n'h) &:= fn'h\,(\mathsf{E_{even}}\ pafn'h) && : p(\mathsf{S}(\mathsf{S}n'))
\end{aligned}$$

The eliminator is defined by recursive case analysis on the inductive argument, which has type $\mathsf{even}\ n$. Note that n acts as an index variable in the target type of $\mathsf{E_{even}}$. The right hand sides of the defining equations receive the types given in the right column. The types are obtained by instantiating the index variable n as required by the proof constructors.

Note that the type of $\mathsf{E_{even}}$ has a **clause** for each of the two constructors of even. There is also a defining equation for each of the two constructors. The defining equation for the recursive proof constructor $\mathsf{even_S}$ is recursive so that it can provide the **inductive hypothesis** pn in the clause for $\mathsf{even_S}$.

When we translate the equational definition of $\mathsf{E_{even}}$ into a computational definition

$$\begin{aligned}
\mathsf{E_{even}} :\ & \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\ \mathsf{even}\ n \to pn \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.\ \mathsf{even}\ n \to pn \\
:=\ & \lambda paf.\ \textsc{fix}\ Fnh.\ \textsc{match}\ h\ [\,\mathsf{even_B} \Rightarrow a \mid \mathsf{even_S}\,n'h' \Rightarrow fn'h'(Fn'h')\,]
\end{aligned}$$

we see that the recursive abstraction must take two arguments so that F receives the dependent function type $\forall n.\ \mathsf{even}\ n \to pn$ necessary to type the recursive application $Fn'h'$. Note that the recursion is on the derivation $h:\mathsf{even}\ n$ and not on the number n.

When we use the eliminator $\mathsf{E}_{\mathsf{even}}$ as a declared constant, we refer to it as **induction lemma**.

Exercise 15.3.1 (Impredicative Characterization) Prove the equivalence

$$\mathsf{even}\ x \ \leftrightarrow\ \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall x.\ px \to p(\mathsf{S}(\mathsf{S}x))) \to px$$

establishing an impredicative characterization of even. Note that there is a clause for each of the two constructors mimicking the type of the constructor.

Exercise 15.3.2 Define a recursive eliminator

$$\tilde{\mathsf{E}}_{\mathsf{even}} :\ \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\ pn \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.\ \mathsf{even}\ n \to pn$$

omitting the assumption $\mathsf{even}\,n$ in the clause for the constructor even_S. The eliminator $\tilde{\mathsf{E}}_{\mathsf{even}}$ suffices for all proofs for even we do in this chapter. Show that the induction lemma $\mathsf{E}_{\mathsf{even}}$ can be obtained from a declared eliminator $\tilde{\mathsf{E}}_{\mathsf{even}}$. We remark that Coq automatically generates the eliminator $\mathsf{E}_{\mathsf{even}}$ shown before.

Exercise 15.3.3 Define a nonrecursive eliminator

$$\mathsf{M}_{\mathsf{even}} :\ \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\ \mathsf{even}\ n \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.\ \mathsf{even}\ n \to pn$$

omitting the inductive hypothesis. Show that an elimination lemma $\mathsf{M}_{\mathsf{even}}$ can be obtained from a declared eliminator $\mathsf{E}_{\mathsf{even}}$.

Exercise 15.3.4 Consider the inductive predicate

$$\begin{aligned}\mathsf{T} &:\ \mathsf{N} \to \mathbb{P}\\ \mathsf{T}_{\mathsf{B}_0} &:\ \mathsf{T}\,0\\ \mathsf{T}_{\mathsf{B}_1} &:\ \mathsf{T}\,1\\ \mathsf{T}_\mathsf{S} &:\ \forall n.\,\mathsf{T}\,n \to \mathsf{T}(\mathsf{S}n) \to \mathsf{T}(\mathsf{S}(\mathsf{S}n))\end{aligned}$$

a) Show that T holds for all numbers. Hint: Generalize the claim so you get a strong enough inductive hypothesis.

b) Derive the induction lemma for T. Notice that the clause for T_S has two inductive hypotheses.

		$\forall n.\ \mathsf{even}\ n \to \exists k.\ n = k \cdot 2$	apply $\mathsf{E}_{\mathsf{even}}$, intros
1		$\exists k.\ 0 = k \cdot 2$	$k \mapsto 0$
		$0 = 0 \cdot 2$	comp. equality
2	IH: $n = k \cdot 2$	$\exists k.\ \mathsf{S}(\mathsf{S}n) = k \cdot 2$	$k \mapsto \mathsf{S}k$
		$\mathsf{S}(\mathsf{S}n) = \mathsf{S}k \cdot 2$	rewrite IH
		$\mathsf{S}(\mathsf{S}(k \cdot 2)) = \mathsf{S}k \cdot 2$	comp. equality

Figure 15.1: Proof diagram for an inductive proof with $\mathsf{E}_{\mathsf{even}}$

15.4 Induction on Derivations

The recursive eliminator $\mathsf{E}_{\mathsf{even}}$ provides for inductive proofs known as **inductions on derivations** in the literature. **Derivations** can be understood as canonical proof terms for inductive propositions (e.g. $\mathsf{even}\ 36$).

We explain the idea with a proof of the proposition

$$\forall n.\ \mathsf{even}\ n \to \exists k.\ n = k \cdot 2 \tag{15.2}$$

The formal proof appears as a proof diagram in Figure 15.1. Informally, we say that we prove (15.2) by induction on the derivation of $\mathsf{even}\ n$. If $\mathsf{even}\ n$ is obtained with $\mathsf{even}_{\mathsf{B}}$, we have $n \mapsto 0$ and must show $\exists k.\ 0 = k \cdot 2$, which follows with $k \mapsto 0$ and computational equality. If $\mathsf{even}\ n$ is obtained with $\mathsf{even}_{\mathsf{S}}$, we have $n \mapsto \mathsf{S}(\mathsf{S}n)$ and must show $\exists k.\ \mathsf{S}(\mathsf{S}n) = k \cdot 2$. We also have the **inductive hypothesis** $\exists k.\ n = k \cdot 2$. The inductive hypothesis gives us some k such that $H : n = k \cdot 2$. To close the proof, it suffices to show $\mathsf{S}(\mathsf{S}n) = \mathsf{S}k \cdot 2$, which follows by rewriting with H and computational equality.

From our perspective, the formal proof laid out as a proof diagram in Figure 15.1 seems clearer than the informal proof talking about derivations. Historically, however, logicians did prove interesting facts about interesting inductive predicates (called proof systems) using the induction on derivations model before the advent of modern type theory.

We remark that Coq automatically derives the eliminator $\mathsf{E}_{\mathsf{even}}$ when the inductive predicate even is defined. Once an induction on derivations for even is initiated with the induction tactic, the eliminator is applied at the proof term level.

Soundness and Completeness

Fact 15.4.1 $\mathsf{even}\ n \leftrightarrow \exists k.\ n = k \cdot 2$.

Proof Follows with (15.2) and (15.1). ■

We may see the equivalence as a **specification** for the inductive predicate even. We call the two directions of the equivalence **soundness** ($\rightarrow$) and **completeness** ($\leftarrow$). Soundness says that everything we can prove to be even with the proof constructors of even is in fact an even number, and completeness says that every even number can in fact be derived with the proof constructors of even.

Informally, soundness results from the fact that the proof constructors for even are sound, while completeness results from the fact that for every even number a derivation can be constructed using the proof constructors for even.

Exercise 15.4.2 Define an inductive predicate odd and show that is satisfies the specification $\mathsf{odd}\ x \leftrightarrow \exists k.\ x = \mathsf{S}(k \cdot 2)$.

Exercise 15.4.3 Give specifications for the inductive predicates zero and eq and prove their corrrectness.

Exercise 15.4.4 Define an inductive proposition $\mathsf{F} : \mathbb{P}$ with a single recursive proof constructor $\mathsf{L} : \mathsf{F} \rightarrow \mathsf{F}$ and show $\mathsf{F} \leftrightarrow \bot$.

Exercise 15.4.5 Prove $\mathcal{D}(\mathsf{even}\ x)$ using the division theorem.

15.5 Inversion Lemmas and Unfolding

Proving negative facts about even such as

$$\neg\mathsf{even}\ 1 \tag{15.3}$$

$$\neg\mathsf{even}\ 3 \tag{15.4}$$

$$\neg\mathsf{even}\ n \rightarrow \neg\mathsf{even}\ (\mathsf{S}(\mathsf{S}n)) \tag{15.5}$$

$$\neg\mathsf{even}\ (\mathsf{S}(n \cdot 2)) \tag{15.6}$$

takes insight and a technique called unfolding. Clearly, (15.3) and (15.4) both follow from (15.6). Moreover, (15.6) follows by induction on n using (15.3) for the base case and (15.5) for the successor case. Finally, (15.5) follows from the positive fact

$$\mathsf{even}\ (\mathsf{S}(\mathsf{S}n)) \rightarrow \mathsf{even}\ n \tag{15.7}$$

We are thus left with (15.3) and (15.7), which we will refer to as **inversion lemmas**. Note that (15.7) is in fact the converse of the proof constructor even_S.

For (15.3) it is best to prove the generalized fact

$$\mathsf{even}\ k \rightarrow k = 1 \rightarrow \bot \tag{15.8}$$

which follows with the elimination lemma using $0 \neq 1$ and $\mathsf{S}(\mathsf{S}k) \neq 1$.

For (15.7) it is again best to prove the generalized fact

$$\mathsf{even}\ k \to k = \mathsf{S}(\mathsf{S}n) \to \mathsf{even}\ n \tag{15.9}$$

which follows with the elimination lemma using $0 \neq \mathsf{S}(\mathsf{S}n)$ and

$$\mathsf{even}\ k \to \mathsf{S}(\mathsf{S}k) = \mathsf{S}(\mathsf{S}n) \to \mathsf{even}\ n$$

where the latter follows by injectivity of S.

Note that the generalisations (15.8) and (15.9) used for proving the inversion lemmas (15.3) and (15.7) are obtained with an **unfolding scheme** from the inversion lemmas: The non-variable term in the index position of the inductive predicate even is **unfolded** using a fresh variablen k. The unfolding scheme is generally useful when working with indexed inductive predicates. For instance, to prove $\neg\mathsf{zero}\,1$, one may unfold 1 from the index position and prove $\mathsf{zero}\,x \to x = 1 \to \bot$ using the eliminator for zero.

We remark that Coq's tactic depelim proves both inversion lemmas in one step.[3]

15.6 Proceed with Care

We now prove some further properties of evenness based on the inductive definition. Mathematically, working with the multiplicative definition $\lambda n.\exists k.\ n = 2 \cdot k$ may be more appropriate. Our motivation for working with the inductive definition of evenness is curiosity and the demonstration of proof techniques for indexed inductive predicates.

Fact 15.6.1 The successors of even numbers are not even.
That is, $\mathsf{even}\,n \to \mathsf{even}(\mathsf{S}n) \to \bot$.

Proof By induction of the derivation of $\mathsf{even}\,n$ using the inversion lemmas (15.3) and (15.7). ■

Next we aim at a native proof of $\mathcal{D}(\mathsf{even}\,n)$. We proceed by induction on n. In the successor case we need $\neg\mathsf{even}\,n \to \mathsf{even}(\mathsf{S}n)$, which we have not shown so far. Showing this fact needs a new idea. The claim follows with induction on n provided we show $\neg\mathsf{even}(\mathsf{S}n) \to \mathsf{even}\,n$ in parallel.

Fact 15.6.2 $(\neg\,\mathsf{even}\,n \to \mathsf{even}(\mathsf{S}n)) \wedge (\neg\,\mathsf{even}(\mathsf{S}n) \to \mathsf{even}\,n)$.

Proof By induction on n using (15.3) and (15.7). ■

[3] The tactic depelim comes with the Equations package supporting definition of functions with equations and well-founded recursion.

Fact 15.6.3 $\mathcal{D}(\mathsf{even}\,n)$.

Proof By induction on n using Facts 15.6.1, 15.6.2, and (15.7). ■

Exercise 15.6.4 Prove the following facts about even.

a) $\mathsf{even}\,x \to \mathsf{even}\,y \to \mathsf{even}(x+y)$

b) $\mathsf{even}\,x \to \mathsf{even}(x+y) \to \mathsf{even}\,y$

15.7 Linear Order on Numbers

Coq defines the linear order on numbers inductively:

$$\begin{aligned}
\mathsf{le} &: \mathsf{N} \to \mathsf{N} \to \mathbb{P} \\
\mathsf{le_B} &: \forall x.\ \mathsf{le}\,x\,x \\
\mathsf{le_S} &: \forall x y.\ \mathsf{le}\,x\,y \to \mathsf{le}\,x\,(\mathsf{S}y)
\end{aligned}$$

Note that the first argument of le is a parameter and the second argument of le is an index. The proof constructors for le may be depicted with the proof rules

$$\frac{}{\mathsf{le}\,x\,x} \qquad\qquad \frac{\mathsf{le}\,x\,y}{\mathsf{le}\,x\,(\mathsf{S}y)}$$

Note that the proof rules express basics facts about the linear order on numbers. Thus every proposition $\mathsf{le}\,x\,y$ that can be derived with the rules entails $x \le y$ (soundness). We can also prove that a proposition $\mathsf{le}\,y\,y$ can be derived with the rules whenever $x \le y$ (completeness).

Fact 15.7.1 $\mathsf{le}\,x\,y \leftrightarrow \exists k.\ k + x = y$.

Proof The direction $\to$ is by induction on the derivation of $\mathsf{le}\,x\,y$. In the base case we have $y \mapsto x$. In the successor case we have the inductive hypothesis $\exists k.\ k+x = y$ and need to show $\exists k.\ k + x = \mathsf{S}y$, which is straightforward.

For the direction $\leftarrow$ we show $\mathsf{le}\,x\,(k + x)$ by induction on k. Straightforward. ■

We have shown $\mathsf{le}\,x\,y \leftrightarrow \exists k.\ k + x = y$ rather than $\mathsf{le}\,x\,y \leftrightarrow \exists k.\ x + k = y$ so that we don't need the commutativity of $+$.

Exercise 15.7.2 Define an eliminator for le that suffices for the induction used for the direction $\to$ of Fact 15.7.1.

Exercise 15.7.3 Prove the following inversion lemma:
$\forall x y.\ \mathsf{le}\,x\,y \to x = y \lor \exists y'.\ y = \mathsf{S}y' \land \mathsf{le}\,x\,y'$.

Exercise 15.7.4 Here is another inductive definition of the linear order on numbers:

$$\begin{aligned} \mathsf{le}' &: \mathsf{N} \to \mathsf{N} \to \mathbb{P} \\ \mathsf{le}'_{\mathsf{B}} &: \forall x.\ \mathsf{le}'\,0x \\ \mathsf{le}'_{\mathsf{S}} &: \forall xy.\ \mathsf{le}'\,xy \to \mathsf{le}'\,(\mathsf{S}x)(\mathsf{S}y) \end{aligned}$$

Note that both arguments of le' are indices.

a) Define an eliminator for le'.

b) Prove $\mathsf{le}'\,xy \leftrightarrow \exists k.\ x + k = y$.

c) Prove $\mathsf{le}'\,xy \leftrightarrow \mathsf{le}\,xy$.

Exercise 15.7.5 It is possible to show the basic facts about linear order starting from the inductive definition of le and not using the translation to addition provided by Fact 15.7.1. Some of the direct proofs are tricky (i.e., strictness $x < x$) but nevertheless provide interesting exercises for working with indexed inductive types. Try the following:

1. $0 \le x$
2. $x \le 0 \to x = 0$
3. $\neg(x < 0)$
4. $x \le y \to \mathsf{S}x \le \mathsf{S}y$ (shift)
5. $x \le y \to y \le z \to x \le z$ (transitivity)
6. $x < y \to y \le z \to x < z$ (strict transitivity)
7. $x \le y \to y < z \to x < z$ (strict transitivity)
8. $x < y \to x \le y$
9. $\mathsf{S}x \le \mathsf{S}y \to x \le y$
10. $x < y \to x \ne y$
11. $\neg(x < x)$ (strictness)
12. $x \le y \to y \le x \to x = y$ (antisymmetry)
13. $x < y \lor x = y \lor y < x$
14. $\mathcal{D}(x \le y)$ (decidability)

Hints: Claim 1 follows by induction on x. Claim 2 follows by inversion on $x \le 0$. Claim 3 follows from (2). Claim 4 follows by induction on $x \le y$. Claim 5 follows by induction on $y \le z$. Claim 6 follows from (5). Claim 7 follows from (5) and (4). Claim 8 follows from (5). Claim 9 follows by inversion of $\mathsf{S}x \le \mathsf{S}y$ and (8). Claim 10 is tricky; follows by induction on y with x quantified using (3) and (9). Claim 11 follows from (10). Claim 12 follows by inversion of $x \le y$ using (11) and (7). Claims 13 and 14 follow by induction on x with y quantified, case analysis of y, and (1), (3), and (4).

