Computational Type Theory and Interactive Theorem Proving with Coq

Lecture Notes Summer 2020
Introduction to Computational Logic

Version of July 20, 2020

Gert Smolka
Saarland University
Saarland Informatics Campus

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

$$\mathsf{B} ::= \mathbf{T} \mid \mathbf{F}$$

introduces three typed constants called **constructors**:

$$\mathsf{B} : \mathbb{T}$$
$$\mathbf{T} : \mathsf{B}$$
$$\mathbf{F} : \mathsf{B}$$

The constructors represent the type B and its two values $\mathbf{T}$ and $\mathbf{F}$. Note that the constructor B also has a type, which is the **universe** $\mathbb{T}$ (a type of types).

Inductive types provide for the **inductive definition of functions**, where a **defining equation** is given for each value constructor. We demonstrate this feature with an inductive definition of a boolean negation function:

$$!: \mathsf{B} \to \mathsf{B}$$
$$!\mathbf{T} := \mathbf{F}$$
$$!\mathbf{F} := \mathbf{T}$$

There is a defining equation for each of the two value constructors of B. The defining equations serve as **computation rules**. For computation, the equations are applied as left-to-right rewrite rules. For instance, we have

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

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

$$!!!\mathbf{T} = !\mathbf{T}$$

follows by computation:

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

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 $!!\mathbf{T} = \mathbf{T}$ and $!!\mathbf{F} = \mathbf{F}$, which both follow by computation.

Next we define functions for boolean conjunction and boolean disjunction:

$$\begin{array}{ll} \& : \mathsf{B} \to \mathsf{B} \to \mathsf{B} & \quad | : \mathsf{B} \to \mathsf{B} \to \mathsf{B} \\ \mathbf{T} \,\&\, y := y & \quad \mathbf{T} \mid y := \mathbf{T} \\ \mathbf{F} \,\&\, y := \mathbf{F} & \quad \mathbf{F} \mid y := y \end{array}$$

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 $\mathbf{F} = \mathbf{F}$. 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, ...

$$\mathsf{N} ::= 0 \mid \mathsf{S}(\mathsf{N})$$

introduces three constructors

$$\begin{array}{l} \mathsf{N} : \mathbb{T} \\ 0 : \mathsf{N} \\ \mathsf{S} : \mathsf{N} \to \mathsf{N} \end{array}$$

The value constructors provide 0 and the successor function S. A number n can 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, S0, S(S0), ... 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:

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

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.

		$x + 0 = x$	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

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

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.

		$x + y - y = x$	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

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.

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 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 1.3.4 Prove that addition is commutative (1.3). You will need equations (1.1) and (1.2) as lemmas.

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 multiplication is commutative. You will need lemmas.

Exercise 1.3.7 (Truncating subtraction) Truncating subtraction is different from the familiar subtraction in that it yields 0 where standard subtraction yields a negative number. Truncating subtraction has the nice property that $x \leq y$ if and only if $x - y = 0$. Prove the following equations:

a) $x - 0 = x$

b) $(x + y) - x = y$

c) $x - x = 0$

d) $x - (x + y) = 0$

Note that $(x - y) + (y - x)$ is the distance between x and y. Write a function $D : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ that computes the distance between two numbers with a single recursion. Try to prove $Dxy = (x - y) + (y - x)$ by induction on x. The proof requires an inductive hypothesis that quantifies over y, a standard technique discussed in detail in Section 6.2.

Exercise 1.3.8 (Maximum) Define a recursive maximum function $M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ and prove $M(x + y)x = x + y$ and $Mx(x + y) = x + y$. 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.

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]

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

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

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

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.

1.5 Strict Structural Recursion

The equations

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

specify a function $E : \mathsf{N} \to \mathsf{B}$ that checks whether a number is even. The recursion appearing in the third equation is structural but not **strictly structural**. We can define a function satisfying the three equations with strict structural recursion if we make use of boolean negation:

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

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

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

The equations

$$F0 = 0$$
$$F1 = 1$$
$$F(\mathsf{S}(\mathsf{S}n)) = Fn + F(\mathsf{S}n)$$

specify the **Fibonacci function** $F : \mathsf{N} \to \mathsf{N}$. The third equation does not qualify for structural recursion (because of the recursive application $F(\mathsf{S}n)$). It is possible to define F with strict structural recursion using an auxiliary function with two extra arguments (Section 9.2).

Coq does not insist on strict structural recursion and accepts structural recursion with some extras. We will not make use of this feature and stick to strict structural recursion throughout this text. Later we will introduce a technique that reduces general terminating recursion to strict structural higher-order recursion.

Exercise 1.5.1 Prove $\mathsf{E}(n \cdot 2) = \mathbf{T}$.

Exercise 1.5.2 Define a function $H : \mathsf{N} \to \mathsf{N}$ satisfying the equations

$$H\,0 = 0$$
$$H\,1 = 0$$
$$H(\mathsf{S}(\mathsf{S}n)) = \mathsf{S}(Hn)$$

using strict structural recursion. Hint: Use an auxiliary function with an extra boolean argument.

1.6 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

$$\mathsf{Pair} \;::=\; \mathsf{pair}(\mathsf{N}, \mathsf{N})$$

which introduces two constructors

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

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:

$$\mathsf{Pair}(X : \mathbb{T},\ Y : \mathbb{T})\ ::=\ \mathsf{pair}(X, Y)$$

This inductive type definition gives us two constructors:

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

The **polymorphic value constructor** pair 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. Put differently, the types X and Y taken as first and second argument provide the types for the components of the pair constructed.

We shall use the familiar notation $X \times Y$ for **product types** $\mathsf{Pair}\ X\ Y$.

We can write **partial applications** of the value constructor pair:

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

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.

1.7 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 $\mathsf{pair}\,0\,\mathbf{T}$ for $\mathsf{pair}\,\mathsf{N}\,\mathsf{B}\,0\,\mathbf{T}$. 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:

$$\pi_1 : \forall X\, Y.\ X \times Y \to X \qquad \pi_2 : \forall X\, Y.\ X \times Y \to Y$$
$$\pi_1\,(x, y) := x \qquad \pi_2\,(x, y) := y$$

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

Exercise 1.7.2 Let a be a variable of type $X \times Y$. Write proof diagrams for the equations $\mathsf{swap}\,(\mathsf{swap}\,a) = a$ and $(\pi_1 a, \pi_2 a) = a$.

1.8 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 1.8.1 Verify the equation $\mathsf{iter}\ \mathsf{S}\ 2 = \lambda x.\ \mathsf{S}(\mathsf{S}x)$ by computation.

Exercise 1.8.2 Prove $n + x = \mathsf{iter}\ \mathsf{S}\ n\ x$ and $x^n = \mathsf{iter}\ (\cdot x)\ n\ 1$ by induction.

Exercise 1.8.3 (Shift) Prove $\mathsf{iter}\ f\ (\mathsf{S}n)\ x = \mathsf{iter}\ f\ n\ (fx)$ by induction.

Exercise 1.8.4 (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 1.8.5 (Even) $\mathsf{iter}\ !\ n\ \mathbf{T}$ tests whether n is even. Prove $\mathsf{iter}\ !\ (n \cdot 2)\ b = b$ and $\mathsf{iter}\ !\ (\mathsf{S}(n \cdot 2))\ b = !b$.