15.8 More Inductive Predicates

Given an informal or formal specification of a predicate, one often can come up with an elegant system of proof constructors that is sound and complete for the predicate and thus yields an inductive definition of the predicate. Coming up with nice systems of proof constructors is a creative process nourished by experience.

Exercise 15.8.1 Give inductive definitions for the predicates specified below. Do not use auxiliary functions like addition or multiplication and do not use auxiliary predicates. Except for one case indexed inductive predicates are needed. Prove that your inductive predicates satisfy their specifying equivalence. In each case try to define the accompanying eliminator.

a) $Dxy \leftrightarrow x = 2 \cdot y$

b) $Mxyz \leftrightarrow x = 3 \cdot y + z \wedge z \leq 2$

c) $Upn \leftrightarrow \exists k.\ k \geq n \wedge pk \qquad (p : \mathsf{N} \to \mathbb{P})$

d) $Lpn \leftrightarrow \exists k.\ k \leq n \wedge pk \qquad (p : \mathsf{N} \to \mathbb{P})$

15.9 Pureness of zero

We will now prove the proposition

$$\forall h^{\mathsf{zero}\,0}.\ h = \mathsf{Z} \tag{15.10}$$

Intuitively, this is an obvious fact. In Coq, there is indeed an automation tactic (depelim) that derives $h = \mathsf{Z}$ from $h : \mathsf{zero}\,0$ in one step. Constructing a formal proof of the fact does require new ideas, however. We need a stronger elimination lemma modeling the dependency on the proof h, and we need to apply the elimination lemma with a clever target predicate to avoid an unexpected typing conflict.

The full eliminator for zero is

$$\begin{aligned} &\hat{\mathsf{E}}_{\mathsf{zero}} : \ \forall p^{\forall x.\ \mathsf{zero}\,x \to \mathbb{T}}.\ p0\mathsf{Z} \to \forall xh.\, pxh \\ &\hat{\mathsf{E}}_{\mathsf{zero}}\ pa\,\_\,\mathsf{Z} \ := \ a \qquad : p0\mathsf{Z} \end{aligned}$$

Note that the defining equation for $\hat{\mathsf{E}}_{\mathsf{zero}}$ is the same as for $\mathsf{E}_{\mathsf{zero}}$, so the difference is just in the more general type of $\hat{\mathsf{E}}_{\mathsf{zero}}$. This time the target predicate p takes both the index x^{N} and the proof $h^{\mathsf{zero}\,x}$ as arguments.

We now prove (15.10) with the term

$$\hat{\mathsf{E}}_{\mathsf{zero}}\ p\ (\mathsf{Q}\ \mathsf{Z})\ 0 \ : \ \forall h^{\mathsf{zero}\,0}.\ h = \mathsf{Z}$$

where $p : \forall x^{\mathsf{N}}.\ \mathsf{zero}\,x \to \mathbb{P}$ is a predicate satisfying

$$p0h \approx (h = \mathsf{Z})$$

A first try defining p as $p := \lambda x h^{\mathsf{zero}\,x}.\ h = \mathsf{Z}$ fails since the equation $h = \mathsf{Z}$ doesn't type check. We fix the problem with a match on x:

$$\begin{aligned} p :\ & \forall x^{\mathsf{N}}.\ \mathsf{zero}\,x \to \mathbb{P} \\ p0h \ &:=\ (h = \mathsf{Z}) \\ p(\mathsf{S}\_)h \ &:=\ \top \end{aligned}$$

The proof we have given uses three essential features of Coq's type theory: Dependent function types ($\hat{\mathsf{E}}_{\mathsf{zero}}$ and p), the conversion rule, and indexed typing in the defining equation of $\hat{\mathsf{E}}_{\mathsf{zero}}$.

Exercise 15.9.1 Explain why $\forall x^{\mathsf{N}} h^{\mathsf{zero}\,x}.\ h = \mathsf{Z}$ does not type check.

Exercise 15.9.2 Define $\mathsf{E}_{\mathsf{zero}}$ with $\hat{\mathsf{E}}_{\mathsf{zero}}$.

Exercise 15.9.3 Write $\hat{\mathsf{E}}_{\mathsf{zero}}$ and p with matches. Check your translation with Coq and notice that Coq elaborates the matches with return type functions.

Exercise 15.9.4 Prove $\mathsf{pure}(\mathsf{zero}\,x)$.

15.10 Axiom K

Axiom K is the proposition

$$\mathsf{K} \ :=\ \forall X^{\mathbb{T}}\, x^X\, p^{\mathsf{eq}Xxx \to \mathbb{P}}.\ p(\mathsf{Q}Xx) \to \forall h.ph.$$

stating that $\mathsf{Q}Xx$ is the only the proof of $\mathsf{eq}\,Xxx$. It turns out that K is independent in Coq's type theory, which is surprising given a naive understanding of the inductive definition of eq. Note that Axiom K is only meaningful for an inductive definition of propositional equality.

It seems that a proof of K should be possible following the ideas of the proof of

$$\forall h^{\mathsf{zero}\,0}.\ h = \mathsf{Z}$$

in Section 15.9. Following the proof for zero, we may try to prove

$$\forall X^{\mathbb{T}}\, x^X\, h^{\mathsf{eq}Xxx}.\ h = \mathsf{Q}Xx$$

which is equivalent to K. Defining a full eliminator for eq is not difficult:

$$\begin{aligned} \hat{\mathsf{E}}_{\mathsf{eq}} :\ & \forall X^{\mathbb{T}}\, x^X\, p^{\forall y.\ \mathsf{eq}Xxy \to \mathbb{T}}.\ px(\mathsf{Q}Xx) \to \forall yh.\ pyh \\ \hat{\mathsf{E}}_{\mathsf{eq}}\, Xxpa\_(\mathsf{Q}\_\_) \ &:=\ a \qquad : px(\mathsf{Q}Xx) \end{aligned}$$

The crux now is that we cannot find a predicate p and a proof a such that

$$\hat{\mathsf{E}}_{\mathsf{eq}}\ Xxpax\ :\ \forall h^{\mathsf{eq}Xxx}.\ h = \mathsf{Q}Xx$$

type checks. The difference with zero is that 0 is a constructor that can be matched on while x is abstract and cannot be matched on.

Exercise 15.10.1 Prove that K is equivalent to $\forall X^{\mathbb{T}}\, x^X\, h^{\mathsf{eq}Xxx}.\ h = \mathsf{Q}Xx$.

Exercise 15.10.2 Prove that PI implies K. See Section 9.3 for the definition of PI.

Exercise 15.10.3 Define E_{eq} with $\hat{\mathsf{E}}_{\mathsf{eq}}$.

Exercise 15.10.4 Define E_{eq} and $\hat{\mathsf{E}}_{\mathsf{eq}}$ with matches. Check your translations with Coq. Note that Coq elaborates the matches with appropriate return type functions.

15.11 Summary

We have defined inductive predicates satisfying the following specifications:

$$\begin{aligned}
\mathsf{zero}\,x &\leftrightarrow x = 0\\
\mathsf{eq}\,Xxy &\leftrightarrow \forall p^{X\to\mathbb{P}}.\ px \to py\\
\mathsf{even}\,n &\leftrightarrow \exists k.\ x = 2\cdot k\\
\mathsf{le}\,xy &\leftrightarrow \exists k.\ x + k = y
\end{aligned}$$

In each case we used proof constructors whose target type instantiates arguments of the inductive predicate. If such an instantiation takes place, we speak of index arguments and of indexed inductive predicates. In each case we proved the specifying equivalence. Proving the direction from right to left (known as completeness) was routine in each case. For the directions from left to right (known as soundness) the eliminators for the inductive predicates were needed. The types of the eliminators have a special form reflecting the parameter-index distinction.

When working with inductive predicates we want to rely on intuitions, given that the formal details are often involved. Working with inductive predicates in Coq profits much from automation, in particular, the automatic derivation of eliminators, the induction tactic, and the dependent elimination tactic depelim.

There are important applications of indexed inductive predicates, including proof systems, operational semantics, type systems, and logic programming. Assuming a reader not familiar with these applications, we have discussed the new technical issues with example predicates that easily could be defined otherwise. The exception is inductive equality, where the inductive definition adds important qualities we will explore in a later chapter.

16 Lists

We study the inductive type for lists providing a recursive representation for finite sequences over a base type. Besides numbers and pairs, lists are the most important data type in constructive type theory. Lists have much in common with numbers since for both data structures recursion and induction are linear. Lists also have much in common with finite sets since they have a notion of membership. In fact, our focus will be on the membership relation for lists.

We will see elegant indexed inductive predicates for membership and disjointness of lists, and also for repeating and non-repeating lists.

We will all see how option types can be used to obtain a total subscript function that yields for a list and a number the element at the corresponding position.

16.1 Inductive Definition

A list represents a finite sequence $[x_1 ; \dots ; x_n]$ of values. Formally, lists are obtained with two constructors **nil** and **cons**:

$$\begin{array}{rcl} [] & \mapsto & \mathsf{nil} \\ [x] & \mapsto & \mathsf{cons}\ x\ \mathsf{nil} \\ [x;y] & \mapsto & \mathsf{cons}\ x\ (\mathsf{cons}\ y\ \mathsf{nil}) \\ [x;y;z] & \mapsto & \mathsf{cons}\ x\ (\mathsf{cons}\ y\ (\mathsf{cons}\ z\ \mathsf{nil})) \end{array}$$

The constructor nil provides the **empty list**. The constructor cons yields for a value x and a list $[x_1 ; \dots ; x_n]$ the list $[x ; x_1 ; \dots ; x_n]$. Given a list cons x A, we call x the **head** and A the **tail** of the list. Given a list $[x_1 ; \dots ; x_n]$, we call n the **length** of the list and $x_1, \dots, x_n$ the **elements** of the list. An element may appear more than once in a list. For instance, $[2;2;3]$ is a list of length 3 that has 2 elements.

Formally, lists are accommodated with an inductive type definition introducing three constructors:

$$\begin{array}{rcl} \mathcal{L} & : & \mathbb{T} \to \mathbb{T} \\ \mathsf{nil} & : & \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \\ \mathsf{cons} & : & \forall X^{\mathbb{T}}.\ X \to \mathcal{L}(X) \to \mathcal{L}(X) \end{array}$$

Lists of type $\mathcal{L}(X)$ are called **lists over X**. The typing discipline enforces that all elements of a list have the same type. For nil and cons, we don't write the first argument X and use the following notations:

$$\begin{aligned} [\,] &:= \mathsf{nil} \\ x :: A &:= \mathsf{cons}\, x\, A \end{aligned}$$

For cons, we admit parentheses as follows:

$$x :: y :: A \quad \rightsquigarrow \quad x :: (y :: A)$$

The inductive definition of lists provides for case analysis, recursion, and induction on lists, in a way that is quite similar to what we have seen for numbers. As examples for recursive definitions we give a function

$$\begin{aligned} \mathsf{len} :\ & \forall X^{\mathbb{T}}.\, \mathcal{L}(X) \to \mathsf{N} \\ \mathsf{len}\, [\,] &:= 0 \\ \mathsf{len}\, (x :: A) &:= S\, (\mathsf{len}\, A) \end{aligned}$$

that yields the **length of a list** (informally, $\mathsf{len}\, [x_1; \dots; x_n] = n$), and a function

$$\begin{aligned} + :\ & \forall X.\, \mathcal{L}(X) \to \mathcal{L}(X) \to \mathcal{L}(X) \\ [\,] + B &:= B \\ (x :: A) + B &:= x :: (A + B) \end{aligned}$$

that **concatenates** two lists, which informally may be written as

$$[x_1; \dots; x_m] + [y_1; \dots; y_n] = [x_1; \dots; x_m; y_1; \dots; y_n]$$

We also define an **eliminator for lists**

$$\begin{aligned} \mathsf{E}_{\mathcal{L}} :\ & \forall X^{\mathbb{T}}\, p^{\mathcal{L}(X) \to \mathbb{T}}.\ p\, [\,] \to (\forall x A.\, pA \to p(x :: A)) \to \forall A.\, pA \\ \mathsf{E}_{\mathcal{L}} X p a f\, [\,] &:= a \\ \mathsf{E}_{\mathcal{L}} X p a f\, (x :: A) &:= f x A (\mathsf{E}_{\mathcal{L}} X p a f A) \end{aligned}$$

providing for inductive proofs and the construction of recursive functions.

Fact 16.1.1

1. Disjointness: $[\,] \neq x :: A$
2. Injectivity: $x :: A = y :: B \to x = y$
3. Injectivity: $x :: A = y :: B \to A = B$
4. Progress: $x :: A \neq A$

Proof The proofs are similar to the corresponding proofs for numbers. Claim (4) corresponds to $\mathsf{S}n \neq n$ and follows by induction on A with x quantified. ∎

Fact 16.1.2 If X is a discrete type, then $\mathcal{L}(X)$ is a discrete type.

Proof Let X be discrete and A, B be lists over X. We show $\mathcal{D}(A = B)$ by induction over A with B quantified followed by destructuring of B using disjointness and injectivity from Fact 16.1.1. In case both lists are nonempty with heads x and y, an additional case analysis on $x = y$ is needed. ∎

Fact 16.1.3 (Associativity) $A +\!\!\!+ (B +\!\!\!+ C) = (A +\!\!\!+ B) +\!\!\!+ C$.

Proof By induction on A. ∎

Fact 16.1.4 (Length) $\mathsf{len}\,(A +\!\!\!+ B) = \mathsf{len}\,A + \mathsf{len}\,B$ and $\mathsf{len}\,A = 0 \leftrightarrow A = []$

Proof By induction and case analysis on A. ∎

Exercise 16.1.5 Prove the above facts in detail (i.e., with Coq).

Exercise 16.1.6 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ \mathcal{D}(A = [])$.

Exercise 16.1.7 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ (A = []) + \Sigma x B.\ A = x :: B$.

16.2 Membership

Informally, we may characterize **membership** for lists with the equivalence

$$x \in [x_1\,;\,\dots\,;\,x_n] \;\leftrightarrow\; x = x_1 \lor \cdots \lor x = x_n \lor \bot$$

Formally, we can define a **membership predicate** either inductively with the rules

$$\frac{}{x \in x :: A} \qquad\qquad \frac{x \in A}{x \in y :: A}$$

or recursively with the equations

$$\begin{aligned}(x \in []) &= \bot \\ (x \in y :: A) &= (x = y \lor x \in A)\end{aligned}$$

Coq chooses the recursive definition and thus we do the same.[1] In full formal glory, the two definitions look as follows:

$$\begin{aligned}
\mathsf{mem} &: \forall X^{\mathbb{T}}.\ X \to \mathcal{L}(X) \to \mathbb{P}\\
\mathsf{mem_B} &: \forall XxA.\ \mathsf{mem}\,Xx(x :: A)\\
\mathsf{mem_C} &: \forall XxyA.\ \mathsf{mem}\,XxA \to \mathsf{mem}\,Xx(y :: A)
\end{aligned}$$

$$\begin{aligned}
\mathsf{member} &: \forall X^{\mathbb{T}}.\ X \to \mathcal{L}(X) \to \mathbb{P}\\
\mathsf{member}\,Xx\,[] &:= \bot\\
\mathsf{member}\,Xx(y :: A) &:= (x = y \lor \mathsf{member}\,XxA)
\end{aligned}$$

Note that member is an indexed inductive predicate where the first two arguments are parameters and the third argument is an index. We treat the type argument X of both predicates as an implicit argument and write $x \in A$ for $\mathsf{mem}\,xA$. We say that x is an **element** of a list A if $x \in A$.