1.9 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:

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

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

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

1.10 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 $f\,\mathbf{T} = \mathbf{T}$ together with $f\,\mathbf{T} = \mathbf{F}$ 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.

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

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**

$$\text{FIX}\ f^{\sigma\to\tau}\ x^{\sigma}.\ s$$

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 13.2.3).

A **match**

$$\text{MATCH}\ s\ [\ c x_1 \ldots x_n \Rightarrow t \mid \cdots\ ]$$

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** $c x_1 \ldots x_n \Rightarrow t$ must be given, where the variables in the **pattern** $c x_1 \ldots x_n$ 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.

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

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

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

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**

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

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**

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

is called **β-reduction** and replaces an application $(\lambda x.s)t$ with the term s_t^x 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**

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

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**

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

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:

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

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

$$\text{LET } x^{\tau} = s \text{ IN } t$$

providing for local definitions. The reduction rule for let expressions

$$\text{LET } x = s \text{ IN } t \;\succ\; t_s^x$$

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 this text, 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:

$$\text{IF } s \text{ THEN } t_1 \text{ ELSE } t_2 \quad \rightsquigarrow \quad \text{MATCH } s \text{ [} \mathbf{T} \Rightarrow t_1 \mid \mathbf{F} \Rightarrow t_2 \text{]}$$

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:

$$\text{LET } (x, y) = s \text{ IN } t \quad \rightsquigarrow \quad \text{MATCH } s \text{ [} \mathsf{pair}\ \_\ \_\ x\ y \Rightarrow t \text{]}$$

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	$P \wedge Q$	P and Q
disjunction	$P \vee Q$	P or Q
implication	$P \to Q$	if P then Q
negation	$\neg P$	not P
equivalence	$P \longleftrightarrow Q$	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

$$\wedge\,(X:\mathbb{P},\,Y:\mathbb{P}):\mathbb{P}\ ::=\ \mathsf{C}(X,Y) \qquad \vee\,(X:\mathbb{P},\,Y:\mathbb{P}):\mathbb{P}\ ::=\ \mathsf{L}(X)\mid\mathsf{R}(Y)$$

giving us the constructors

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

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

$X \to X$	$\lambda x.x$
$X \to Y \to X$	$\lambda xy.x$
$X \to Y \to Y$	$\lambda xy.y$
$(X \to Y \to Z) \to (Y \to X \to Z)$	$\lambda fyx.fxy$
$X \to Y \to X \wedge Y$	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

accompanied by normal proofs. The propositions formulate familiar logical laws. Note that we supply as subscripts the implicit arguments of the proof constructors 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,

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

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

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

Figure 3.2: Normal proofs with nested patterns

we 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.

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

$$X \longleftrightarrow Y \ :=\ (X \to Y) \wedge (Y \to X)$$

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 $X \longleftrightarrow Y$, 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.

$X \wedge Y \longleftrightarrow Y \wedge X$	$X \vee Y \longleftrightarrow Y \vee X$	*commutativity*
$X \wedge (Y \wedge Z) \longleftrightarrow (X \wedge Y) \wedge Z$	$X \vee (Y \vee Z) \longleftrightarrow (X \vee Y) \vee Z$	*associativity*
$X \wedge (Y \vee Z) \longleftrightarrow X \wedge Y \vee X \wedge Z$	$X \vee (Y \wedge Z) \longleftrightarrow (X \vee Y) \wedge (X \vee Z)$	*distributivity*
$X \wedge (X \vee Y) \longleftrightarrow X$	$X \vee (X \wedge Y) \longleftrightarrow X$	*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:

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

3.5 Truth, Falsity, and Negation

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

$$\top : \mathbb{P} ::= \mathsf{I} \qquad\qquad \bot : \mathbb{P} ::= [\,]$$

giving us the constructors

$$\top : \mathbb{P} \qquad\qquad \bot : \mathbb{P}$$
$$\mathsf{I} : \top$$

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.

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

Figure 3.4: Proofs for propositions with negations

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

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) $\neg\neg\bot \longleftrightarrow \bot$

c) $\neg\neg\top \longleftrightarrow \top$

d) $\neg\neg\neg X \longleftrightarrow \neg X$

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

f) $\neg(X \longleftrightarrow \neg X)$

g) $\neg(X \vee Y) \longleftrightarrow \neg X \wedge \neg Y$

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**:

		$\neg(X \longleftrightarrow \neg X)$	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

$$\lambda h^{X \longleftrightarrow \neg X}.\ \text{MATCH}\ h\ [\ \mathsf{C} f g \Rightarrow \text{LET}\ x = g(\lambda x.fxx)\ \text{IN}\ fxx\ ]$$

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

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.

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

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.

		$\neg\neg(X \to Y) \longleftrightarrow (\neg\neg X \to \neg\neg Y)$	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:

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

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) $\neg(X \vee Y) \longleftrightarrow \neg X \wedge \neg Y$

c) $\neg\neg\neg X \longleftrightarrow \neg X$

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

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

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

3.7 Notational Issues

Following Coq, we use the precedence order

$$\neg \quad \wedge \quad \vee \quad \longleftrightarrow \quad \rightarrow$$

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

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

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

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

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_t^x 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:

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

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

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

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 $\top$.

$$\bot \longleftrightarrow \forall Z^{\mathbb{P}}.\, Z$$
$$\mathsf{C}\ (\lambda h.\ \text{MATCH}\ h\ [])\ (\lambda f.\ f\bot)$$

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

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

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

$$(\forall x \forall y.\, pxy) \longleftrightarrow (\forall y \forall x.\, pxy)$$

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:

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

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: $\lambda X p f x g.\, f(\lambda f'.\, g(f'x))$

Figure 4.2: Proof diagram for a double negation law

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

Proof term: $\lambda X p q f g x.\ \textsc{match}\ fx\ [\ \mathsf{C\_}h \Rightarrow h(gx)\ ]$

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 $\forall X^{\mathbb{T}} \forall p^{X\to\mathbb{P}} \forall q^{X\to\mathbb{P}}.\ (\forall x.\, px \wedge qx) \longleftrightarrow (\forall x.\, px) \wedge (\forall x.\, qx)$.

Exercise 4.3.2 Find out which direction of the equivalence $\forall X^{\mathbb{T}} \forall Z^{\mathbb{P}}.\ (\forall x^X.\, Z) \longleftrightarrow Z$ 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

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

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:

$$\begin{aligned} \neg s &:= s \to \bot \\ s \longleftrightarrow t &:= (s \to t) \wedge (t \to s) \end{aligned}$$

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

$$\begin{aligned} \neg &: \mathbb{P} \to \mathbb{P} := \lambda X.\ X \to \bot \\ \longleftrightarrow &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} := \lambda XY.\ (X \to Y) \wedge (Y \to X) \end{aligned}$$

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 $s \longleftrightarrow t$ and $(s \to t) \wedge (t \to s)$. 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, $\mathsf{N} \times \mathsf{N}$, and $\mathsf{N} \to \mathsf{N}$ 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_t^x if s has type $\forall x:u.v$ and t has type u.

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

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 Propositional Equality as 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:

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

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 **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 $!\mathbf{T} = \mathbf{F}$. Make all implicit arguments explicit and explain which type checking rules are needed to establish that your proof term has type $!\mathbf{T} = \mathbf{F}$. Explain why the same proof term also proves $\mathbf{F} = !!\mathbf{F}$.

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

$\top \neq \bot$	propositional disjointness
$\mathbf{T} \neq \mathbf{F}$	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

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**:

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

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

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

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 $\bot$, we can prove everything (ex falso quodlibet) and our carefully constructed logical system collapses.

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

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

	$\mathbf{T} \neq \mathbf{F}$	intros
$H : \mathbf{T} = \mathbf{F}$	$\bot$	conversion
	$(\lambda x^{\mathsf{B}}.\ \textsc{match}\ x\ [\,\mathbf{T} \Rightarrow \top \mid \mathbf{F} \Rightarrow \bot\,])\,\mathbf{F}$	apply R_H
	$(\lambda x^{\mathsf{B}}.\ \textsc{match}\ x\ [\,\mathbf{T} \Rightarrow \top \mid \mathbf{F} \Rightarrow \bot\,])\,\mathbf{T}$	conversion
	$\top$	I

Proof term: $\lambda H.\,\mathsf{R}\,(\lambda x^{\mathsf{B}}.\ \textsc{match}\ x\ [\,\mathbf{T}\Rightarrow\top\mid\mathbf{F}\Rightarrow\bot\,])\,H\,\mathsf{I}$

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

Proof term: $\lambda x y H.\,\mathsf{R}\,(\lambda z.\ x = \textsc{match}\ z\ [\,0\Rightarrow 0\mid\mathsf{S}z'\Rightarrow z'\,])\,H\,(\mathsf{Q}x)$

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

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

Figure 5.2: Proofs of basic equational facts

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:

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

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.

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

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 $X \wedge Y \longleftrightarrow X \wedge' Y$. 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 proof techniques 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 structural case analysis in the definition of eliminators is often dependently typed using return type functions. Return type functions also appear as constituents of the corresponding matches.

We will also look at eliminators for inductive predicates. Here we will encounter the elim restriction, which constrains structural cases analysis on proofs such that proofs must be returned. We will touch upon special inductive predicates called transfer predicates, which are exempted from the elim restriction.

We will see proofs for three prominent problems: Kaminski's equation, decidability of equality of numbers, and disequality of the types N and B.

6.1 Boolean Elimination

Recall the inductive type of booleans from § 1.1 :

$$\mathsf{B} ::= \mathbf{T} \mid \mathbf{F}$$

We can define a single function that can express all boolean case analysis we need for definitions and proofs. We call this function **boolean eliminator** and define it as follows:

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

First look at the type of E_B. It says that we can prove $\forall x.px$ by proving $p\,\mathbf{T}$ and $p\,\mathbf{F}$. This amounts to a general boolean case analysis since we can choose the **return**

	$\forall x.\ x = \mathbf{T} \vee x = \mathbf{F}$	conversion
	$\forall x.\ (\lambda x.\ x = \mathbf{T} \vee x = \mathbf{F})\ x$	apply $\mathsf{E_B}$
1	$(\lambda x.\ x = \mathbf{T} \vee x = \mathbf{F})\ \mathbf{T}$	conversion
	$\mathbf{T} = \mathbf{T} \vee \mathbf{T} = \mathbf{F}$	trivial
2	$(\lambda x.\ x = \mathbf{T} \vee x = \mathbf{F})\ \mathbf{F}$	conversion
	$\mathbf{F} = \mathbf{T} \vee \mathbf{F} = \mathbf{F}$	trivial

Figure 6.1: Proof diagram for a boolean elimination

type function p freely. We have seen the use of a general return type function before with the replacement constant for propositional equality.

Note that the type of p is $\mathsf{B} \to \mathbb{T}$. Since $\mathbb{P}$ is a subuniverse of $\mathbb{T}$ (write $\mathbb{P} \subseteq \mathbb{T}$), $\mathsf{B} \to \mathbb{P}$ is a subuniverse of $\mathsf{B} \to \mathbb{T}$ (write $(\mathsf{B} \to \mathbb{P}) \subseteq (\mathsf{B} \to \mathbb{T})$). Thus we can use the boolean eliminator for proofs where p is a predicate $\mathsf{B} \to \mathbb{P}$.

Now look at the defining equations of $\mathsf{E_B}$. They are well-typed since the patterns $\mathsf{E_B}\ pab\ \mathbf{T}$ and $\mathsf{E_B}\ pab\ \mathbf{F}$ on the left do have the types $p\ \mathbf{T}$ and $p\ \mathbf{F}$, which they also give to the variables a and b, respectively.

First example

Suppose we want to prove

$$\forall x.\ x = \mathbf{T} \vee x = \mathbf{F}$$

Then we can use the boolean eliminator and obtain the **partial proof term**

$$\mathsf{E_B}\ (\lambda x.\ x = \mathbf{T} \vee x = \mathbf{F})\ \ulcorner \mathbf{T} = \mathbf{T} \vee \mathbf{T} = \mathbf{F} \urcorner\ \ulcorner \mathbf{F} = \mathbf{T} \vee \mathbf{F} = \mathbf{F} \urcorner$$

which poses the subgoals $\ulcorner \mathbf{T} = \mathbf{T} \vee \mathbf{T} = \mathbf{F} \urcorner$ and $\ulcorner \mathbf{F} = \mathbf{T} \vee \mathbf{F} = \mathbf{F} \urcorner$. Note that the subgoals are obtained with conversion. We now use the proof terms $\mathsf{L}(\mathsf{Q}\,\mathbf{T})$ and $\mathsf{R}(\mathsf{Q}\,\mathbf{F})$ for the subgoals and obtain the complete proof term

$$\mathsf{E_B}\ (\lambda x.\ x = \mathbf{T} \vee x = \mathbf{F})\ (\mathsf{L}(\mathsf{Q}\,\mathbf{T}))\ (\mathsf{R}(\mathsf{Q}\,\mathbf{F}))$$

Figure 6.1 shows a proof diagram for the above proof term. The diagram makes explicit the various conversions involving the return type function. That we can model all boolean case analysis with a single eliminator crucially depends on the fact that type checking builds in (through the conversion rule) the conversions handling return type functions.

Second example

Here is a more challenging fact known as **Kaminski's equation**[1] that can be shown with boolean elimination:

$$\forall f^{\mathsf{B}\to\mathsf{B}}\, \forall x.\ f(f(fx)) = fx$$

Obviously, a boolean case analysis on just x does not suffice for a proof. What we need in addition is boolean case analysis on the terms $f\,\mathbf{T}$ and $f\,\mathbf{F}$. To make this possible, we prove the equivalent claim

$$\forall xyz.\ \ f\,\mathbf{T} = y \ \to\ f\,\mathbf{F} = z \ \to\ f(f(fx)) = fx$$

by boolean case analysis on x, y, and z. This gives us 8 subgoals, all of which have straightforward equational proofs. Here is the subgoal for $x = \mathbf{F}$, $y = \mathbf{F}$, and $z = \mathbf{T}$:

$$f\,\mathbf{T} = \mathbf{F} \ \to\ f\,\mathbf{F} = \mathbf{T} = \ \to\ f(f(f\,\mathbf{F})) = f\,\mathbf{F}$$

Matches with return type functions

In Coq, the equational definition of E_B must be carried out with a **dependently typed match** carrying a return type function:

$$\mathsf{E}_\mathsf{B}\ :=\ \lambda pabx.\ \textsc{match}\ x\uparrow p\ [\,\mathbf{T}\Rightarrow a \mid \mathbf{F}\Rightarrow b\,]$$

The return type function p is necessary since the two clauses of the match yield different types ($p\,\mathbf{T}$ and $p\,\mathbf{F}$), which are again different from the return type (px) of the match. A **simply typed match** is a match whose clauses all yield the return type of the match. A simply typed match may be seen as a match with a constant return type function.