Fact 16.2.1 $\mathsf{mem}\,xA \leftrightarrow \mathsf{member}\,xA$.

Proof Direction $\to$ follows by induction on the derivation of $\mathsf{mem}\,xA$. Direction $\leftarrow$ follows by induction on A. ■

Fact 16.2.2 Let A be a list over a discrete type X. Then $\mathcal{D}(x \in A)$.

Proof By induction on A. ■

Recall that bounded quantification over numbers preserves decidability (Fact 14.3.5). Similarly, quantification over the elements of a list preserves decidability.

Fact 16.2.3 (Bounded Quantification) Let $p : X \to \mathbb{P}$ and $A : \mathcal{L}(X)$. Then:

1. $(\forall x.\ \mathcal{D}(px)) \to \mathcal{D}(\forall x.\ x \in A \to px)$.
2. $(\forall x.\ \mathcal{D}(px)) \to \mathcal{D}(\exists x.\ x \in A \land px)$.

Proof By induction on A. ■

Fact 16.2.4 (Concatenation) $x \in A + B \leftrightarrow x \in A \lor x \in B$.

Proof By induction on A. ■

Membership can also be characterized with existential quantification and concatenation. We speak of the explicit characterization of list membership.

[1] That Coq defines membership recursively seems outdated given the current preference for inductively defined predicates.

Fact 16.2.5 (Explicit Characterization) $x \in A \leftrightarrow \exists A_1 A_2.\ A = A_1 + x :: A_2$.

Proof Direction $\rightarrow$ follows by induction on A. Direction $\leftarrow$ follows by induction on A_1. ■

Fact 16.2.6 (Factorization) For every discrete type X there is a function $\forall x^X A^{\mathcal{L}(X)}.\ x \in A \rightarrow \Sigma A_1 A_2.\ A = A_1 + x :: A_2$.

Proof By induction on A. The nil case is contradictory. In the cons case a case analysis on $\mathcal{D}(x = y)$ closes the proof. ■

Exercise 16.2.7 Define an eliminator for the inductive predicate member that suffices for the inductive proof of direction $\rightarrow$ of Fact 16.2.1.

Exercise 16.2.8 (Pigeonhole) Prove that a list of numbers whose sum is greater than the length of the list must contain a number that is at least 2:

$$\mathsf{sum}\,A > \mathsf{len}\,A \;\rightarrow\; \exists x.\ x \in A \wedge x \geq 2$$

First define the function sum.

16.3 Positions and Options

The positions of a list $[x_1; \dots; x_n]$ are the numbers $0, \dots, n-1$. More formally, a number n is a **position** of a list A if $n < \mathsf{len}\,A$. We now want to define a **subscript function** sub that given a list A and a number n yields the element at position n. For instance, for $[1;2;3]$ and 1 the function sub should yield 2. So far so good, but what type can we give sub? Our first attempt is

$$\forall X.\ \mathcal{L}(X) \rightarrow \mathsf{N} \rightarrow X$$

but this cannot work since it would imply that every type is inhabited (take the empty list and the number 0). Our second attempt is

$$\forall X^{\mathbb{T}} A^{\mathcal{L}(X)} n^{\mathsf{N}}.\ (n < \mathsf{len}\,A) \rightarrow X$$

and this does in fact work. However, we would like to define sub such that takes only A and n as argument. The only way to reach this goal is to give sub a return type $\mathcal{O}(X)$ that extends X with an extra element $\emptyset$ that is returned in case n is not a position of A:

$$\forall X.\ \mathcal{L}(X) \rightarrow \mathsf{N} \rightarrow \mathcal{O}(X)$$

One says that $\mathcal{O}(X)$ is the *option type* for X.

Formally, we define **option types** inductively:

$$\begin{aligned} \mathcal{O} &: \mathbb{T} \to \mathbb{T} \\ \emptyset &: \forall X^{\mathbb{T}}.\ \mathcal{O}(X) \\ {}^\circ &: \forall X^{\mathbb{T}}.\ X \to \mathcal{O}(X) \end{aligned}$$

We now define a subscript function by recursion on the given list and case analysis on the given number:

$$\begin{aligned} \mathsf{sub} &: \forall X.\ \mathcal{L}(X) \to \mathsf{N} \to \mathcal{O}(X) \\ \mathsf{sub}\, X\, []\, \_ &:= \emptyset \\ \mathsf{sub}\, X\, (x :: A)\, 0 &:= {}^\circ x \\ \mathsf{sub}\, X\, (x :: A)\, (\mathsf{S}n) &:= \mathsf{sub}\, A\, n \end{aligned}$$

Fact 16.3.1 $x \in A \to \exists n.\ \mathsf{sub}\, An = {}^\circ x$.

Proof By induction on A. The base case is contradictory. Case analysis on $x \in y :: A$ in the cons case. If $x = y$, then $n = 0$. Otherwise $\mathsf{sub}\, An = {}^\circ x$ by the inductive hypothesis. The claim follows with $n \mapsto \mathsf{S}n$. ■

Fact 16.3.2 $\mathsf{sub}\, An = {}^\circ x \ \to\ x \in A$.

Proof By induction on A. The base case is contradictory. Case analysis on n in the cons case. ■

Fact 16.3.3 $n < \mathsf{len}\, A \ \to \Sigma x.\ \mathsf{sub}\, An = {}^\circ x$.

Proof By induction on A with n quantified. The base case is contradictory. Case analysis on n in the cons case. ■

Exercise 16.3.4 Prove $a \neq \emptyset \Leftrightarrow \Sigma x.\ a = {}^\circ x$.

Exercise 16.3.5 Let A be a list over a discrete type. Prove $x \in A \to \Sigma n.\ \mathsf{sub}\, An = {}^\circ x$.

Exercise 16.3.6 Prove $(\forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathsf{N} \to X) \to \bot$.

Exercise 16.3.7 Let X be a discrete type. Define a function $\mathsf{pos} : \mathcal{L}(X) \to X \to \mathcal{O}(\mathsf{N})$ such that $\mathsf{pos}\, Ax \neq \emptyset \leftrightarrow x \in A$ and $\mathsf{pos}\, Ax = {}^\circ n \to \mathsf{sub}\, An = {}^\circ x$. Verify that your function satisfies the specification. Is the specification unique (up to functional extensionality)?

16.4 List Inclusion and List Equivalence

We may see lists as representations of finite sets. List membership then corresponds to set membership. The list representation of sets is not unique since the same set may have different list representations. For instance, $[1;2]$, $[2;1]$, and $[1;1;2]$ are different lists all representing the set $\{1,2\}$. In contrast to sets, lists are ordered structures providing for multiple occurrences of elements.

From the type-theoretic perspective, sets are informal objects that may or may not have representations in type theory. This is in sharp contrast to set-based mathematics where sets are taken as basic formal objects. The reason sets don't appear natively in Coq's type theory is that Coq's type theory is a computational theory while sets in general are noncomputational.

We will take lists over X as type-theoretic representations of finite sets over X. With this interpretation of lists in mind, we define **list inclusion** and **list equivalence** as follows:

$$\begin{aligned} A \subseteq B &:= \forall x.\, x \in A \to x \in B \\ A \equiv B &:= A \subseteq B \land B \subseteq A \end{aligned}$$

Note that two lists are equivalent if and only if they represent the same set.

Fact 16.4.1 List inclusion $A \subseteq B$ is reflexive and transitive. List equivalence $A \equiv B$ is reflexive, symmetric, and transitive.

Fact 16.4.2 We have the following properties for membership, inclusion, and equivalence of lists.

$$\begin{array}{ll}
x \notin [] & x \in [y] \leftrightarrow x = y \\
[] \subseteq A & A \subseteq [] \to A = [] \\
x \in y :: A \to x \neq y \to x \in A & x \notin y :: A \to x \neq y \land x \notin A \\
A \subseteq B \to x \in A \to x \in B & A \equiv B \to x \in A \leftrightarrow x \in B \\
A \subseteq B \to x :: A \subseteq x :: B & A \equiv B \to x :: A \equiv x :: B \\
A \subseteq B \to A \subseteq x :: B & x :: A \subseteq B \leftrightarrow x \in B \land A \subseteq B \\
x :: A \subseteq x :: B \to x \notin A \to A \subseteq B & x :: A \subseteq [y] \leftrightarrow x = y \land A \subseteq [y] \\
x :: A \equiv x :: x :: A & x :: y :: A \equiv y :: x :: A \\
x \in A \to A \equiv x :: A & \\
x \in A +\!\!+ B \;\leftrightarrow\; x \in A \lor x \in B & \\
A \subseteq A' \to B \subseteq B' \to A +\!\!+ B \subseteq A' +\!\!+ B' & A +\!\!+ B \subseteq C \;\leftrightarrow\; A \subseteq C \land B \subseteq C
\end{array}$$

Proof Except for the membership fact for concatenation, which appeared as Fact 16.2.4, all claims have straightforward proofs not using induction. ∎

Fact 16.4.3 Let A and B be lists over a discrete type. Then $\mathcal{D}(A \subseteq B)$ and $\mathcal{D}(A \equiv B)$.

Proof Holds since membership is decidable (Fact 16.2.2) and bounded quantification preserves decidability (Fact 16.2.3). ■

16.5 Setoid Rewriting

It is possible to rewrite a claim or an assumption in a proof goal with a propositional equivalence $P \leftrightarrow P'$ or a list equivalence $A \equiv A'$, provided the subterm P or A to be rewritten occurs in a **compatible position**. This form of rewriting is known as **setoid rewriting**. The following facts identify compatible positions by means of compatibility laws.

Fact 16.5.1 (Compatibility laws for propositional equivalence)
Let $P \leftrightarrow P'$ and $Q \leftrightarrow Q'$. Then:

$$\begin{array}{ccc} P \wedge Q \leftrightarrow P' \wedge Q' & P \vee Q \leftrightarrow P' \vee Q' & (P \to Q) \leftrightarrow (P' \to Q') \\ \neg P \leftrightarrow \neg P' & & (P \leftrightarrow Q) \leftrightarrow (P' \leftrightarrow Q') \end{array}$$

Fact 16.5.2 (Compatibility laws for list equivalence)
Let $A \equiv A'$ and $B \equiv B'$. Then:

$$\begin{array}{ccc} x \in A \leftrightarrow x \in A' & A \subseteq B \leftrightarrow A' \subseteq B' & A \equiv B \leftrightarrow A' \equiv B' \\ A +\!\!\!+ B \equiv A' +\!\!\!+ B' & f@A \equiv f@A' & A \,|\, f \equiv A' \,|\, f \\ x :: A \equiv x :: A' & & \end{array}$$

Coq's setoid rewriting facility makes it possible to use the rewriting tactic for rewriting with equivalences, provided the necessary compatibility laws and equivalence relations have been registered with the facility. The compatibility laws for propositional equivalence are preregistered.

Exercise 16.5.3 Which of the compatibility laws are needed to justify rewriting the claim $\neg(x \in y :: (f@A) +\!\!\!+ B)$ with the equivalence $A \equiv A'$?

16.6 Repeating Lists

A list is repeating if it contains some element more than once. For instance, $[1;2;1]$ is repeating and $[1;2;3]$ is non-repeating. Formally, we define **repeating lists** over a base type X with the inductive predicate

$$\begin{aligned} \mathsf{rep} &: \mathcal{L}(X) \to \mathbb{P} \\ \mathsf{rep}_{\mathsf{B}} &: \forall x A.\ x \in A \to \mathsf{rep}(x :: A) \\ \mathsf{rep}_{\mathsf{S}} &: \forall x A.\ \mathsf{rep}\,A \to \mathsf{rep}(x :: A) \end{aligned}$$

Fact 16.6.1 (Explicit Characterization)
$\mathsf{rep}\,A \;\leftrightarrow\; \exists x A_1 A_2 A_3.\ A = A_1 \mathbin{+\!\!+} x :: A_2 \mathbin{+\!\!+} x :: A_3.$

Proof Direction $\rightarrow$ follows by induction on $\mathsf{rep}\,A$. Direction $\leftarrow$ follows by induction on A_1 using Fact 16.2.5. ∎

We also define an inductive predicate for non-repeating lists over a base type X:

$$\begin{aligned} \mathsf{nrep} &: \ \mathcal{L}(X) \rightarrow \mathbb{P} \\ \mathsf{nrep}_\mathsf{B} &: \ \mathsf{nrep}\,[\,] \\ \mathsf{nrep}_\mathsf{S} &: \ \forall x A.\ x \notin A \rightarrow \mathsf{nrep}\,A \rightarrow \mathsf{nrep}(x :: A) \end{aligned}$$

We find the two-dimensional display of the proof constructors of rep and nrep as proof rules helpful.

$$\frac{x \in A}{\mathsf{rep}(x :: A)} \qquad \frac{\mathsf{rep}\,A}{\mathsf{rep}(x :: A)} \qquad\qquad \frac{}{\mathsf{nrep}\,[\,]} \qquad \frac{x \notin A \qquad \mathsf{nrep}\,A}{\mathsf{nrep}(x :: A)}$$

Theorem 16.6.2 (Partition) Let A be a list over a discrete type. Then:

1. $\mathsf{rep}\,A \rightarrow \mathsf{nrep}\,A \rightarrow \bot$.
2. $\mathsf{rep}\,A + \mathsf{nrep}\,A$.

Proof The first claim follows by induction on $\mathsf{rep}\,A$. The second claim follows by induction on A, where in the cons case with $\mathsf{nrep}\,A$ a case analysis on $\mathcal{D}(x \in A)$ is needed (justified by Fact 16.2.2). Discreteness is only needed for the second claim. ∎

Corollary 16.6.3 Let A be a list over a discrete type. Then:

1. $\mathcal{D}(\mathsf{rep}\,A)$ and $\mathcal{D}(\mathsf{nrep}\,A)$.
2. $\mathsf{rep}\,A \leftrightarrow \neg\mathsf{nrep}\,A$ and $\mathsf{nrep}\,A \leftrightarrow \neg\mathsf{rep}\,A$.

Exercise 16.6.4 (Inversion Lemmas) For the proof of the first claim of Theorem 16.6.2 one needs the inversion lemma $\mathsf{nrep}(x :: A) \rightarrow x \notin A \wedge \mathsf{nrep}\,A$. Prove the inversion lemma by hand. In Coq, the inversion lemma can be applied on the fly with the automation tactic depelim. Also prove the inversion lemmas $\mathsf{rep}\,[\,] \rightarrow \bot$ and $\mathsf{rep}(x :: A) \rightarrow x \in A \vee \mathsf{rep}\,A$.

Exercise 16.6.5 (Factorization) Let A be a list over a discrete type.
Prove $\mathsf{rep}\,A \;\leftrightarrow\; \Sigma x A_1 A_2 A_3.\ A = A_1 \mathbin{+\!\!+} x :: A_2 \mathbin{+\!\!+} x :: A_3$.

Exercise 16.6.6 (Partition) The proof of Corollary 16.6.3 is straightforward and follows a general scheme. Let P and Q be propositions such that $P \rightarrow Q \rightarrow \bot$ and $P + Q$. Prove $\mathsf{dec}\,P$ and $P \leftrightarrow \neg Q$. Note that $\mathsf{dec}\,Q$ and $Q \leftrightarrow \neg P$ follow by symmetry.

Exercise 16.6.7 (Even and Odd) Define inductive predicates even and odd for numbers and show that the predicates partition the numbers: $\mathsf{even}\,n \rightarrow \mathsf{odd}\,n \rightarrow \bot$ and $\mathsf{even}\,n + \mathsf{odd}\,n$.

16.7 Cardinality

The cardinality of a list is the number of different elements in the list. For instance, $[1;1;1]$ has cardinality 1 and $[1;2;3;2]$ has cardinality 3. Intuitively, it is clear that the cardinality of a list is bounded by its length, and that a list is non-repeating if and only if its cardinality equals its length.

In this section we assume that lists are taken over a discrete type X. Discreteness is needed so that we can define a function that yields the cardinality of a list.

We define the **cardinality function** for lists over a discrete type X as follows:

$$\begin{aligned}
\mathsf{card} :\ & \mathcal{L}(X) \to \mathsf{N} \\
\mathsf{card}\,[] \ &:=\ 0 \\
\mathsf{card}(x :: A) \ &:=\ \text{IF } \ulcorner x \in A \urcorner \text{ THEN } \mathsf{card}\,A \text{ ELSE } \mathsf{S}(\mathsf{card}\,A)
\end{aligned}$$

Note that we write $\ulcorner x \in A \urcorner$ for the application of the membership decider that is provided by Fact 16.2.2.