We have

$$\mathsf{E}_\mathsf{B}\,pabx\ \approx\ \textsc{match}\ x\uparrow p\ [\,\mathbf{T}\Rightarrow a \mid \mathbf{F}\Rightarrow b\,]$$

Thus a boolean match can be seen as an application of the boolean eliminator.

We recommend taking the view that the eliminator E_B is equationally defined and that boolean matches are a notational convenience for applications of the boolean eliminator. We don't like Coq's design decision to provide primitive matches rather than native inductive function definitions with defining equations.

Exercise 6.1.1 Define boolean negation and boolean conjunction with the boolean eliminator.

[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.

Exercise 6.1.2 For each of the following propositions give a proof term applying the boolean eliminator.

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

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

c) $x \mathbin{\&} y = \mathbf{T} \longleftrightarrow x = \mathbf{T} \wedge y = \mathbf{T}.$

d) $x \mid y = \mathbf{F} \longleftrightarrow x = \mathbf{F} \wedge y = \mathbf{F}.$

6.2 Elimination for Numbers

Recall the inductive type of numbers from § 1.2 :

$$\mathsf{N} \ ::= \ 0 \mid \mathsf{S}(\mathsf{N})$$

Match eliminator for numbers

Suppose we have a constant

$$\mathsf{M}_\mathsf{N} : \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to (\forall n.\, p(\mathsf{S}n)) \to \forall n.\, pn$$

Then we can use M_N to do case analysis on numbers in proofs: To prove $\forall n.\, pn$, we prove a *base case* $p0$ and a *successor case* $\forall n.\, p(\mathsf{S}n)$. Once we add the reduction rules

$$\begin{aligned} \mathsf{M}_\mathsf{N}\, paf\, 0 &\ \succ\ a \\ \mathsf{M}_\mathsf{N}\, paf(\mathsf{S}n) &\ \succ\ fn \end{aligned}$$

we obtain the full power of matches:

$$\mathsf{M}_\mathsf{N}\, paf n \ \approx\ \textsc{match}\ n \uparrow p\ [\, 0 \Rightarrow a \mid \mathsf{S}n \Rightarrow fn\,]$$

Note that the match is given with a return type function p. The return type function is needed so that the match can be type checked. Matches for numbers are type checked following the scheme given by the type of M_N.

Here is the equational definition of M_N :

$$\begin{aligned} \mathsf{M}_\mathsf{N} : \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to (\forall n.\, p(\mathsf{S}n)) \to \forall n.\, pn \\ \mathsf{M}_\mathsf{N}\, paf\, 0 \ :=\ a \qquad &: p0 \\ \mathsf{M}_\mathsf{N}\, paf(\mathsf{S}n) \ :=\ fn \qquad &: p(\mathsf{S}n) \end{aligned}$$

The types of the defining equations as they are determined by type checking their patterns are annotated on the right.

Recursive eliminator for numbers

Look at the type of the match eliminator for numbers:

$$\mathsf{M}_\mathsf{N} : \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to (\forall n.\, p(\mathsf{S}n)) \to \forall n.\, pn$$

The type gives us the structure we need for structural induction on numbers except that the inductive hypothesis is missing. Our informal understanding of inductive proofs suggests that we add the inductive hypothesis as implicational premise to the successor clause:

$$\mathsf{E}_\mathsf{N} : \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to (\forall n.\, pn \to p(\mathsf{S}n)) \to \forall n.\, pn$$

There are two questions now: Can we define a **recursive eliminator** E_N with the given type, and does the type of E_N really suffice to do proofs by structural induction?

The definition of E_N is pleasantly straightforward: We take the defining equations for M_N and obtain the additional argument for the inductive hypothesis of the continuation function f in the successor case with structural recursion:

$$\begin{aligned}
&\mathsf{E}_\mathsf{N} : \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to (\forall n.\, pn \to p(\mathsf{S}n)) \to \forall n.\, pn \\
&\quad \mathsf{E}_\mathsf{N}\, paf\, 0 \ := \ a && : \ p0 \\
&\mathsf{E}_\mathsf{N}\, paf(\mathsf{S}n) \ := \ f\, n\, (\mathsf{E}_\mathsf{N}\, paf n) && : \ p(\mathsf{S}n)
\end{aligned}$$

The type of E_N clarifies many aspects of informal inductive proofs. For instance, the type of E_N makes clear that the variable n in the initial claim $\forall n.\, pn$ is different from the variable n in the successor case $\forall n.\ pn \to p(\mathsf{S}n)$. However, it makes sense to use the same name for both variables since this makes the inductive hypothesis pn agree with the initial claim.

First example

We can now do inductive proofs completely formally. As first example we consider the fact

$$\forall x.\, x + 0 = x$$

We do the proof by induction on n, which amounts to an application of the eliminator E_N:

$$\mathsf{E}_\mathsf{N}\ (\lambda x.\ x + 0 = x)\ \ulcorner 0 + 0 = 0\urcorner\ \ulcorner \forall x.\, x + 0 = x \to \mathsf{S}x + 0 = \mathsf{S}x\urcorner$$

The partial proof term leaves two subgoals known as base case and successor case. Both subgoals have straightforward proofs. Note how the inductive hypothesis appears as am implicational premise in the successor case. Figure 6.2 shows a proof diagram for a proof term completing the partial proof term obtained with E_N.

		$x + 0 = x$	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: Proof diagram for $x + 0 = x$

Second example

Our second example

$$\forall x^{\mathsf{N}}\, y^{\mathsf{N}}.\;\; x = y \vee x \neq y$$

says that **equality of numbers is logically decidable**. To prove this claim we need induction on x and case analysis on y. Moreover, it is essential that y is quantified in the inductive hypothesis. We start with the partial proof term

$$\begin{aligned}
&\mathsf{E_N}\ (\lambda x.\ \forall y.\, x = y \vee x \neq y)\\
&\ulcorner \forall y.\, 0 = y \vee 0 \neq y \urcorner\\
&\ulcorner \forall x.\, (\forall y.\, x = y \vee x \neq y) \to \forall y.\, \mathsf{S}x = y \vee \mathsf{S}x \neq y \urcorner
\end{aligned}$$

The base case follows with case analysis on y:

$$\begin{aligned}
&\mathsf{M_N}\ (\lambda y.\, 0 = y \vee 0 \neq y)\\
&\ulcorner 0 = 0 \vee 0 \neq 0 \urcorner\\
&\ulcorner \forall y.\, 0 = \mathsf{S}y \vee 0 \neq \mathsf{S}y \urcorner
\end{aligned}$$

The first subgoal is trivial, and the second subgoal follows with constructor disjointness. The successor case also needs case analysis on y:

$$\begin{aligned}
\lambda x h^{\forall y.\ x=y \vee x \neq y}.\ \ &\mathsf{M_N}\ (\lambda y.\, x = y \vee x \neq y)\\
&\ulcorner \mathsf{S}x = 0 \vee \mathsf{S}x \neq 0 \urcorner\\
&\ulcorner \forall y.\, \mathsf{S}x = \mathsf{S}y \vee \mathsf{S}x \neq \mathsf{S}y \urcorner
\end{aligned}$$

The first subgoal follows with constructor disjointness. The second subgoal follows with the instantiated inductive hypothesis hy and injectivity of S.

Figure 6.3 shows a proof diagram for the partial proof term developed above.

		$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \vee x \neq y$	apply E_{N}, intros
1		$0 = y \vee 0 \neq y$	destruct y
1.1		$0 = 0 \vee 0 \neq 0$	trivial
1.2		$0 = \mathsf{S}y \vee 0 \neq \mathsf{S}y$	trivial
2	IH: $\forall y^{\mathsf{N}}.\ x = y \vee x \neq y$	$\mathsf{S}x = y \vee \mathsf{S}x = y$	destruct y
2.1		$\mathsf{S}x = 0 \vee \mathsf{S}x \neq 0$	trivial
2.2		$\mathsf{S}x = \mathsf{S}y \vee \mathsf{S}x \neq \mathsf{S}y$	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

We have described to above proof with much formal detail. This was done so that the reader understands that inductive proofs can be formalized with only a few basic type-theoretic principles. If we do the proof with a proof assistant, a fully formal proof is constructed but most of the details are taken care of by automation. If we want to document the proof informally for a human reader, we may just write something like the following:

> *The claim follows by induction on x and case analysis on y, where y is quantified in the inductive hypothesis and disjointness and injectivity of the constructors 0 and S are used.*

Exercise 6.2.1 Define M_{N} with E_{N}.

Exercise 6.2.2 Define a function $A : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ for addition using E_{N} and prove $Axy = x + y$ using E_{N}.

Exercise 6.2.3 Define a function $M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ for truncating subtraction using E_{N} and M_{N}. Prove $Mxy = x - y$ using E_{N} and M_{N}.

Exercise 6.2.4 Prove the following propositions in Coq using E_{N} and M_{N}.

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

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

c) $x + y = x + z \to y = z$ (addition is injective in its 2nd argument)

Also write high-level proof diagrams in the style of Chapter 1.

Exercise 6.2.5 (Boolean equality decider for numbers)
Write a function $\mathsf{eq}_{\mathsf{N}} : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$ such that $\forall xy.\ x = y \longleftrightarrow \mathsf{eq}_{\mathsf{N}}\, xy = \mathbf{T}$.
Prove the equivalence.

6.3 Eliminator for Pairs

Following the scheme we have seen for booleans and numbers, we can define an eliminator for pairs (Section 1.6):

$$\mathsf{E}_\times : \ \forall X^{\mathbb{T}} Y^{\mathbb{T}} \forall p^{X\times Y\to\mathbb{T}}.\ (\forall xy.\ p(x,y)) \to \forall a.pa$$
$$\mathsf{E}_\times\, XYpf\,(x,y) \ :=\ fxy$$

Exercise 6.3.1 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 6.3.2 Use $\mathsf{E}_\times$ to write terms that are computationally equal to π_1, π_2, and swap (see Section 1.6).

Exercise 6.3.3 By now you know enough to do all proofs of Chapter 1 with proof terms. Do some of the proofs in Coq without using the tactics for destructuring and induction. Use the eliminators you have seen in this chapter instead.

6.4 Elim Restriction and Transfer Predicates

A structural case analysis on the proofs of a proposition obtained with an inductive predicate is restricted to yield a proof, except the inductive predicate is a so-called transfer predicate. We refer to this important feature of Coq's type theory as **elim restriction**. There are important reasons for imposing the elim restriction that we will explain later.

A **transfer predicate** is an inductive predicate that has either no proof constructor, or has a single proof constructor having the additional property that all its non-parametric arguments are proofs. We will see in later chapters that transfer predicates provide essential features in Coq's type theory.

We may define eliminators for the transfer predicates $\bot$ and $\top$ as follows:

$$\mathsf{E}_\bot : \ \forall Z^{\mathbb{T}}.\ \bot \to Z$$

$$\mathsf{E}_\top : \ \forall p^{\top\to\mathbb{T}}.\ p\,\mathsf{I} \to \forall x.\, px$$
$$\mathsf{E}_\top\, pa\,\mathsf{I} \ :=\ a \qquad : p\,\mathsf{I}$$

Note that there is no defining equation for $\mathsf{E}_\bot$ since $\bot$ has no proof constructor. Also note that $\mathbb{T}$ would have to be replaced by $\mathbb{P}$ in the types of $\mathsf{E}_\bot$ and $\mathsf{E}_\top$ if the elim restriction would apply to $\bot$ and $\top$.

For the inductive predicates conjunction and disjunction we define the following eliminators:

$$\begin{aligned}
&\mathsf{E}_\wedge : \forall Z^{\mathbb{T}}.\ (X \to Y \to Z) \to X \wedge Y \to Z\\
&\mathsf{E}_\wedge\ Z f\ (\mathsf{C}\, x\, y) \ :=\ f x y
\end{aligned}$$

$$\begin{aligned}
&\mathsf{E}_\vee : \forall Z^{\mathbb{P}}.\ (X \to Z) \to (Y \to Z) \to X \vee Y \to Z\\
&\mathsf{E}_\vee\ Z f g\ (\mathsf{L}\, x) \ :=\ f x\\
&\mathsf{E}_\vee\ Z f g\ (\mathsf{R}\, y) \ :=\ g y
\end{aligned}$$

Note that Z can be any type in the type of $\mathsf{E}_\wedge$, but is restricted to propositions in the type of $\mathsf{E}_\vee$. This is because $\wedge$ is a transfer predicate but $\vee$ is not.

It is common language to refer to a structural case analysis as an elimination. We speak of a **computational elimination** if the result of the elimination is not a proof, and of a **propositional elimination** if the result of the elimination is a proof.

The elim restriction is a device that disallows computational eliminations for most inductive predicates. Computational eliminations on transfer predicates with a single proof constructor are often referred to as **singleton eliminations**. There is the intuition that singleton eliminations are admissible since they don't leak equality of proofs to equality of non-proofs.

The elim restriction is the price we pay so that assuming the law of excluded middle $\forall X^{\mathbb{P}}.\ X \vee \neg X$ does not enable a proof of falsity and thus remains logically meaningful. We will see in a later chapter that propositions have at most one proof (under propositional equality) if the law of excluded middle is assumed. We remark that even without the law of excluded middle the impredicativity of the universe $\mathbb{P}$ of propositions would enable a proof of falsity. We may say that we get the impredicativity of $\mathbb{P}$ as a free extra once we impose the elim restriction.

In Chapter 3 we have seen many propositional eliminations for $\bot$, $\wedge$, and $\vee$. The matches used there were all simply typed and relate to the eliminators $\mathsf{E}_\bot$, $\mathsf{E}_\wedge$, and $\mathsf{E}_\vee$ as follows:

$$\begin{aligned}
\mathsf{E}_\bot\ Z h &\ \approx\ \text{MATCH}\ h\ [\,]\\
\mathsf{E}_\wedge\ Z f h &\ \approx\ \text{MATCH}\ h\ [\,\mathsf{C}\, x\, y \Rightarrow f x y\,]\\
\mathsf{E}_\vee\ Z f g h &\ \approx\ \text{MATCH}\ h\ [\,\mathsf{L}\, x \Rightarrow f x \mid \mathsf{R}\, y \Rightarrow g y\,]
\end{aligned}$$

Exercise 6.4.1 Prove $\forall x^{\top} \forall y^{\top}.\ x = y$ using $\mathsf{E}_\top$.