Fact 16.7.1 $\mathsf{card}\,A \le \mathsf{len}\,A$.

Proof By induction on A and case analysis on $\mathcal{D}(x \in A)$ in the cons case. ■

Fact 16.7.2 $\mathsf{rep}\,A \ \leftrightarrow\ \mathsf{card}\,A < \mathsf{len}\,A$.

Proof Direction $\rightarrow$ follows by induction on $\mathsf{rep}\,A$ and case analysis on $\mathcal{D}(x \in A)$ in both cases using Fact 16.7.1 for the base case. Direction $\leftarrow$ follows by induction on A and case analysis on $\mathcal{D}(x \in A)$ in the cons case. ■

Note that direction $\leftarrow$ of Fact 16.7.2 formulates a pigeonhole principle: If the length of a list is greater than its cardinality, then some element must occur at least twice in the list. One may see the positions of the list as the pigeons and the elements of the list as the holes.

Fact 16.7.3 $\mathsf{nrep}\,A \ \leftrightarrow\ \mathsf{card}\,A = \mathsf{len}\,A$.

Proof By Corollary 16.6.3 and Fact 16.7.2 it suffices to show the equivalence

$$\neg(\mathsf{card}\,A < \mathsf{len}\,A) \ \leftrightarrow\ \mathsf{card}\,A = \mathsf{len}\,A$$

which follows by contraposition (Fact 14.3.2), Fact 16.7.1, and antisymmetry. ■

16.8 Extensionality of Cardinality and Element Removal

We will now prove that equivalent lists have the same cardinality. We refer to this fact as extensionality of cardinality. While this fact is intuitively obvious, the proof requires a new idea and takes some effort. First note that

$$A \equiv B \;\to\; \mathsf{card}\,A = \mathsf{card}\,B$$

follows from the more general fact

$$A \subseteq B \;\to\; \mathsf{card}\,A \le \mathsf{card}\,B$$

We will prove this fact by induction on A. The base case is trivial. For the cons case we have to prove

$$x :: A \subseteq B \;\to\; \mathsf{card}(x :: A) \le \mathsf{card}\,B$$

given the inductive hypothesis. If $x \in A$, we have $\mathsf{card}(x :: A) = \mathsf{card}\,A$ and thus the claim follows by the inductive hypothesis. Otherwise, $x \notin A$ (the base type is discrete). The idea is now to use the inductive hypothesis for $A \subseteq B \setminus x$, where $B \setminus x$ is B with x removed. This yields

$$\mathsf{card}\,A \le \mathsf{card}(B \setminus x) < \mathsf{card}\,B$$

which yields the claim. Note that the above uses the lemma

$$x \in B \to \mathsf{card}(B \setminus x) < \mathsf{card}\,B$$

which will be shown by induction on B.

We start by defining the function $A \setminus x$ for **element removal**:

$$\begin{aligned} \setminus :\; & \mathcal{L}(X) \to X \to \mathcal{L}(X) \\ [\,] \setminus \_ \;&:=\; [\,] \\ (x :: A) \setminus y \;&:=\; \text{IF } x = y \text{ THEN } A \setminus y \text{ ELSE } x :: (A \setminus y) \end{aligned}$$

Fact 16.8.1 $x \in A \setminus y \;\leftrightarrow\; x \in A \land x \ne y$.

Proof By induction on A. In the cons case a case analysis on $\mathcal{D}(z = y)$ is needed to simplify $(z :: A) \setminus y$. ∎

Fact 16.8.2 $x \notin A \to A \setminus x = A$.

Proof By induction on A. ∎

Lemma 16.8.3 $x \in A \rightarrow \mathsf{card}\, A = \mathsf{S}(\mathsf{card}(A \setminus x))$.

Proof By induction on A. The base case is trivial. For the cons case we show

$$x \in (y :: A) \rightarrow \mathsf{card}(y :: A) = \mathsf{S}(\mathsf{card}((y :: A) \setminus x))$$

using the inductive hypothesis for A. We distinguish four cases.

1. $x = y$ and $y \in A$. Claim follows with inductive hypothesis.
2. $x = y$ and $y \notin A$. Claim follows with Fact 16.8.2.
3. $x \neq y$, $x \in A$, and $y \in A$. Claim follows with inductive hypothesis.
4. $x \neq y$, $x \in A$, and $y \notin A$. Claim follows with inductive hypothesis. ■

Theorem 16.8.4 $A \subseteq B \rightarrow \mathsf{card}\, A \leq \mathsf{card}\, B$.

Proof By induction on A with B quantified using Lemma 16.8.3 in the cons case. ■

Corollary 16.8.5 $A \equiv B \rightarrow \mathsf{card}\, A = \mathsf{card}\, B$.

Exercise 16.8.6 Prove $x \in A \rightarrow A \equiv x :: (A \setminus x)$.

Exercise 16.8.7 Prove $A \subseteq B \rightarrow \mathsf{len}\, B < \mathsf{len}\, A \rightarrow \mathsf{rep}\, A$. Note that this is yet another instance of the pigeonhole principle.

16.9 Map and Filter

We define the list operations **map** ($f@A$) and **filter** ($A\,|\,f$) by recursion on lists:

$$\begin{aligned} @ :\ & \forall XY.\ (X \rightarrow Y) \rightarrow \mathcal{L}(X) \rightarrow \mathcal{L}(Y) \\ f@\,[] &:= [] \\ f@(x :: A) &:= fx :: (f@A) \end{aligned}$$

$$\begin{aligned} |\ :\ & \forall X.\ \mathcal{L}(X) \rightarrow (X \rightarrow \mathsf{B}) \rightarrow \mathcal{L}(X) \\ []\,|\,f &:= [] \\ (x :: A)\,|\,f &:= \text{IF } fx \text{ THEN } x :: (A\,|\,f) \text{ ELSE } A\,|\,f \end{aligned}$$

Fact 16.9.1 (Membership)

$$\begin{aligned} x \in f@A &\leftrightarrow \exists a.\ a \in A \wedge x = fa \\ x \in A\,|\,f &\leftrightarrow x \in A \wedge fx = \mathsf{T} \end{aligned}$$

Proof By induction on A. ■

Corollary 16.9.2 (Inclusion)

$$\begin{aligned}
&A \subseteq A' \to f@A \subseteq f@A' \\
&A \subseteq A' \to A \,|\, f \subseteq A' \,|\, f \\
&(\forall x.\, x \in A \to fx = \mathsf{T} \to gx = \mathsf{T}) \to A \,|\, f \subseteq A \,|\, g \\
&A \,|\, f \subseteq A
\end{aligned}$$

Fact 16.9.3 (Length) $\mathsf{len}\,(f@A) = \mathsf{len}\,A$ and $\mathsf{len}\,(A \,|\, f) \le \mathsf{len}\,A$.

Proof By induction on A. ∎

Fact 16.9.4 (Filter Equations)

$$\begin{array}{ll}
(A + B) \,|\, f = (A \,|\, f) + (B \,|\, f) & (\forall x.\, x \in A \to fx = \mathsf{T}) \to A \,|\, f = A \\
(A \,|\, f) \,|\, g = (A \,|\, g) \,|\, f & (\forall x.\, x \in A \to fx = gx) \to A \,|\, f = A \,|\, g \\
(A \,|\, f) \,|\, g = A \,|\, (\lambda x. fx \,\&\, gx) &
\end{array}$$

Proof By induction on A. ∎

Exercise 16.9.5 (Pigeonhole) Let A be list over a discrete type. Prove
$\mathsf{card}(f@A) < \mathsf{card}\,A \to \exists xy.\, x \in A \wedge y \in A \wedge x \neq y \wedge fx = fy$.

16.10 Disjointness

Two lists are **disjoint** if they don't have a common element. We write $A \parallel B$ to say that A and B are disjoint lists over the same base type. Formally, there are three obvious characterizations of disjointness, where each of them provides for a formal definition of disjointness. The **explicit characterization**

$$A \parallel B \;\leftrightarrow\; \neg\exists x.\, x \in A \wedge x \in B$$

rephrases the informal characterization. The **inductive characterization**

$$\frac{}{[\,] \parallel B} \qquad\qquad \frac{x \notin B \qquad A \parallel B}{x :: A \parallel B}$$

implicitly realizes a recursion on A. Finally, the **recursive characterization** makes the recursion on A explicit:

$$\begin{aligned}
([\,] \parallel B) &= \top \\
(x :: A \parallel B) &= (x \notin b \wedge A \parallel B)
\end{aligned}$$

Fact 16.10.1 The three characterizations of disjointness are equivalent (explicit, inductive, recursive).

Proof The direction from the inductive predicate to the explicit predicate follows by induction on the inductive derivation. The converse direction follows by induction on A. The equivalence between the inductive and the recursive predicate follows with analogous inductions. ∎

Fact 16.10.2 Disjointness of lists over a discrete type is decidable.

Proof Follows with the explicit characterization, bounded quantification (Fact 16.2.3), and decidability of membership (Fact 16.2.2). ∎

Fact 16.10.3 List disjointness has the following properties:

1. $A \parallel B \to B \parallel A$
2. $A \subseteq A' \to A' \parallel B \to A \parallel B$
3. $A \equiv A' \to A \parallel B \leftrightarrow A' \parallel B$

Proof Claim 1 and claim 2 are obvious from the explicit characterization. Claim 3 is a straightforward consequence of claim 2. ∎

Exercise 16.10.4 Do all of the above definitions in Coq. The inductive characterization is best suited for the primary definition of disjointness. The equivalence with the explicit characterization is important for most proofs.

There is a detail at the Coq level we have not mentioned so far. Mathematically, the inductive disjointness predicate has two parameters X and B, and one index A. In Coq, however, the parameters must precede the indices, which makes both A and B into indices. The automatically derived eliminator thus models more dependencies then necessary. It turns out that this doesn't complicate proofs much. The minimal eliminator can be defined with the automatically derived eliminator. However, using the minimal eliminator in practice doesn't pay since it cannot be handled by Coq's induction tactic.

It one wants to work with a minimal eliminator in Coq, it is best to define the inductive disjointness predicate using recursion on the second list argument B.

17 Natural Deduction

We formalize and study basic proof systems for propositional logic in Coq's type theory. Our main system is an intuitinistic ND system for which we show that it cannot prove double negation using a Heyting interpretation. We contrast the intuitinistic system with a classical variant that is sound for boolean semantics. We prove Glivenko's theorem (classical provability reduces to intuitinistic provability) and a sandwich theorem for abstract entailment predicates. There is a follow-up chapter proving completeness and decidability of classical ND using the tableau method.

17.1 Intuitionistic ND

We consider **formulas** as follows:

$$s,t,u \ := \ x \mid \bot \mid s \to t \mid s \wedge t \mid s \vee t \qquad (x : \mathsf{N})$$

Formally, we accommodate formulas with an inductive type For representing each syntactic form with a value constructor. We use the notation $\neg s := s \to \bot$.

We define an inductive predicate $\vdash \ : \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{P}$ called **intuitionistic ND** as follows:

$$\frac{s \in A}{A \vdash s} \qquad \frac{A \vdash \bot}{A \vdash s} \qquad \frac{A, s \vdash t}{A \vdash s \to t} \qquad \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

$$\frac{A \vdash s \qquad A \vdash t}{A \vdash s \wedge t} \qquad \frac{A \vdash s \wedge t \qquad A, s, t \vdash u}{A \vdash u}$$

$$\frac{A \vdash s}{A \vdash s \vee t} \qquad \frac{A \vdash t}{A \vdash s \vee t} \qquad \frac{A \vdash s \vee t \qquad A, s \vdash u \qquad A, t \vdash u}{A \vdash u}$$

Note the notation $A, s := s :: A$ for lists of formulas. We will write $\vdash s$ for $[] \vdash s$. The symbol $\vdash$ is pronounced "turnstile".

A proposition $A \vdash s$ is like a goal in Coq. Given a proposition $A \vdash s$, we call A the **context** and s the **claim**. It turns out that intuitionistic ND can prove $A \vdash s$ if and only if Coq can prove the goal (A, s) where variables in formulas are accommodated as propositional variables in Coq.

The rules with a logical constant (i.e., $\bot$, $\to$, $\wedge$, $\vee$) in the conclusion are called **introduction rules**, and the rules with a logical constant in the leftmost premise are called **elimination rules**. The first rule in the above listing is called **assumption rule**. Note that every rule but the assumption rule is either an introduction or an elimination rule for a particular logical constant. Note that there is no introduction rule for $\bot$, and that there are two introduction rules for $\vee$. The elimination rule for $\bot$ is also called **explosion rule**.

Fact 17.1.1 $s \vdash \neg\neg s$ and $\neg\neg\bot \vdash \bot$.

Fact 17.1.2 (Cut) $A \vdash s \;\to\; A, s \vdash t \;\to\; A \vdash t$.

Proof Follows with the two implication rules. ■

Fact 17.1.3 (Weakening) $A \vdash s \to A \subseteq B \to B \vdash s$.

Proof By induction on $A \vdash s$ with B quantified. ■

Fact 17.1.4 (Implication) $A, s \vdash t \;\leftrightarrow\; A \vdash (s \to t)$.

Proof Follows with weakening. ■

We define a function $A \cdot s$ such that $[] \cdot s = s$ and $(u :: A) \cdot s = A \cdot (u \to s)$.

Fact 17.1.5 (Shift) $A \vdash s \leftrightarrow \;\vdash A \cdot s$.

Proof By induction on A using implication. ■

A formula is **ground** if it contains no variable.

Fact 17.1.6 (Ground Completeness) Let s be ground. Then either $\vdash s$ or $\vdash \neg s$.

Proof By induction on s. ■

Fact 17.1.7 (Double Application) $A \vdash (s_1 \to s_2 \to t) \;\to\; A \vdash s_1 \;\to\; A \vdash s_2 \;\to\; A \vdash t$.

Fact 17.1.8 (Double Negation)

1. $\neg\neg s \in A \;\to\; A, s \vdash \bot \;\to\; A \vdash \bot$.
2. $A, s, \neg t \vdash \bot \;\to\; A \vdash \neg\neg(s \to t)$.

Proof Follows with implication, cut, and double application. ■

Exercise 17.1.9 Prove the following propositions.

a) $(\neg s \to \neg\neg\bot) \vdash \neg\neg s$.

b) $(s \to \neg\neg t) \vdash \neg\neg(s \to t)$.

c) $\neg\neg(s \to t),\ \neg\neg s \vdash \neg\neg t$.

d) $\vdash (\neg\neg\neg s \to \neg s) \wedge (\neg s \to \neg\neg\neg s)$

Exercise 17.1.10 (Ground Completeness)

a) Define a boolean test ground for groundness of formulas.

b) Declare a function $\forall s.\ \text{IF ground}\, s\ \text{THEN}\ (\vdash s) + (\vdash \neg s)\ \text{ELSE}\ \top$.

c) Prove $\vdash s \vee \neg s$ for all ground formulas s.

d) Prove $A, s \vdash t \ \to\ A, \neg s \vdash t \ \to\ A \vdash t$ for all ground formulas s.

17.2 Heyting Entailment

We show $\neg\neg x \not\vdash x$. The trick is to come up with a predicate $A \vDash s$ such that (1) $A \vdash s \to A \vDash s$ and (2) $\neg\neg x \not\vDash x$. We will obtain such a predicate by evaluating formulas into a three-valued ordered type. We call the predicate $A \vDash s$ *Heyting entailment* in honor of the inventor Arend Heyting (around 1930).