Exercise 6.4.2 Define an eliminator $\forall X^{\mathbb{P}} Y^{\mathbb{P}} \forall p^{X \wedge Y \to \mathbb{T}}.\ (\forall x y.\ p(\mathsf{C}\, x\, y)) \to \forall a.\, p a$ for conjunction.

6.5 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 8.1.3 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

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

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

$$0 = 1 \vee 0 = 2 \vee 1 = 2$$

which can be disproved with disjunctive elimination and disjointness and injectivity of 0 and S.

Fact 6.5.1 $\mathsf{N} \neq \mathsf{B}$.

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 6.5.2 Prove $\mathsf{B} \neq \top$ and $\mathsf{B} \neq \mathsf{B} \times \mathsf{B}$.

Exercise 6.5.3 Prove $\mathsf{B} \neq \mathbb{T}$.

Exercise 6.5.4 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 $\mathsf{B} = \mathsf{B} \times \top$.

6.6 Abstract Return Types

Eliminators have *abstract return types* providing great flexibility. Two typical examples are

$$\mathsf{E}_\bot :\ \forall Z^{\mathbb{T}}.\ \bot \to Z$$
$$\mathsf{E}_\mathsf{B} :\ \forall p^{\mathsf{B}\to\mathbb{T}}.\ \ p\,\mathbf{T} \to p\,\mathbf{F} \to \forall x.px$$

The point is that Z and px may be arbitrary types. This means in particular that eliminators are functions that are polymorphic in the number of their arguments. For instance:

$$\mathsf{E}_\perp\,\mathsf{N} : \perp \to \mathsf{N}$$
$$\mathsf{E}_\perp\,(\mathsf{N} \to \mathsf{N}) : \perp \to \mathsf{N} \to \mathsf{N}$$
$$\mathsf{E}_\perp\,(\mathsf{N} \to \mathsf{N} \to \mathsf{N}) : \perp \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$

7 Case Study: Pairing Function

Cantor discovered that numbers are in bijection with pairs of numbers. Cantor's proof rests on a counting scheme where pairs appear as points in the plane. Based on Cantors scheme, we realize the bijection between numbers and pairs with two functions inverting each other. We obtain an elegant formal development using only a few basic facts about numbers.

7.1 Definitions

We will construct and verify two functions

$$\begin{array}{ll} E : \mathsf{N} \times \mathsf{N} \to \mathsf{N} & \textit{encode} \\ D : \mathsf{N} \to \mathsf{N} \times \mathsf{N} & \textit{decode} \end{array}$$

that invert each other: $D(E(x,y)) = (x,y)$ and $E(Dn)) = n$. The functions are based on the counting scheme for pairs shown in Figure 7.1. The pairs appear as points in the plane following the usual coordinate representation. Counting starts at the origin $(0,0)$ and follows the diagonals from right to left:

$(0,0)$	1st diagonal	0
$(1,0),\ (0,1)$	2nd diagonal	$1,2$
$(2,0),\ (1,1),\ (0,2)$	3rd diagonal	$3,4,5$

Assuming a function

$$\eta : \mathsf{N} \times \mathsf{N} \to \mathsf{N} \times \mathsf{N}$$

that for every pair yields its successor on the diagonal walk described by the counting scheme, we define the decoding function D as follows:

$$D(n) := \eta^n(0,0)$$

The definition of the successor function η for pairs is also straightforward:

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

y	$\vdots$						
5	20						
4	14	19					
3	9	13	18				
2	5	8	12	17			
1	2	4	7	11	16		
0	0	1	3	6	10	15	$\cdots$
	0	1	2	3	4	5	x

Figure 7.1: Counting scheme for pairs of numbers

We now come to the definition of the encoding function E. We first observe that all pairs (x, y) on a diagonal have the same sum $x + y$, and that the length of the nth diagonal is n. We start with the equation

$$E(x, y) := \sigma(x + y) + y$$

where $\sigma(x + y)$ is the first number on the diagonal $x + y$. We now observe that

$$\sigma n = 0 + 1 + 2 + \cdots + n$$

Thus we define σ recursively as follows:

$$\begin{aligned} \sigma(0) &:= 0 \\ \sigma(\mathsf{S}n) &:= \mathsf{S}n + \sigma n \end{aligned}$$

We remark that σn is known as Gaussian sum.

7.2 Proofs

We start with a useful equation saying that under the encoding function successors of pairs agree with successors of numbers.

Fact 7.2.1 (Successor equation) $E(\eta c) = \mathsf{S}(Ec)$ for all pairs c.

Proof Case analysis on $c = (0, y)$, $(\mathsf{S}x, y)$ and straightforward arithmetic. ∎

Fact 7.2.2 $E(Dn) = n$ for all numbers n.

Proof By induction on n using Fact 7.2.1 for the successor case. ∎

Fact 7.2.3 $D(Ec) = c$ for all pairs c.

Proof Given the recursive definition of D and E, we need to do an inductive proof. The idea is to do induction on the number Ec. Formally, we prove the proposition

$$\forall c.\ Ec = n \to Dn = c$$

by induction on n.

For $n = 0$ the premise gives us $c = (0,0)$ making the conclusion trivial.

For the successor case we prove

$$Ec = \mathsf{S}n \to D(\mathsf{S}n) = c$$

We consider three cases: $c = (0,0)$, $(\mathsf{S}x, 0)$, $(x, \mathsf{S}y)$. The case $c = (0,0)$ is trivial since the premise is contradictory. The second and third case are similar. We show the third case

$$E(x, \mathsf{S}y) = \mathsf{S}n \to D(\mathsf{S}n) = (x, \mathsf{S}y)$$

We have $\eta(\mathsf{S}x, y) = (x, \mathsf{S}y)$, hence using Fact 7.2.1 and the definition of D it suffices to show

$$\mathsf{S}(E(\mathsf{S}x, y)) = \mathsf{S}n \to \eta(Dn) = \eta(\mathsf{S}x, y)$$

The premise yields $E(\mathsf{S}x, y) = n$, thus $Dn = (\mathsf{S}x, y)$ by the inductive hypothesis. ∎

Exercise 7.2.4 A bijection between two types X and Y consists of two functions $f : X \to Y$ and $g : Y \to X$ such that $\forall x.\ g(fx) = x$ and $\forall y.\ f(gy) = y$.

a) Give and verify a bijection between N and $(\mathsf{N} \times \mathsf{N}) \times \mathsf{N}$.

b) Prove that there is no bijection between B and $\top$.

7.3 Discussion

Technically, the most intriguing point of the development is the implicational inductive lemma used in the proof of Fact 7.2.3 and the accompanying insertion of η-applications (idea due to Andrej Dudenhefner, March 2020). Realizing the development with Coq is pleasant, with the exception of the proof of the successor equation (Fact 7.2.1), where Coq's otherwise powerful tactic for linear arithmetic fails since it cannot look into the recursive definition of σ.

What I like about the development of the pairing function is the interesting interplay between geometric speak (e.g., diagonals) and formal definitions and proofs. Their is great elegance at all levels. Cantor's pairing function is a great example for an educated Programming 1 course addressing functional programming and program verification.