Let V be an inductive type consisting of three values 0, 1, 2 we call **truth values**. We represent the linear order $0 < 1 < 2$ on V as a boolean function $a \le b$. An **assignment** is a function $\alpha : \mathsf{N} \to \mathsf{V}$. We define **evaluation of formulas** as follows:

$$\begin{aligned}
\mathcal{E}\alpha x &= \alpha x\\
\mathcal{E}\alpha\bot &= 0\\
\mathcal{E}\alpha(s \to t) &= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha t \text{ THEN } 2 \text{ ELSE } \mathcal{E}\alpha t\\
\mathcal{E}\alpha(s \wedge t) &= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha t \text{ THEN } \mathcal{E}\alpha s \text{ ELSE } \mathcal{E}\alpha t\\
\mathcal{E}\alpha(s \vee t) &= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha t \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathcal{E}\alpha s
\end{aligned}$$

Note that conjunction evaluates as minimum and disjunction evaluates as maximum. We extend evaluation to contexts such that A evaluates to the minimum of the truth values the formulas in A evaluate to:

$$\begin{aligned}
\mathcal{E}\alpha([]) &= 2\\
\mathcal{E}\alpha(s :: A) &= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha A \text{ THEN } \mathcal{E}\alpha s \text{ ELSE } \mathcal{E}\alpha A
\end{aligned}$$

We now define **Heyting entailment** as $A \vDash s := \forall \alpha.\ \mathcal{E}\alpha A \le \mathcal{E}\alpha s = \mathsf{T}$.

Fact 17.2.1 Let $\alpha n := 1$. Then $\mathcal{E}\alpha(\neg\neg x) = 2$ and $\mathcal{E}\alpha x = 1$.

Fact 17.2.2 (Soundness) $A \vdash s \ \to\ A \vDash s$.

Proof By induction on $A \vdash s$. ■

Theorem 17.2.3 (Double Negation) $\neg\neg x \nvdash x$.

Proof Follows with Facts 17.2.1 and 17.2.2. ■

Corollary 17.2.4 (Intuitionistic Consistency) $\nvdash \bot$.

Proof Follows with the explosion rule. ■

Exercise 17.2.5 Show $\nvdash x$ and $\nvdash x \vee \neg x$ and $\nvdash ((x \to y) \to x) \to x$.

17.3 Classical ND

Classical ND is obtained from intuitionistic ND by replacing the **ex falso rule**

$$\frac{A \vdash \bot}{A \vdash s}$$

with the **contradiction rule**:

$$\frac{A, \neg s \vdash \bot}{A \vdash s}$$

Formally, we have a separate inductive predicate $\dot\vdash : \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{P}$ for classic ND. Classical ND can prove double negation.

Fact 17.3.1 (Double Negation) $\neg\neg s \dot\vdash s$.

Fact 17.3.2 (Weakening) $A \dot\vdash s \to A \subseteq B \to B \dot\vdash s$.

Proof By induction on $A \dot\vdash s$ with B quantified. Same proof as for intuitionistic ND, except that now the proof obligation for the contradiction rule must be checked: $(\forall B.\ A, \neg s \subseteq B \to B \dot\vdash \bot) \ \to\ A \subseteq B \ \to\ B \dot\vdash s$. ■

Fact 17.3.3 (Explosion) $A \dot\vdash \bot \ \to\ A \dot\vdash s$.

Proof By contradiction and weakening. ■

Fact 17.3.4 (Extension) $A \vdash s \ \to\ A \dot\vdash s$.

Proof By induction on $A \vdash s$ using explosion. ■

Fact 17.3.5 (Implication) $A, s \dot\vdash t \ \leftrightarrow\ A \dot\vdash (s \to t)$.

Proof Follows with weakening. ■

Fact 17.3.6 (Cut) $A \dot{\vdash} s \ \rightarrow \ A, s \dot{\vdash} t \ \rightarrow \ A \dot{\vdash} t$.

Proof Follows with implication. ∎

Fact 17.3.7 (Refutation Completeness) $A \dot{\vdash} s \ \leftrightarrow \ A, \neg s \dot{\vdash} \bot$.

Proof Direction $\rightarrow$ follows with weakening. Direction $\leftarrow$ follows with contradiction. ∎

While refutation completeness tells us that classical ND can represent all information in the context, implication tells us that both intuitionistic and classical ND can represent all information in the claim.

Exercise 17.3.8 Show $\dot{\vdash} s \vee \neg s$ and $\dot{\vdash} ((s \rightarrow t) \rightarrow s) \rightarrow s$.

Exercise 17.3.9 Show that classical ND is not sound for Heyting entailment, that is, $\neg(\forall As.\ A \dot{\vdash} s \ \rightarrow \ A \vDash s)$.

17.4 Glivenko's Theorem

Lemma 17.4.1 $A \dot{\vdash} s \ \rightarrow \ A \vdash \neg\neg s$.

Proof By induction on $A \dot{\vdash} s$. This yields the following proof obligations.

1. $s \in A \ \rightarrow \ A \vdash \neg\neg s$.
2. $A, \neg s \vdash \neg\neg\bot \ \rightarrow \ A \vdash \neg\neg s$.
3. $A, s \vdash \neg\neg t \ \rightarrow \ A \vdash \neg\neg(s \rightarrow t)$.
4. $A \vdash \neg\neg(s \rightarrow t) \ \rightarrow \ A \vdash \neg\neg s \ \rightarrow \ A \vdash \neg\neg t$.

The obligations for conjunctions and disjunctions are omitted. The proofs are routine with Fact 17.1.8 and the other facts from Section 17.1. ∎

Theorem 17.4.2 $A \dot{\vdash} s \ \leftrightarrow \ A \vdash \neg\neg s$.

Proof Follows with Lemma 17.4.1 and Facts 17.3.4 and 17.3.1. ∎

Corollary 17.4.3 (Refutation Agreement) $A \vdash \bot \ \leftrightarrow \ A \dot{\vdash} \bot$.

Corollary 17.4.4 (Classical Consistency) $\neg(\dot{\vdash} \bot)$.

Proof Follows from intuitionistic consistency (Corollary 17.2.4). ∎

Corollary 17.4.5 Classical ND is decidable if intuitionistic ND is decidable.

Exercise 17.4.6 Show $\neg(\dot{\vdash} x)$ and $\neg(\dot{\vdash} \neg x)$.

Exercise 17.4.7 Show that $A \dot{\vdash} (s \vee t) \ \leftrightarrow \ A \dot{\vdash} s \vee A \dot{\vdash} t$ does not hold.

17.5 Boolean Entailment

Recall that classical ND is not sound for Heyting entailment (Exercise 17.3.9). We now define *boolean entailment* $A \mathrel{\dot{\models}} s$ using **boolean assignments** $\alpha : \mathsf{N} \to \mathsf{B}$. Classical ND will be sound for boolean entailment. We define **boolean evaluation** and **boolean entailment** as follows:

$$
\begin{aligned}
\mathcal{E}\alpha x &= \alpha x \\
\mathcal{E}\alpha\bot &= \mathsf{F} \\
\mathcal{E}\alpha(s \to t) &= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathsf{T} \\
\mathcal{E}\alpha(s \wedge t) &= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathsf{F} \\
\mathcal{E}\alpha(s \vee t) &= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathsf{T} \text{ ELSE } \mathcal{E}\alpha t \\
\mathcal{E}\alpha([\,]) &= \mathsf{T} \\
\mathcal{E}\alpha(s :: A) &= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha A \text{ ELSE } \mathsf{F} \\
A \mathrel{\dot{\models}} s &:= \forall\alpha.\ \text{IF } \mathcal{E}\alpha A \text{ THEN } \mathcal{E}\alpha s = \mathsf{T} \text{ ELSE } \top
\end{aligned}
$$

Fact 17.5.1 $A \mathrel{\dot{\models}} s \;\leftrightarrow\; \forall\alpha.\ (\forall u \in A.\ \mathcal{E}\alpha u = \mathsf{T}) \to \mathcal{E}\alpha s = \mathsf{T}$.

Fact 17.5.2 (Soundness) $A \mathrel{\dot{\vdash}} s \;\to\; A \mathrel{\dot{\models}} s$.

Proof By induction on $A \mathrel{\dot{\vdash}} s$. ■

Fact 17.5.3 (Refutation Completeness) $A \mathrel{\dot{\models}} s \;\leftrightarrow\; A, \neg s \mathrel{\dot{\models}} \bot$.

Exercise 17.5.4 Show $\neg(\dot{\vdash} \bot)$, $\neg(\dot{\vdash} x)$, and $\neg(\dot{\vdash} \neg x)$ using boolean entailment (rather than Heyting entailment and Glivenko).

Exercise 17.5.5 Show that $\dot{\vdash} s \vee t \leftrightarrow \dot{\vdash} s \vee \dot{\vdash} t$ does not hold for all t.

Exercise 17.5.6 Give a function f from formulas to formulas such that $\mathcal{E}\alpha(fst) = \mathcal{E}\alpha(s \wedge t)$ and all formulas fxy are obtained just with variables, falsity, and implication. Do the same for disjunction.

17.6 Substitution

A **substitution** is a function $\theta : \mathsf{N} \to \mathsf{For}$ mapping every variable to a formula. We define application of substitutions to formulas and lists of formulas such that every variable is replaced by the term provided by the substitution:

$$\begin{aligned}
\theta\cdot x &= \theta x \\
\theta\cdot\bot &= \bot \\
\theta\cdot(s\to t) &= \theta\cdot s\to\theta\cdot t \\
\theta\cdot(s\wedge t) &= \theta\cdot s\wedge\theta\cdot t \\
\theta\cdot(s\vee t) &= \theta\cdot s\vee\theta\cdot t
\end{aligned}$$

$$\begin{aligned}
\theta\cdot[\,] &= [\,] \\
\theta\cdot(s :: A) &= \theta\cdot s :: \theta\cdot A
\end{aligned}$$

We will write θs and θA for $\theta\cdot s$ and $\theta\cdot A$.

Fact 17.6.1 $s\in A\to\theta s\in\theta A$.

Proof By induction on A. ∎

Fact 17.6.2 $A\vdash s\to\theta A\vdash\theta s$ and $A\dot{\vdash} s\to\theta A\dot{\vdash}\theta s$.

Proof By induction on $A\vdash s$ and $A\dot{\vdash} s$ using Fact 17.6.1. ∎

Lemma 17.6.3 $\mathcal{E}\alpha(\theta s)=\mathcal{E}(\lambda n.\mathcal{E}\alpha(\theta n))\,s$ holds both for Heyting and for boolean evaluation.

Fact 17.6.4 $A\vDash s\to\theta A\vDash\theta s$ and $A\dot{\vDash} s\to\theta A\dot{\vDash}\theta s$.

Proof By induction on A using Lemma 17.6.3. ∎

17.7 Entailment Predicates

An **entailment predicate** is a predicate $\Vdash\;:\;\mathcal{L}(\mathsf{For})\to\mathsf{For}\to\mathbb{P}$ satisfying the properties listed in Figure 17.1. Note that the first four requirements don't make any assumptions on formulas; they are called **structural requirements.** Each of the remaining requirements concerns a particular form of formulas: Variables, falsity, implication, conjunction, and disjunction.

Fact 17.7.1 Intuitionistic ND ($A\vdash s$), classical ND ($A\dot{\vdash} s$), Heyting entailment ($A\vDash s$), and boolean entailment ($A\dot{\vDash} s$) are entailment predicates.

Proof Follows with the consistency and substitution results shown in the preceding sections. ∎

1. *Assumption:* $s \in A \to A \Vdash s$.
2. *Cut:* $A \Vdash s \;\to\; A, s \Vdash t \;\to\; A \Vdash t$.
3. *Weakening:* $A \Vdash s \;\to\; A \subseteq B \;\to\; B \Vdash s$.
4. *Consistency:* $\exists s.\ \nVdash s$.
5. *Substitutivity:* $A \Vdash s \;\to\; \theta A \Vdash \theta s$.
6. *Explosion:* $A \Vdash \bot \;\to\; A \Vdash s$.
7. *Implication:* $A \Vdash (s \to t) \;\leftrightarrow\; A, s \Vdash t$.
8. *Conjunction:* $A \Vdash (s \wedge t) \;\leftrightarrow\; A \Vdash s \;\wedge\; A \Vdash t$.
9. *Disjunction:* $A \Vdash (s \vee t) \;\leftrightarrow\; \forall u.\ A, s \Vdash u \;\to\; A, t \Vdash u \;\to\; A \Vdash u$.

Figure 17.1: Requirements for entailment predicates

We will show that every entailment predicate $\Vdash$ satisfies $A \vdash s \to A \Vdash s$ and $A \Vdash s \to A \dot{\vDash} s$; that is, every entailment predicate is sandwiched between intuitionistic ND at the bottom and boolean entailment at the top. Let $\Vdash$ be an entailment predicate in the following.

Fact 17.7.2 (Modus Ponens) $A \Vdash (s \to t) \;\to\; A \Vdash s \;\to\; A \Vdash t$.

Proof By implication and cut. ■

Fact 17.7.3 $A \vdash s \;\to\; A \Vdash s$. That is, intuitionistic ND is a least entailment predicate.

Proof By induction on $A \vdash s$ using modus ponens. ■

Fact 17.7.4 $\Vdash s \;\to\; \Vdash \neg s \;\to\; \bot$.

Proof Let $\Vdash s$ and $\Vdash \neg s$. By Fact 17.7.2 we have $\Vdash \bot$. By consistency and explosion we obtain a contradiction. ■

Fact 17.7.5 (Shift) $A \Vdash s \leftrightarrow\; \Vdash A \cdot s$.

Proof By induction on A using implication. ■

We now come to the key lemma for showing that abstract entailment implies boolean entailment. The lemma was conceived by Tobias Tebbi in 2015. We define a conversion function that given a boolean assignment $\alpha : \mathsf{N} \to \mathsf{B}$ yields a substitution as follows: $\hat{\alpha} n :=$ IF αn THEN $\neg\bot$ ELSE $\bot$.

Lemma 17.7.6 (Tebbi) IF $\mathcal{E}\alpha s$ THEN $\Vdash \hat{\alpha} s$ ELSE $\Vdash \neg \hat{\alpha} s$.

Proof Induction on s using Fact 17.7.2 and assumption, weakening, explosion, and implication. ■

Note that we have formulated the lemma with a conditional. While this style of formulation is uncommon in Mathematics it is compact and convenient in a type theory with computational equality.

Lemma 17.7.7 $\Vdash s \to \dot{\vDash} s$.

Proof Let $\Vdash s$ and α. We assume $\mathcal{E}\alpha s = \mathsf{F}$ and derive a contradiction. By Tebbi's Lemma we have $\Vdash \neg\hat{\alpha}s$. By substitutivity we obtain $\Vdash \hat{\alpha}s$ from the primary assumption. Contradiction by Fact 17.7.4. ■

Theorem 17.7.8 (Sandwich)
Let $\Vdash$ be an entailment predicate. Then $A \vdash s \;\to\; A \Vdash s$ and $A \Vdash s \;\to\; A \dot{\vDash} s$.

Proof Claim 1 is Fact 17.7.3. Claim 2 follows with Lemma 17.7.7 and Facts 17.7.5 and 17.7.1. ■

Exercise 17.7.9 Let $\Vdash$ be an entailment predicate. Prove the following:

a) $\forall s.\, \mathsf{ground}\ s \to (\Vdash s) + (\Vdash \neg s)$.

b) $\forall s.\, \mathsf{ground}\ s \to \mathsf{dec}(\Vdash s)$.

Exercise 17.7.10 Tebbi's lemma provides for a particularly elegant proof of Lemma 17.7.7. Verify that Lemma 17.7.7 can also be obtained from the facts (1) $\vdash \hat{\alpha}s \vee \vdash \neg\hat{\alpha}s$ and (2) $\dot{\vDash} \hat{\alpha}s \to \mathcal{E}\alpha s = \mathsf{T}$ using Facts 17.7.3 and 17.7.4.

17.8 Outlook

Completeness of classical ND One can show that classical ND agrees with boolean entailment. The direction $A \dot{\vDash} s \to A \dot{\vdash} s$ we don't have is called completeness of classical ND.

Decidability of classical ND One can show that classical ND is decidable.

Decidability of intuitionistic ND One can show that intuitionistic ND is decidable. This can be done with a method devised by Gentzen in the 1930s. First one shows that intuitionistic ND is equivalent to a proof system called sequent calculus that has the subformula property. Then one shows that sequent calculus is decidable, which is feasible since it has the subformula property.

Kripke structures and Heyting structures One can construct evaluation-based entailment predicates that coincide with intuitionistic ND using either finite Heyting structures or finite Kripke structures. In contrast to classical ND, where a single two-valued boolean structure invalidates all classically unprovable formulas, one needs either infinitely many finite Heyting structures or infinitely many finite Kripke structures to invalidate all intuitionistically unprovable formulas. Heyting structures are usually presented as Heyting algebras and were invented by Arend Heyting around 1930. Kripke structures were invented by Saul Kripke in the late 1950's.

Certifying Solver for intuitionistic ND One can construct a certifying solver for intuitionistic ND using the tableau method. Given A and s, the solver yields either a proof of $A \vdash s$ or a finite Kripke structure satisfying A and dissatisfying s. From the certifying solver one can obtain a decision function for intuitionistic ND. The tableau method was developed starting in 1955 by Evert Beth, Raymond Smullyan, and Melvin Fitting.

Intuitionistic Independence of logical constants Using boolean entailment, one can show that falsity and implication can express conjunction and disjunction. On the other hand, one can prove using Heyting structures that in intuitionistic ND the logical constants are independent.

Exercise 17.8.1 Assume a function $\forall A.\ (\exists \alpha.\ \mathcal{E}\alpha A = \top) + (A \vdash \bot)$ and show that classical ND is complete and decidable.

18 Boolean Satisfiability

We study satisfiability of boolean formulas using a DNF-based solver and a tableau system. The solver translates boolean formulas to equivalent clausal DNFs and thereby decides satisfiability. The tableau system provides a proof system for unsatisfiability and bridges the gap between natural deduction and satisfiability. Based on the tableau system one can prove completeness and decidability of propositional natural deduction.

The development presented here works for any choice of boolean connectives, except for the final step making the connection with natural deduction. The independence from particular connectives is obtained by representing conjunctions and disjunctions with lists and negations with signs.

The (formal) proofs of the development are instructive in that they showcast the interplay between evaluation of expressions, nontrivial recursive functions (the DNF solver), and inductive predicates (the tableau system). Of particular interest is the completeness proof for the tableau system, which is obtained by functional induction on the recursion structure of the DNF solver.

18.1 Boolean Operations

We will work with the boolean operations conjunction, disjunction, and negation, which we define as follows:

$$\begin{array}{lll} \mathsf{T} \mathbin{\&} b = b \qquad & \mathsf{T} \mid b = \mathsf{T} \qquad & !\mathsf{T} = \mathsf{F} \\ \mathsf{F} \mathbin{\&} b = \mathsf{F} & \mathsf{F} \mid b = b & !\mathsf{F} = \mathsf{T} \end{array}$$

With these definitions all boolean identities have straightforward proofs by boolean case analysis and computation. Recall that boolean conjunction and disjunction are commutative and associative.

The idea behind disjunctive normal form (DNF) is that conjunctions are below disjunctions, and that negations are below conjunctions. Negations can be pushed downwards with the negation laws

$$!(a \mathbin{\&} b) = !a \mid !b \qquad !(a \mid b) = !a \mathbin{\&} !b \qquad !!a = a$$

and conjunctions can be pushed below disjunctions with the distribution law

$$a \mathbin{\&} (b \mid c) = (a \mathbin{\&} b) \mid (a \mathbin{\&} b)$$

We will also make use of the negation law

$$b \wedge !b = \mathsf{F}$$

to eliminate conjunctions.

There are also the **reflection laws**

$$\begin{aligned} a \,\&\, b = \mathsf{T} &\leftrightarrow a = \mathsf{T} \wedge b = \mathsf{T} \\ a \mid b = \mathsf{T} &\leftrightarrow a = \mathsf{T} \vee b = \mathsf{T} \\ !a = \mathsf{T} &\leftrightarrow \neg(a = \mathsf{T}) \end{aligned}$$

which offer the possibility to replace boolean operations with logical connectives. As it comes to proofs, this is usually a bad idea since one looses the computation coming with the boolean operations. An exception is the reflection rule for conjunction, which offers the possibility to replace the argument terms of a conjunction with T.

18.2 Boolean Formulas

We will consider the boolean **formulas**

$$s, t, u \;:=\; x \mid \bot \mid s \to t \mid s \wedge t \mid s \vee t \qquad (x : \mathsf{N})$$

realized with an inductive data type For representing each syntactic form with a value constructor. **Variables** x are represented as numbers.

Our development will work with any choice of boolean connectives for formulas. We have made the unusual design decision to have boolean implication as an explicit connective. On the other hand, we have omitted truth $\top$ and negation $\neg$. We accommodate truth and negation with the notations

$$\top := \bot \to \bot \qquad\qquad \neg s := s \to \bot$$

An **assignment** is a function $\alpha : \mathsf{N} \to \mathsf{B}$ mapping every variable to a boolean. We define the **evaluation function** for boolean formulas as shown in Figure 18.1. Note that every function $\mathcal{E}\alpha$ translates boolean formulas (object level) to boolean terms (meta level). Also note that implications are expressed with negation and disjunction. We define the notation

$$\alpha \vDash s \;:=\; \mathcal{E}\alpha s = \mathsf{T}$$

and say that α **satisfies** s, or that α **solves** s, or that α is a **solution** of s. We say that a formula s is **satisfiable** and write sat s if s has a solution. Finally, we say that two formulas are **equivalent** if they have the same solutions.

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= \mathsf{F} \\
\mathcal{E}\alpha(s \to t) &:= \,!\mathcal{E}\alpha s \mid \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \wedge t) &:= \mathcal{E}\alpha s \,\&\, \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \vee t) &:= \mathcal{E}\alpha s \mid \mathcal{E}\alpha t
\end{aligned}$$

Figure 18.1: Definition of the evaluation function $\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathsf{B}$

As it comes to proofs, it will be important to keep in mind that the notation $\alpha \vDash s$ abbreviates the boolean equation $\mathcal{E}\alpha s = \mathsf{T}$. Reasoning with boolean equations will be the main workhorse in our proofs.

Exercise 18.2.1 Convince yourself that the predicate $\alpha \vDash s$ is decidable.

Exercise 18.2.2 Verify a function translating formulas into equivalent formulas not containing conjunctions and disjunctions.

Exercise 18.2.3 Verify the reflection laws

$$\begin{aligned}
\alpha \vDash (s \wedge t) &\leftrightarrow \alpha \vDash s \wedge \alpha \vDash t \\
\alpha \vDash (s \vee t) &\leftrightarrow \alpha \vDash s \vee \alpha \vDash t \\
\alpha \vDash \neg s &\leftrightarrow \neg(\alpha \vDash s)
\end{aligned}$$

18.3 Clausal DNFs

Informally, a *DNF* (disjunctive normal form) is a disjunction $s_1 \vee \cdots \vee s_n$ of *solved formulas* s_i, where a solved formula is a conjunction of variables and negated variables where no variable appears both negated and unnegated. One can show that every formula is equivalent to a DNF. There may be many different DNFs for a formula. For instance, the DNFs $x \vee \neg x$ and $y \vee \neg y$ are equivalent since they are satisfied by every assignment. On the other hand, we will arrange the exact DNF format such that all unsatisfiable formulas have the same DNF, which may be seen as the empty disjunction.

Formulas by themselves are not a good data structure for computing DNFs of formulas. We will work with lists of signed formulas we call clauses:

S, T :	SFor	$::=$	$s^+ \mid s^-$	**signed formula**
C, D :	Cla	$:=$	$\mathcal{L}(\mathsf{SFor})$	**clause**

Clauses represent conjunctions. We define evaluation of signed formulas and clauses as follows:

$$\mathcal{E}\alpha(s^+) := \mathcal{E}\alpha s \qquad\qquad \mathcal{E}\alpha\,[] := \mathsf{T}$$
$$\mathcal{E}\alpha(s^-) := \,!\mathcal{E}\alpha s \qquad\qquad \mathcal{E}\alpha(S :: C) := \mathcal{E}\alpha S \;\&\; \mathcal{E}\alpha C$$

Note that the empty clause represents truth. We also consider lists of clauses

$$\Delta : \mathcal{L}(\mathsf{Cla})$$

and interpret them disjunctively:

$$\mathcal{E}\alpha\,[] := \mathsf{F}$$
$$\mathcal{E}\alpha\,(C :: \Delta) := \mathcal{E}\alpha C \mid \mathcal{E}\alpha\Delta$$

Satisfaction of signed formulas, clauses, and lists of clauses is defined analogously to formulas, and so are the notations $\alpha \vDash S$, $\alpha \vDash C$, $\alpha \vDash \Delta$, and sat C. Since formulas, signed formulas, clauses, and lists of clauses all come with the notion of satisfying assignments, we can speak about **equivalence** between these objects although they belong to different types. For instance, s, s^+, $[s^+]$, and $[[s^+]]$, are all equivalent since they are satisfied by the same assignments.

A **solved clause** is a clause consisting of signed variables (i.e., x^+ and x^-) such that no variable appears positively and negatively. Note that a solved clause C is satisfied by every assignment that maps the positive variables in C to T and the negative variables in C to F.

Fact 18.3.1 Solved clauses are satisfiable. More specifically, a solved clause C is satisfied by the assignment $\lambda x.\, \ulcorner x^+ \in C \urcorner$.

A **clausal DNF** is a list of solved clauses.

Corollary 18.3.2 Every nonempty clausal DNF is satisfiable.

Exercise 18.3.3 Prove $\mathcal{E}\alpha(C + D) = \mathcal{E}\alpha C \;\&\; \mathcal{E}\alpha D$ and $\mathcal{E}\alpha(\Delta + \Delta') = \mathcal{E}\alpha\Delta \mid \mathcal{E}\alpha\Delta'$.

Exercise 18.3.4 Write a function that maps lists of clauses to equivalent formulas.

Exercise 18.3.5 Our formal proof of Fact 18.3.1 is unexpectedly tedious in that it requires two inductive lemmas:

1. $\alpha \vDash C \;\leftrightarrow\; \forall S \in C.\ \alpha \vDash S$.
2. $\mathsf{solved}\ C \;\to\; S \in C \;\to\; \exists x.\ (S = x^+ \wedge x^- \notin C) \vee (S = x^- \wedge x^+ \notin C)$.

The formal development captures solved clauses with an inductive predicate. This is convenient for most purposes but doesn't provide for a convenient proof of Fact 18.3.1. Can you do better?

$$
\begin{aligned}
\mathsf{dnf}\ C\ [] &:= [C] \\
\mathsf{dnf}\ C\ (x^+ :: D) &:= \text{IF } \ulcorner x^- \in C \urcorner \text{ THEN } [] \text{ ELSE } \mathsf{dnf}\ (x^+ :: C)\ D \\
\mathsf{dnf}\ C\ (x^- :: D) &:= \text{IF } \ulcorner x^+ \in C \urcorner \text{ THEN } [] \text{ ELSE } \mathsf{dnf}\ (x^- :: C)\ D \\
\mathsf{dnf}\ C\ (\bot^+ :: D) &:= [] \\
\mathsf{dnf}\ C\ (\bot^- :: D) &:= \mathsf{dnf}\ C\ D \\
\mathsf{dnf}\ C\ ((s \to t)^+ :: D) &:= \mathsf{dnf}\ C\ (s^- :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \to t)^- :: D) &:= \mathsf{dnf}\ C\ (s^+ :: t^- :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^+ :: D) &:= \mathsf{dnf}\ C\ (s^+ :: t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^- :: D) &:= \mathsf{dnf}\ C\ (s^- :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^- :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^+ :: D) &:= \mathsf{dnf}\ C\ (s^+ :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^- :: D) &:= \mathsf{dnf}\ C\ (s^- :: t^- :: D)
\end{aligned}
$$

Figure 18.2: Definition of the DNF function $\mathsf{dnf} : \mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$

18.4 DNF Function

We now define a function dnf that for every clause yields an equivalent clausal DNF. The function has the type

$$\mathsf{dnf}\ :\ \mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$$

and satisfies two **correctness properties**:

$$\mathcal{E}\alpha(\mathsf{dnf}\ C\ D) = \mathcal{E}\alpha C\ \&\ \mathcal{E}\alpha D \tag{18.1}$$

$$\mathsf{solved}\ C\ \to\ E \in \mathsf{dnf}\ C\ D\ \to\ \mathsf{solved}\ E \tag{18.2}$$

Thus $\mathsf{dnf}\ []\ [s^+]$ computes a clausal DNF for the formula s. The second argument of dnf (the **agenda**) holds the signed formulas still to be processed, and the first argument of dnf (the **accumulator**) collects the signed variables taken from the agenda. The function dnf is recursive and processes the formulas on the agenda one by one decreasing the size of the agenda with every recursion step. The equations defining dnf are shown in Figure 18.2. Note that the defining equations are clear from the two correctness properties, the boolean identities given in Section 18.1, and the idea that the first formula on the agenda controls the recursion.

Theorem 18.4.1 $\mathsf{dnf}\ []\ C$ is a clausal DNF equivalent to C.

Proof The statement of the theorem follows from the two correctness properties given above. Both correctness properties follow by induction on the recursion structure of dnf. There are 13 cases for each of the inductions where every case is straightforward. ■

Corollary 18.4.2 C is satisfiable if and only if $\mathsf{dnf}\ []\ C$ is nonempty.

Corollary 18.4.3 Satisfiability of clauses and formulas is decidable.

Corollary 18.4.4 There is a solver $\forall C.\ (\Sigma\alpha.\ \alpha \vDash C) + \neg\mathsf{sat}\ C$.

Corollary 18.4.5 There is a solver $\forall s.\ (\Sigma\alpha.\ \alpha \vDash s) + \neg\mathsf{sat}\ s$.

Exercise 18.4.6 Convince yourself that the predicate $S \in C$ is decidable.

Exercise 18.4.7 Write a size function for clauses such that every recursion step of the DNF function decreases the size of the agenda.

Exercise 18.4.8 Rewrite the DNF function so that you obtain a boolean decider $\mathcal{D} : \mathsf{Cla} \to \mathsf{Cla} \to \mathsf{B}$ for satisfiability of clauses. Find suitable correctness properties and verify the correctness of $\mathcal{D}$.

18.5 Validity

A formula is **valid** if it is satisfied by all assignments. Validity reduces to unsatisfiability, and satisfiability reduces to non-validity. The latter fact follows with the decidability of satisfiability.

Fact 18.5.1

1. A formula s is valid if and only if its negation is unsatisfiable.
2. A formula s is satisfiable if and only if its negation is not valid.

Proof Both directions of (1) and the left-to-right direction of (2) are routine. The right-to-left direction of (2) follows by proof by contradiction, which is justified since satisfiability of formulas is decidable (Corollary 18.4.3). ■

Exercise 18.5.2 Declare a function $\forall s.\ \mathsf{valid}\ s + (\Sigma\alpha.\ \mathcal{E}\alpha s = \mathsf{F})$ that checks whether a formula is valid and returns a counterexample in the negative case.

$$\frac{\mathsf{tab}\,(S :: C + D)}{\mathsf{tab}\,(C + S :: D)} \qquad \frac{}{\mathsf{tab}\,(x^+ :: x^- :: C)} \qquad \frac{}{\mathsf{tab}\,(\bot^+ :: C)}$$

$$\frac{\mathsf{tab}\,(s^- :: C) \qquad \mathsf{tab}\,(t^+ :: C)}{\mathsf{tab}\,((s \to t)^+ :: C)} \qquad \frac{\mathsf{tab}\,(s^+ :: t^- :: C)}{\mathsf{tab}\,((s \to t)^- :: C)}$$

$$\frac{\mathsf{tab}\,(s^+ :: t^+ :: C)}{\mathsf{tab}\,((s \wedge t)^+ :: C)} \qquad \frac{\mathsf{tab}\,(s^- :: C) \qquad \mathsf{tab}\,(t^- :: C)}{\mathsf{tab}\,((s \wedge t)^- :: C)}$$

$$\frac{\mathsf{tab}\,(s^+ :: C) \qquad \mathsf{tab}\,(t^+ :: C)}{\mathsf{tab}\,((s \vee t)^+ :: C)} \qquad \frac{\mathsf{tab}\,(s^- :: t^- :: C)}{\mathsf{tab}\,((s \vee t)^- :: C)}$$

Figure 18.3: Definition of $\mathsf{tab} : \mathsf{Cla} \to \mathbb{P}$

18.6 Tableau Predicate

The DNF function can be reformulated into an inductive predicate that derives exactly the unsatisfiable clauses. Because termination is no longer an issue, the accumulator argument is not needed anymore. Instead we add a rule that moves signed formulas in the agenda. Figure 18.3 shows the resulting inductive predicate tab. We speak of a **tableau predicate** since tab formalizes a proof system that belongs to the family of tableau systems. We call the rules defining tab tableau rules.