It is interesting to look up Cantor's pairing function in the mathematical literature and in Wikipedia, where the computational aspects of the construction are

ignored as much as possible. There one typically starts with the encoding function and uses the Gaussian sum formula to avoid the recursion. Then injectivity and surjectivity of the encoding function are shown, which non-constructively yields the existence of the decoding function. The simple recursive definition of the decoding function does not appear.

8 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.

8.1 Inductive Definition and Basic Facts

Following the design laid out above, we accommodate existential quantification with the inductive definition

$$\mathsf{ex}\,(X : \mathbb{T},\, p : X \to \mathbb{P}) : \mathbb{P} \ ::= \ \mathsf{E}\,(x : X,\, px)$$

providing the constructors

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

	$X:\mathbb{T}$, $p:X\to\mathbb{P}$	$\neg(\exists x.px) \longleftrightarrow \forall x.\neg px$	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
		$\mathsf{ex}\,p$	$\mathsf{E}\,x\,a$
2		$(\forall x.\neg px) \to \neg(\exists x.px)$	intros
	$f:\forall x.\neg px$, $x:X$, $a:px$	$\bot$	$f\,x\,a$

Proof term: $\mathsf{C}\ (\lambda f x a.f(\mathsf{E}_p x a))\ (\lambda f h.\ \text{MATCH}\ h\ [\,\mathsf{E}\,x\,a \Rightarrow f\,x\,a\,])$

Figure 8.1: Proof of existential de Morgan law

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

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

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

Note that the elim restriction applies to the inductive predicate ex since the witness argument $x:X$ of the proof constructor E is not a proof. Thus, we cannot extract the witness from a proof of $\exists x.s$ in a computational context. We define an eliminator for existential quantification that suffices for all eliminations in this chapter:

$$\mathsf{E}_\exists :\ \forall X^{\mathbb{T}}\forall q^{X\to\mathbb{P}}\forall Z^{\mathbb{P}}.\ (\forall x.\ qx\to Z)\to \mathsf{ex}\,q\to Z$$
$$\mathsf{E}_\exists\,X\,q\,Z\,f\,(\mathsf{E}\,x\,h)\ :=\ f\,x\,h$$

Figure 8.1 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 8.1.1 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) $(\exists x.\ px\vee qx)\ \longleftrightarrow\ (\exists x.px)\vee(\exists x.qx)$.

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

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

e) $\neg\neg(\exists x.px)\ \longleftrightarrow\ \neg\forall x.\neg px$.

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

Exercise 8.1.2 Prove $\forall X^{\mathbb{P}}.\ X \longleftrightarrow \exists x{:}X.\top$.

Exercise 8.1.3 Verify the following existential characterization of disequality:

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

Exercise 8.1.4 Verify the impredicative characterization of existential quantification:

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

Exercise 8.1.5 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 8.1.6 Universal and existential quantification are compatible with propositional equivalence. Prove the following compatibility laws:

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

Exercise 8.1.7 Prove $\forall X^{\mathbb{T}}\forall p^{X\to\mathbb{P}}\forall Z^{\mathbb{P}}.\ (\exists x.px) \wedge Z \longleftrightarrow \exists x.\ px \wedge Z$.

8.2 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: $\neg\exists x\,\forall y.\ y \in x \longleftrightarrow y \notin y$.
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: $\neg\exists x\,\forall y.\ Hxy \longleftrightarrow \neg Hyy$. 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 8.2.1 (Barber Theorem) Let X be a type and p be a binary predicate on X. Then $\neg\exists x\,\forall y.\ pxy \longleftrightarrow \neg pyy$.

Proof Suppose there is an x such that $\forall y.\ pxy \longleftrightarrow \neg pyy$. Then $pxx \longleftrightarrow \neg pxx$. Contradiction by Russell's law $\neg(X \longleftrightarrow \neg X)$ 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 8.2.2 Give a proof diagram and a proof term for the barber theorem. Construct a detailed proof with Coq.

8.3 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 8.3.1 Boolean negation has no fixed point.

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

Fact 8.3.2 Propositional negation $\lambda P.\neg P$ has no fixed point.

Proof Suppose $\neg P = P$. Then $\neg P \longleftrightarrow P$. Contradiction with Russell's law. ∎

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

Theorem 8.3.3 (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 8.3.4 There is no surjective function $X \to (X \to \mathsf{B})$.

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

Corollary 8.3.5 There is no surjective function $X \to (X \to \mathbb{P})$.

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

We remark that Corollaries 8.3.4 and 8.3.5 may be seen as variants of Cantor's theorem.

Exercise 8.3.6 Construct with Coq detailed proofs of the results in this section.

Exercise 8.3.7 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 8.3.8 Show that every function $\top \to \top$ has a fixed point.

Exercise 8.3.9 With Lawvere's theorem we can give another proof of Fact 8.3.2 (propositional negation has no fixed point). In contrast to the proof given with Fact 8.3.2, 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.

9 Recursive Specification of Functions

Coq's type theory limits recursion in the definition of functions to structural recursion on inductive types. This way, termination of computation can be ensured. We can still write specifications with unconstrained recursion and ask whether a function satisfying the specification can be defined. Often, a function satisfying a recursive specification can be constructed although the recursion in the specification is not structural. Specifications with unconstrained recursion can be elegantly expressed as higher-order functions taking a continuation function as argument. In this chapter, we will look at a few example specifications and discuss the notion of functional extensionality. General techniques for constructing functions satisfying specifications with terminating recursion will be studied in later chapters.

9.1 Step Functions as Specifications

Consider the following equations

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

for a function $f : \mathsf{N} \to \mathsf{B}$. We would hope that we can define a function f satisfying the equations, and that all functions satisfying the equations agree on all numbers. Also we would like to prove that every function satisfying the equations yields $\mathbf{T}$ for exactly those arguments that are even numbers.

As it comes to defining a function f satisfying the equations, we notice that the equations qualify for a definition except for the fact that the recursion is not strictly structural. We aim at defining a function satisfying the equations using strict structural recursion only.

As a first and essential step we define a function

$$\begin{aligned} \mathsf{Even} :\ & (\mathsf{N} \to \mathsf{B}) \to \mathsf{N} \to \mathsf{B} \\ \mathsf{Even}\, f\, 0 &:= \mathbf{T} \\ \mathsf{Even}\, f\, 1 &:= \mathbf{F} \\ \mathsf{Even}\, f\, (\mathsf{S}(\mathsf{S}n)) &:= f\, n \end{aligned}$$

capturing the above equations except for the fact that the recursion is modeled with a **continuation function** f taking the type of the specified function. We will refer to Even as a **step function**.