We refer to the first rule of the tableau predicate as **move rule** and to the second rule as **clash rule**. Note the use of list concatenation in the move rule.

The tableau rules are best understood in backwards fashion (from the conclusion to the premises). All but the first rule are decomposition rules simplifying the clause to be derived. The second and third rule derive clauses that are obviously unsatisfiable. The move rule is needed so that non-variable formulas can be moved to the front of a clause as it is required by most of the other rules.

Fact 18.6.1 (Soundness) Clauses derivable with tab are unsatisfiable.

Proof $\mathsf{tab}\ C \to \alpha \vDash C \to \bot$ follows by induction on tab. ■

Fact 18.6.2 (Weakening) The following rules hold for tab:

$$\frac{\mathsf{tab}\,(C)}{\mathsf{tab}\,(S :: C)}$$

Proof By induction on tab. ■

The move rule is strong enough to reorder clauses freely.

Fact 18.6.3 (Move Rules) The following rules hold for tab:

$$\frac{\mathsf{tab}\,(\mathsf{rev}\,D + C + E)}{\mathsf{tab}\,(C + D + E)} \qquad \frac{\mathsf{tab}\,(D + C + E)}{\mathsf{tab}\,(C + D + E)} \qquad \frac{\mathsf{tab}\,(C + S :: D)}{\mathsf{tab}\,(S :: C + D)}$$

We refer to the last rule as **inverse move rule**.

Proof The first rule follows by induction on tab. The second rule follows from the first rule with $C = []$ and $\mathsf{rev}\,(\mathsf{rev}\,D) = D$. The third rule follows from the second rule with $C = [S]$. ■

Lemma 18.6.4 $\mathsf{dnf}\ C\ D = [] \to \mathsf{tab}\,(D + C)$.

Proof By functional induction on $\mathsf{dnf}\ C\ D$ using the weakening and inverse move rule. The weakening rule is needed for the deletion of $\bot^-$ and the inverse move rule is needed to account for the move of variables from the agenda to the accumulator. ■

The proof of Lemma 18.6.4 demonstrates the power of functional induction. With functional induction we can do induction on the recursion structure of dnf, which is exactly what we need for constructing tableau derivations for unsatisfiable clauses.

Theorem 18.6.5 The clauses derivable with tab are exactly the unsatisfiable clauses.

Proof Follows with Fact 18.6.1, Corollary 18.4.2, and Lemma 18.6.4. ■

Corollary 18.6.6 The tableau predicate is decidable.

We remark that the DNF function and the tableau predicate adapt to any choice of boolean connectives. We just add or delete equations as needed. An extreme case would be to not have variables. That one can choose the boolean connectives freely is due to the use of clauses with signed formulas.

The tableau rules have the **subformula property**, that is, a derivation of a clause C does only employ subformulas of formulas in C. That the tableau rules satisfies the subformula property can be verified rule by rule.

Exercise 18.6.7 Prove $\mathsf{tab}\,(C + S :: D + T :: E) \leftrightarrow \mathsf{tab}\,(C + T :: D + S :: E)$.

Exercise 18.6.8 Give an inductive predicate that derives exactly the satisfiable clauses. Start with an inductive predicate deriving exactly the solved clauses.

18.7 Refutation Predicates

An **unsigned clause** is a list of formulas. We will now consider a tableau predicate for unsigned clauses that comes close to the refutation predicate associated with natural deduction. For the tableau predicate we will show decidability and agreement with unsatisfiability. Based on the results for the tableau predicate one can prove decidability and completeness of classical natural deduction.

The switch to unsigned clauses requires negation and falsity, but as it comes to the other connectives we are still free to choose what we want. Negation could be accommodated as an additional connective, but formally we continue to represent negation with implication and falsity.

We can turn a signed clause C into an unsigned clause by replacing positive formulas s^+ with s and negative formulas s^- with negations $\neg s$. We can also turn an unsigned clause into a signed clause by labeling every formula with the positive sign. The two conversions do not change the boolean value of a clause for a given assignment. Moreover, going from an unsigned clause to a signed clause and back yields the initial clause. From the above it is clear that satisfiability of unsigned clauses reduces to satisfiability of signed clauses and thus is decidable.

Formalizing the above ideas is straightforward. The letters A and B will range over unsigned clauses. We define $\alpha \vDash A$ and satisfiability of unsigned clauses analogous to signed clauses. We use $\hat{C}$ to denote the unsigned version of a signed clause and A^+ to denote the signed version of an unsigned clause.

Fact 18.7.1 $\mathcal{E}\alpha\hat{C} = \mathcal{E}\alpha C$, $\mathcal{E}\alpha A^+ = \mathcal{E}\alpha A$, and $\widehat{A^+} = A$.

Fact 18.7.2 (Decidability) Satisfiability of unsigned clauses is decidable.

Proof Follows with Corollary 18.4.3 and $\mathcal{E}\alpha A^+ = \mathcal{E}\alpha A$. ■

We call a predicate ρ on unsigned clauses a **refutation predicate** if it satisfies the rules in Figure 18.4. Note that the rules are obtained from the tableau rules for signed clauses by replacing positive formulas s^+ with s and negative formulas s^- with negations $\neg s$.

Lemma 18.7.3 Let ρ be a refutation predicate. Then $\mathsf{tab}\ C \rightarrow \rho\hat{C}$.

Proof Straightforward by induction on $\mathsf{tab}\ C$. ■

Fact 18.7.4 (Completeness)
Every refutation predicate holds for all unsatisfiable unsigned clauses.

Proof Follows with Theorem 18.6.5 and Lemma 18.7.3. ■

$$\frac{\rho\,(s :: A +\!\!\!+ B)}{\rho\,(A +\!\!\!+ s :: B)} \qquad \frac{}{\rho\,(x :: \neg x :: A)} \qquad \frac{}{\rho\,(\bot :: A)}$$

$$\frac{\rho\,(\neg s :: A) \qquad \rho\,(t :: A)}{\rho\,((s \to t) :: A)} \qquad \frac{\rho\,(s :: \neg t :: A)}{\rho\,(\neg(s \to t) :: A)}$$

$$\frac{\rho\,(s :: t :: A)}{\rho\,((s \wedge t) :: A)} \qquad \frac{\rho\,(\neg s :: A) \qquad \rho\,(\neg t :: A)}{\rho\,(\neg(s \wedge t) :: A)}$$

$$\frac{\rho\,(s :: A) \qquad \rho\,(t :: A)}{\rho\,((s \vee t) :: A)} \qquad \frac{\rho\,(\neg s :: \neg t :: A)}{\rho\,(\neg(s \vee t) :: A)}$$

Figure 18.4: Rules for refutation predicates $\rho : \mathcal{L}(\mathsf{For}) \to \mathbb{P}$

We call a refutation predicate **sound** if it holds only for unsatisfiable unsigned clauses (that is, $\forall A.\ \rho A \to \neg \mathsf{sat}\, A$).

Fact 18.7.5 Every sound refutation predicate is decidable and holds exactly for unsatisfiable unsigned clauses.

Proof Facts 18.7.4 and 18.7.2. ■

Theorem 18.7.6 The minimal refutation predicate inductively defined with the rules for refutation predicates derives exactly the unsatisfiable unsigned clauses.

Proof Follows with Fact 18.7.4 and a soundness lemma similar to Fact 18.6.1. ■

Exercise 18.7.7 (Certifying Solver) Declare a function $\forall A.\ (\Sigma\alpha.\ \alpha \vDash A) + \neg \mathsf{sat}\, A$.

Exercise 18.7.8 Show that boolean entailment

$$A \dot{\vDash} s \ := \ \forall \alpha.\ \alpha \vDash A \to \alpha \vDash s$$

is decidable.

Exercise 18.7.9 Let $A \dot{\vdash} s$ be the inductive predicate for classical natural deduction. Prove that $A \dot{\vdash} s$ is decidable and agrees with boolean entailment. Hint: Exploit refutation completeness and show that $A \dot{\vdash} \bot$ is a refutation predicate.

19 Well-founded Recursion

Type theory admits only total functions. That is, a function must yield a result for every argument that is admitted by the type of the function. Consequently, recursively defined functions must be restricted such that the recursion terminates for every argument. Coq's type theory imposes a particularly strong termination requirement that requires recursive functions to be structurally recursive on a fixed inductive argument.

It turns out that structural recursion is surprisingly expressive in a higher-order type theory. In fact, it is fair to say that a recursively specified function can be defined in Coq provided the recursion can be shown terminating in Coq. There is a systematic method that yields a definition of a recursively specified function by first defining an auxiliary function that recurses on an extra argument providing a structural termination certificate. Termination certificates are obtained as derivations for special inductive predicates representing well-founded relations that are known as accessibility predicates.

19.1 Accessibility Predicate for Numbers

We start with the accessibility predicate for numbers, which captures the well-foundedness of the canonical order $x < y$ for numbers. Well-foundedness means that there are no infinite decreasing chains $x_0 > x_1 > x_2 > \cdots$. Thus a process that periodically decreases a number must terminate. Using suggestive notation, we can write the definition of the accessibility predicate for numbers as follows:

$$\mathsf{Acc}\,(x : \mathsf{N}) : \mathbb{P} \;:=\; [\mathsf{Acc_I}\,\langle\forall y.\, y < x \to \mathsf{Acc}\,y\rangle]$$

According to this definition, a derivation $\mathsf{Acc_I}\,h$ of a proposition $\mathsf{Acc}\,x$ comprises a **continuation function**

$$h : \forall y.\, y < x \to \mathsf{Acc}\,y$$

that for every number y and every proof $H : y < x$ yields a derivation hyH of $\mathsf{Acc}\,y$ that is structurally smaller than the derivation $\mathsf{Acc_I}\,h$. This means that structural recursion on $\mathsf{Acc}\,x$ can recurse on every y for which there is a proof of $y < x$.

The types of the two constructors for the accessibility predicate are as follows:

$$\begin{aligned} \mathsf{Acc} &: \mathsf{N} \to \mathbb{P} \\ \mathsf{Acc_I} &: \forall x.\, (\forall y.\, y < x \to \mathsf{Acc}\, y) \to \mathsf{Acc}\, x \end{aligned}$$

Note that the first argument x of $\mathsf{Acc_I}$ is a (nonuniform) parameter and the second argument of $\mathsf{Acc_I}$ is a proof. Thus derivations of Acc are exempted from the elim restriction.

We can now define functions $\forall x.\, \mathsf{Acc}\, x \to px$ that recurse on derivations of $\mathsf{Acc}\, x$. Given the continuation function provided by a derivation of $\mathsf{Acc}\, x$, we can have recursive calls for all $y < x$. Moreover, given a function $\alpha : \forall x.\, \mathsf{Acc}\, x$ and a function $g : \forall x.\, \mathsf{Acc}\, x \to px$, we can define a function $\lambda x.\, gx(\alpha x) : \forall x.\, px$ that informally speaking may recurse on every $y < x$. The trick is that we structurally recurse on a derivation of $\mathsf{Acc}\, x$ rather than the argument x itself.

Before we consider concrete examples, we first show some facts for Acc.

We call a number x **accessible** if there is a derivation of $\mathsf{Acc}\, x$. It turns out that every number is accessible.

Fact 19.1.1 (Seed Function) There is a function $\alpha : \forall x : \mathsf{Acc}\, x$.

Proof By induction on x.

For the base case, we need a function $\forall y.\, y < 0 \to \mathsf{Acc}\, y$. Straightforward since we can get a proof of $y < 0$ yields a proof of falsity.

For the successor case, we need a function $\forall y.\, y < \mathsf{S}x \to \mathsf{Acc}\, y$. We assume $y < \mathsf{S}x$ and show $\mathsf{Acc}\, y$ by constructing a continuation function $\forall z.\, z < y \to \mathsf{Acc}\, z$. We assume $z < y$ and show $\mathsf{Acc}\, z$. Since $z < x$, the inductive hypothesis gives us a derivation of $\mathsf{Acc}\, z$. ■

We derive an eliminator for Acc.

Fact 19.1.2 (Eliminator) There is a function
$\mathsf{E_{Acc}} : \forall p^{\mathsf{N}\to\mathbb{T}}.\, (\forall x.\, (\forall y.\, y < x \to py) \to px) \to \forall x.\, \mathsf{Acc}\, x \to px$.

Proof $\lambda pf.\, \textsc{fix}\, Fxa.\, \textsc{match}\, a\, [\, \mathsf{Acc_I}\, h \Rightarrow fx(\lambda yH.\, Fy(hyH))\,]$. ■

With the eliminator and the seed function we can derive an operator for complete induction.

Fact 19.1.3 (Complete Induction) There is a function
$\forall p^{\mathsf{N}\to\mathbb{T}}.\, (\forall x.\, (\forall y.\, y < x \to py) \to px) \to \forall x. px$.

Proof $\lambda pfx.\, \mathsf{E_{Acc}}\, pfx(\alpha x)$. ■

The two facts and their proofs suggest that complete induction may be seen as an abstract formulation of Acc recursion. As it comes to the definition of recursive functions, direct using Acc recursion turns out to be more appropriate.

As it comes to the so far informal notion that the order $x < y$ on numbers is well-founded, we now have two ways to express it formally: either by saying that every number is accessible or by stating the complete induction principle.

Exercise 19.1.4 Prove $\mathsf{Acc}\,x \;\leftrightarrow\; \forall y.\, y < x \to \mathsf{Acc}\,y$.

19.2 Example: Division

Given two numbers x and y, we may divide x by $\mathsf{S}y$. The division yields the maximal number $\mathsf{S}y$ can be subtracted from x without truncation. We may specify such a division function $D : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ with the conditional equation

$$Dxy \;=\; \begin{cases} x & \text{if } x \le y \\ D(x - \mathsf{S}y)y & \text{if } x > y \end{cases} \tag{19.1}$$

The conditional equation may be realized with a decider

$$\delta : \forall xy.\, (x \le y) + (x > y)$$

which we assume given in the following. The specification suggests a definition

$$Dxy \;:=\; D'xy(\alpha x)$$

with an auxiliary function $D' : \forall x.\, \mathsf{N} \to \mathsf{Acc}\,x \to \mathsf{N}$ that recurses on the derivation of $\mathsf{Acc}\,x$.

19.3 Lead Example: Division

Consider the following specification of a *division function*:

$$\begin{aligned} D :\ & \mathsf{N} \to \mathsf{N} \to \mathsf{N} && && (19.2)\\ D\,x\,0 &= 0 \\ D\,x\,(\mathsf{S}y) &= 0 && \text{if } x \le y && (19.3)\\ &= \mathsf{S}\,(D\,(x - \mathsf{S}y)\,(\mathsf{S}y)) && \text{if } x > y \end{aligned}$$

Here are the essential features of the specification:

1. There is a nonstandard equation $D\,x\,0 = 0$ so that we fully specify a total function $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$.[1]
2. There is a structural case analysis on the second argument.
3. There is a nested case analysis using the function $\forall xy.\ (x \le y) + (x > y)$.
4. There is a (nonstructural) recursion decreasing the first argument upon recursion (happens in the third equation).

The main concern of this chapter is the presentation of a method that for specifications like the one above defines a function satisfying the specification. The specification must be presented such that the exhaustiveness of the equations is obvious. Moreover, a **termination function** must be given as part of the specification, which for D is the function mapping the two arguments to the first argument. For each recursive application a proof obligation saying that the arguments are smaller must be shown. For D the exact proof obligation for termination is

$$\forall xy.\, x > y \to x - \mathsf{S}y < x$$

Following the scheme used for D, we can also specify a *modulo function*:

$$\begin{aligned} M :\ & \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ M\,x\,0 &= 0 \\ M\,x\,(\mathsf{S}y) &= x && \text{if } x \le y \\ &= M\,(x - \mathsf{S}y)\,(\mathsf{S}y) && \text{if } x > y \end{aligned}$$

Given functions satisfying the above specifications, we can for instance prove that the functions satisfy the equation of the division theorem.

Fact 19.3.1 $x = Dx(\mathsf{S}y) \cdot \mathsf{S}y + Mx(\mathsf{S}y)$ and $Mx(\mathsf{S}y) \le y$.

Proof Both claims follow by complete induction on x and can be shown independently. We show the first claim.

[1] Recall that Coq's type theory omits only total functions.

Following the specification of D and M, we do a case analysis $(x \le y) + (x > y)$.

If $x \le y$, the claim is $x = 0 \cdot \mathsf{S}y + x$, an equation that holds by computational equality.

If $x > y$, the claim is

$$x = \mathsf{S}\,(D\,(x - \mathsf{S}y)\,(\mathsf{S}y)) \cdot \mathsf{S}y + M\,(x - \mathsf{S}y)\,(\mathsf{S}y)$$

Since $x - \mathsf{S}y < x$, the inductive hypothesis gives us

$$x - \mathsf{S}y = D\,(x - \mathsf{S}y)\,(\mathsf{S}y) \cdot \mathsf{S}y + M\,(x - \mathsf{S}y)\,(\mathsf{S}y)$$

The claim follows since $x = \mathsf{S}y + (x - \mathsf{S}y)$. ■

19.4 Acc Recursion for Numbers

We now show how D and M can be realized with auxiliary functions that are structurally recursive on an extra argument. The type of the recursive argument is

$$\begin{aligned} \mathsf{Acc} :&\ \ \mathsf{N} \to \mathbb{P} \\ \mathsf{Acc_I} :&\ \ \forall x.\ (\forall y.\ y < x \to \mathsf{Acc}\,y) \to \mathsf{Acc}\,x \end{aligned}$$

Note that the argument of Acc is a (nonuniform) parameter and that Acc is excepted from the elim restriction since the nonparametric argument of $\mathsf{Acc_I}$ is a proof. Also note that a value $\mathsf{Acc_I}\,xh$ of $\mathsf{Acc}\,x$ carries a function h that for every $y < x$ yields a value of $\mathsf{Acc}\,y$ that is structurally smaller than $\mathsf{Acc_I}\,xh$. This way a function that recurses on an argument $\mathsf{Acc}\,x$ can structurally recurse on every $y < x$.

Fact 19.4.1 $\forall x^{\mathsf{N}}.\ \mathsf{Acc}\,x$.

Proof By complete induction on x. ■

Fact 19.4.2 $\mathsf{Acc}\,x \leftrightarrow \forall y.\ y < x \to \mathsf{Acc}\,y$.

Proof Straightforward. Direction $\to$ follows with destructuring. Direction $\leftarrow$ follows with $\mathsf{Acc_I}$. ■

We now define an auxiliary function D' that recurses structurally on an extra argument of type $\mathsf{Acc}\,x$

$$\begin{aligned} D' :&\ \ \forall x^{\mathsf{N}}.\ \mathsf{N} \to \mathsf{Acc}\,x \to \mathsf{N} \\ D'\,x\,0\,(\mathsf{Acc_I}\,\_\,h) \ :=&\ \ 0 \\ D'\,x\,(\mathsf{S}y)\,(\mathsf{Acc_I}\,\_\,h) \ :=&\ \ \text{MATCH}\ \delta xy \\ &\ \ [\,\mathsf{L}\,\_ \Rightarrow 0 \\ &\ \ \mid \mathsf{R}\,H \Rightarrow \mathsf{S}\,(D\,(x - \mathsf{S}y)\,(\mathsf{S}y)\,(h\,(x - \mathsf{S}y)\,(\tau xyH)))\,] \end{aligned}$$

and uses two auxiliary functions

$$\begin{aligned}\delta &: \forall xy.\,(x \le y) + (x > y)\\ \tau &: \forall xy.\, x > y \to x - \mathsf{S}y < x\end{aligned}$$

whose exact definition does not matter. We now define

$$Dxy \;:=\; D'xy(\alpha x)$$

where α is some function $\forall x^{\mathsf{N}}.\,\mathsf{Acc}\,x$ as established by Fact 19.4.1. Things can be arranged such that D in fact reduces (it suffices that α is defined transparently).

To show properties about D, we need complete induction and the three specifying equations:

$$Dx0 = 0 \tag{19.4}$$

$$x \le y \to Dx(Sy) = x \tag{19.5}$$

$$x > y \to Dx(Sy) = \mathsf{S}\,(D\,(x - \mathsf{S}y)\,(\mathsf{S}y)) \tag{19.6}$$

Equations (19.4) and (19.5) can be shown by unfolding and case analysis. Equation (19.6) follows by unfolding and case analysis and the lemma

$$\forall aa'.\,D'xya \;=\; D'xya' \tag{19.7}$$

which follows by complete induction. Note that Lemma (19.7) says that D' does not distinguish between different derivations of $\mathsf{Acc}\,x$. We say that D' is **extensional** for its Acc argument.

Acc is known as an **accessibility predicate**. **Acc recursion** (i.e., recursion on an argument of type $\mathsf{Acc}\,x$) provides for well-founded recursion on numbers. For this purpose it is essential that Acc is not subject to the elim restriction.

Summary of Inductive Definitions

We list the most important inductive definitions we have seen in this text. In each case we give the equational definition of an eliminator and point out which arguments are non-uniform parameters or indices.

Booleans

$$\begin{aligned} \mathsf{B} &: \mathbb{T} \\ \mathsf{T} &: \mathsf{B} \\ \mathsf{F} &: \mathsf{B} \end{aligned}$$

$$\begin{aligned} \mathsf{E_B} &: \forall p^{\mathsf{B}\to\mathbb{T}}.\ p\,\mathsf{T} \to p\,\mathsf{F} \to \forall x.px \\ \mathsf{E_B}\,p\,a\,b\,\mathsf{T} &:= a \\ \mathsf{E_B}\,p\,a\,b\,\mathsf{F} &:= b \end{aligned}$$

$$\mathsf{E_B} = \lambda pabx.\ \text{MATCH}\ x\ [\,\mathsf{T} \Rightarrow a \mid \mathsf{F} \Rightarrow b\,]$$

Numbers

$$\begin{aligned} \mathsf{N} &: \mathbb{T} \\ 0 &: \mathsf{N} \\ \mathsf{S} &: \mathsf{N} \to \mathsf{N} \end{aligned}$$

$$\begin{aligned} \mathsf{E_N} &: \forall p^{\mathsf{N}\to\mathbb{T}}.\ p\,0 \to (\forall n.\ pn \to p(\mathsf{S}n)) \to \forall n.pn \\ \mathsf{E_N}\,p\,a\,f\,0 &:= a \\ \mathsf{E_N}\,p\,a\,f\,(\mathsf{S}n) &:= fn\,(\mathsf{E_N}\,pafn) \end{aligned}$$

$$\mathsf{E_N} = \lambda paf.\ \text{FIX}\ Fn.\ \text{MATCH}\ n\ [\,0 \Rightarrow a \mid \mathsf{S}n' \Rightarrow fn'(Fn')\,]$$

Pairs and Product Types

$$\begin{aligned} \times &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{pair} &: \forall X^{\mathbb{T}} Y^{\mathbb{T}}.\ X \to Y \to X \times Y \end{aligned}$$

$$\begin{aligned} &\mathsf{E}_\times : \ \forall X^{\mathbb{T}}\, Y^{\mathbb{T}}\, p^{X \times Y \to \mathbb{T}}.\ (\forall xy.\ p(\mathsf{pair}\, XYxy)) \to \forall a.pa \\ &\mathsf{E}_\times\, XYpf\, (\mathsf{pair}\, \_\,\_\, xy) \ := \ fxy \end{aligned}$$

$$\mathsf{E}_\times \ = \ \lambda XYpfh.\ \text{MATCH}\ h\ [\, \mathsf{pair}\, \_\,\_\, xy \Rightarrow fxy\,]$$

Decisions and Sum Types

$$\begin{aligned} + &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{L} &: \forall X^{\mathbb{T}}\, Y^{\mathbb{T}}.\ X \to X + Y \\ \mathsf{R} &: \forall X^{\mathbb{T}}\, Y^{\mathbb{T}}.\ Y \to X + Y \end{aligned}$$

$$\begin{aligned} &\mathsf{E}_+ : \ \forall X^{\mathbb{T}}\, Y^{\mathbb{T}}\, p^{X + Y \to \mathbb{T}}.\ (\forall x.\ p(\mathsf{L}\, XYx)) \to (\forall y.\ p(\mathsf{R}\, XYy)) \to \forall a.pa \\ &\mathsf{E}_+\, XYpfg\, (\mathsf{L}\, \_\,\_\, x) \ := \ fx \\ &\mathsf{E}_+\, XYpfg\, (\mathsf{R}\, \_\,\_\, y) \ := \ gy \end{aligned}$$

$$\mathsf{E}_+ \ = \ \lambda XYpfga.\ \text{MATCH}\ s\ [\, \mathsf{L}\, \_\,\_\, x \Rightarrow fx \mid \mathsf{R}\, \_\,\_\, y \Rightarrow gy\,]$$

Existential Quantification

$$\begin{aligned} \mathsf{ex} &: \forall X^{\mathbb{T}}.\ (X \to \mathbb{P}) \to \mathbb{P} \\ \mathsf{ex_I} &: \forall X^{\mathbb{T}}\, p^{X \to \mathbb{P}}\, x^X.\ px \to \mathsf{ex}\, X\, p \end{aligned}$$

$$\begin{aligned} &\mathsf{E_{ex}} : \ \forall X^{\mathbb{T}}\, p^{X \to \mathbb{P}}\, Z^{\mathbb{P}}.\ (\forall x.\ px \to Z) \to \mathsf{ex}\, Xp \to Z \\ &\mathsf{E_{ex}}\, XpZf\, (\mathsf{ex_I}\, \_\,\_\, xa) \ := \ fxa \end{aligned}$$

$$\mathsf{E_{ex}} \ = \ \lambda XpZfh.\ \text{MATCH}\ h\ [\, \mathsf{ex_I}\, \_\,\_\, xa \Rightarrow fxa\,]$$

Dependent Pairs and Sigma Types

$$\begin{aligned}\mathsf{sig} :\;& \forall X^{\mathbb{T}}.\,(X \to \mathbb{T}) \to \mathbb{T}\\ \mathsf{sig_I} :\;& \forall X^{\mathbb{T}}\, p^{X\to\mathbb{T}}\, x^{X}.\; px \to \mathsf{sig}\,X\,p\end{aligned}$$

$$\begin{aligned}&\mathsf{E_{sig}} :\; \forall X^{\mathbb{T}}\, p^{X\to\mathbb{T}}\, q^{\mathsf{sig}\,X\,p\,\to\mathbb{T}}.\,(\forall xb.\,q(\mathsf{sig_I}\,Xpxb)) \to \forall a^{\mathsf{sig}\,Xp}.\,qa\\ &\mathsf{E_{sig}}\,Xpqf\,(\mathsf{sig_I}\,\_\,\_\,x\,b) \;:=\; fxb\end{aligned}$$

$$\mathsf{E_{sig}} \;=\; \lambda Xpqfh.\,\textsc{match}\;h\;[\,\mathsf{sig_I}\,\_\,\_\,xb \Rightarrow fxb\,]$$

Transfer Predicate for Witness Operator

$$\begin{aligned}\mathsf{G} :\;& (\mathsf{N} \to \mathsf{B}) \to \mathsf{N} \to \mathbb{P}\\ \mathsf{G_I} :\;& \forall f^{\mathsf{N}\to\mathsf{B}}\, n^{\mathsf{N}}.\;\; (fn = \mathsf{F} \to \mathsf{G}\,f\,(\mathsf{S}n)) \to \mathsf{G}\,fn\end{aligned}$$

$$\begin{aligned}&\mathsf{E_G} :\; \forall f^{\mathsf{N}\to\mathsf{B}}\, p^{\mathsf{N}\to\mathbb{T}}.\,(\forall n.\,(fn = \mathsf{F} \to p(\mathsf{S}n)) \to pn) \to \forall n.\,\mathsf{G}\,fn \to pn\\ &\mathsf{E_G}\,fpg\,n\,(\mathsf{G_I}\,\_\,\_\,h) \;:=\; g\,n\,(\lambda e.\,\mathsf{E_G}\,fpg\,(\mathsf{S}n)\,(he))\end{aligned}$$

$$\mathsf{E_G} \;=\; \lambda fpg.\,\textsc{fix}\;Fnh.\,\textsc{match}\;h\;[\,\mathsf{G_I}\,\_\,\_\,\mathsf{h}' \Rightarrow gn(\lambda e.\,F(\mathsf{S}n)(h'e))\,]$$

The second argument of G is a non-uniform parameter.
$\mathsf{G_I}$ has a functional argument that is recursive (guarded recursion).

Zero

$$\begin{aligned}\mathsf{zero} :\;& \mathsf{N} \to \mathbb{P}\\ \mathsf{Z} :\;& \mathsf{zero}\;0\end{aligned}$$

$$\begin{aligned}&\mathsf{E_{zero}} :\; \forall p^{\mathsf{N}\to\mathbb{T}}.\;\; p0 \to \forall x.\,\mathsf{zero}\;x \to px\\ &\mathsf{E_{zero}}\;pa\,\_\,\mathsf{Z} \;:=\; a \qquad : p0\end{aligned}$$

$$\mathsf{E_{zero}} \;=\; \lambda paxh.\,\textsc{match}\;h\;[\,\mathsf{Z} \Rightarrow a\,]$$

The single argument of zero is an index.

Inductive Equality

$$\begin{aligned} \mathsf{eq} &: \forall X^{\mathbb{T}}.\ X \to X \to \mathbb{P} \\ \mathsf{Q} &: \forall X^{\mathbb{T}}\, x^X.\ \mathsf{eq}Xxx \end{aligned}$$

$$\begin{aligned} &\mathsf{E}_{\mathsf{eq}} : \forall X^{\mathbb{T}}\, x^X\, p^{X \to \mathbb{T}}.\ px \to \forall y.\ \mathsf{eq}Xxy \to py \\ &\mathsf{E}_{\mathsf{eq}}\ Xxpa\,\_\,(\mathsf{Q}\,\_\,\_) \ := \ a \qquad : px \end{aligned}$$

$$\mathsf{E}_{\mathsf{eq}} \ = \ \lambda Xxpaxh.\ \textsc{match}\ h\ [\,\mathsf{Q}\,\_\,\_ \Rightarrow a\,]$$

The second argument of eq is an index.

Even Numbers

$$\begin{aligned} \mathsf{even} &: \mathsf{N} \to \mathbb{P} \\ \mathsf{even_B} &: \mathsf{even}\ 0 \\ \mathsf{even_S} &: \forall n.\ \mathsf{even}\ n \to \mathsf{even}(\mathsf{S}(\mathsf{S}n)) \end{aligned}$$

$$\begin{aligned} \mathsf{E}_{\mathsf{even}} &: \forall p^{\mathsf{N} \to \mathbb{P}}.\ p0 \to (\forall n.\ \mathsf{even}\,n \to pn \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.\ \mathsf{even}\ n \to pn \\ \mathsf{E}_{\mathsf{even}}\ p\,a\,f\,\_\ \mathsf{even_B} &:= a \qquad\qquad : p0 \\ \mathsf{E}_{\mathsf{even}}\ p\,a\,f\,\_\,(\mathsf{even_S}\,n'h) &:= fn'h\,(\mathsf{E}_{\mathsf{even}}\,pafn'h) \quad : p(\mathsf{S}(\mathsf{S}n')) \end{aligned}$$

$$\mathsf{E}_{\mathsf{even}} \ = \ \lambda paf.\ \textsc{fix}\ Fnh.\ \textsc{match}\ h\ [\,\mathsf{even_B} \Rightarrow a \mid \mathsf{even_S}\,n'h' \Rightarrow fn'h'(Fn'h')\,]$$

The single argument of even is an index.

Strict Positivity Condition

Coq's type theory disallows recursive proof constructors where the recursion passes through the left hand side of a (dependent) function type (so-called strict positivity condition). For instance, the following inductive definition is not admissible:

$$\begin{aligned} \mathsf{even} &: \mathsf{N} \to \mathbb{P} \\ \mathsf{even_B} &: \mathsf{even}\ 0 \\ \mathsf{even_S} &: \forall n.\ \neg\mathsf{even}\ n \to \mathsf{even}(\mathsf{S}n) \end{aligned}$$

Note that recursion through the right hand side of (dependent) function types is fine and is used for the constructor $\mathsf{G_I}$ of the transfer predicate G.