We now note that a function f satisfies the above equations if and only if it satisfies the **step equation**

$$\forall n.\ f\,n = \mathsf{Even}\,f\,n$$

Moreover, if we have a function f satisfying the step equation, we can carry out abstract computations by rewriting with the step equation:

$$f\,5 = \mathsf{Even}\,f\,5 = f\,3 = \mathsf{Even}\,f\,3 = f\,1 = \mathsf{Even}\,f\,1 = \mathbf{F}$$

Abstract definition of step functions

Step functions can serve as recursive specifications of functions $X \to Y$ in general. Let us spell out the necessary definitions. A **step function** is a function

$$F : (X \to Y) \to X \to Y$$

and a function $f : X \to Y$ **satisfies** a step function F if it satisfies the **step equation**

$$\forall x.\ f\,x = F\,f\,x$$

The first argument of a step function is referred to as **continuation function**. Moreover, we say that a step function F is **unique** if any two functions satisfying F agree on all arguments:

$$\mathsf{unique}\,F \ :=\ \forall f\,g.\ (\forall x.\ f\,x = F\,f\,x) \to (\forall x.\ g\,x = F\,g\,x) \to \forall x.\ f\,x = g\,x$$

Expressing recursive specifications of functions with step functions turns out to be a good choice. Step functions provide the full definitional power of the underlying type theory and add unconstrained recursion for the specified function. Moreover, rewriting with the step equation provides an abstract form of computation. Step functions also make precise which systems of equations we admit as specifications. In fact, step functions admit equational specifications as they are provided by the type theory but remove the constraints on recursion.

Satisfiability and Uniqueness of Even

Having completed the abstract definitions, we now return to the concrete step function Even and define a satisfying function even using structural recursion. The trick is to flip booleans with boolean negation '!':

$$\begin{aligned}
&\mathsf{even} : \mathsf{N} \to \mathsf{B}\\
&\mathsf{even}(0) := \mathbf{T}\\
&\mathsf{even}(\mathsf{S}n) := {!}\,\mathsf{even}(n)
\end{aligned}$$

We now define a function even satisfying the step function Even using strict structural recursion. The trick is to use boolean negation ! to flip booleans:

$$\begin{aligned} &\mathsf{even} : \mathsf{N} \to \mathsf{B} \\ &\quad \mathsf{even}(0) := \mathbf{T} \\ &\mathsf{even}(\mathsf{S}n) := !\,\mathsf{even}(n) \end{aligned}$$

Fact 9.1.1 (Satisfiability) $\forall n.\ \mathsf{even}\, n = \mathsf{Even}\, \mathsf{even}\, n$

Proof We consider the cases $n = 0$, $\mathsf{S}0$, $\mathsf{S}(\mathsf{S}n)$, which give us the three equations of the informal specification as proof obligations. Satisfaction of the first two equations follows by computational equality, and satisfaction of the third equation follows by computational equality and the double negation law for boolean negation: $\mathsf{even}(\mathsf{S}(\mathsf{S}n)) = !\,\mathsf{even}(\mathsf{S}n) = !!\,\mathsf{even}(n) = \mathsf{even}(n)$. ∎

Fact 9.1.2 (Uniqueness) Even is unique.

Proof Let f and g satisfy Even. We prove

$$\forall n.\ fn = gn \wedge f(\mathsf{S}n) = g(\mathsf{S}n)$$

by induction on n. The base case follows with the specifying equations for 0 and 1. For the successor case we prove $f(\mathsf{S}n) = g(\mathsf{S}n)$ and $f(\mathsf{S}(\mathsf{S}n)) = g(\mathsf{S}(\mathsf{S}n))$. The first equation is exactly the second inductive hypothesis. The second equation follows with the third specifying equation and the first inductive hypothesis. ∎

Corollary 9.1.3 All functions satisfying Even agree with even.

Proof Follows with Fact 9.1.1 and uniqueness. ∎

Exercise 9.1.4 Prove $\mathsf{even}\, n = \mathbf{T} \longleftrightarrow \exists k.\ n = 2 \cdot k$.

Exercise 9.1.5 Show that the following step function is unique and satisfiable:

$$\begin{aligned} D &: (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\ Df0 &:= 0 \\ Df(\mathsf{S}n) &:= \mathsf{S}(\mathsf{S}(fn)) \end{aligned}$$

Exercise 9.1.6 Show that the following recursive specification is unique and satisfiable:

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

Exercise 9.1.7 (Hardt's identity) Show that the step function

$$\begin{aligned} H &: (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\ Hf0 &:= 0 \\ Hf(\mathsf{S}n) &:= \mathsf{S}(f(fn)) \end{aligned}$$

is unique and satisfiable. Hint: Prove $\forall n.\ fn = n$ for every function satisfying H.

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 9.1.8 Consider the tail-recursive specification

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

a) Write a cascaded step function $\mathsf{Add} : (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ for the specification.

b) Write a cartesian step function $\mathsf{Add}' : (\mathsf{N} \times \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \times \mathsf{N} \to \mathsf{N}$ for the specification.

c) Prove that $\lambda xy.\ x + y$ satisfies the step function Add.

d) Prove that every function satisfying the step function Add agrees with the standard addition function: $(\forall xy.\ fxy = \mathsf{Add}\, fxy) \to \forall xy.\ fxy = x + y$.

9.2 Fibonacci Numbers

The sequence of **Fibonacci numbers**

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

is obtained by starting with 0 and 1 and proceeding by repeatedly taking the sum of the two preceding numbers. The step function

$$\begin{aligned} \mathsf{Fib} &: (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\ \mathsf{Fib}\, f\, 0 &:= 0 \\ \mathsf{Fib}\, f\, 1 &:= 1 \\ \mathsf{Fib}\, f\, (\mathsf{S}(\mathsf{S}n)) &:= fn + f(\mathsf{S}n) \end{aligned}$$

formulates the iterative scheme behind the sequence with functional recursion. A function satisfying Fib will yield for n the n-th Fibonacci number. We call functions satisfying the step function Fib **Fibonacci functions**.

Following the iterative intuition behind the Fibonacci sequence, it is not difficult to define a Fibonacci function with strict structural recursion. The trick is well-known to functional programmers and consists in incrementing the initial pair $(0,1)$ of successive Fibonacci numbers using the function $(a,b) \mapsto (b,a+b)$ until the n-th Fibonacci number is reached:

$$\begin{aligned} &F : \mathsf{N} \to \mathsf{N} && F' : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &Fn \;:=\; F'\,0\,1\,n && F'\,a\,b\,0 \;:=\; a \\ & && F'\,a\,b\,(\mathsf{S}n) \;:=\; F'\,b\,(a+b)\,n \end{aligned}$$

The example computation

$$F5 = F'\,0\,1\,5 = F'\,1\,1\,4 = F'\,1\,2\,3 = F'\,2\,3\,2 = F'\,3\,5\,1 = F'\,5\,8\,0 = 5$$

shows how the tail recursive function F' realizes the iterative computation

$$(0,1) \;\mapsto\; (1,1) \;\mapsto\; (1,2) \;\mapsto\; (2,3) \;\mapsto\; (3,5) \;\mapsto\; (5,8)$$

We will prove that F satisfies the step function Fib. The first and the second equation follow by computational equality. The third equation

$$F'\,0\,1\,(\mathsf{S}(\mathsf{S}n)) = F'\,0\,1\,n + F'\,0\,1\,(\mathsf{S}n)$$

follows by induction on n provided 0 and 1 are generalized to a and b.

Lemma 9.2.1 $F'ab(\mathsf{S}(\mathsf{S}n)) = F'abn + F'ab(\mathsf{S}n)$.

Proof By induction on n quantifying a and b. The base case follows by computational equality. For the successor case, we have

$$\begin{aligned} F'ab(\mathsf{S}(\mathsf{S}(\mathsf{S}n))) &= F'b(a+b)(\mathsf{S}(\mathsf{S}n)) && \text{conversion} \\ &= F'b(a+b)n + F'b(a+b)(\mathsf{S}n) && \text{inductive hypothesis} \\ &= F'ab(\mathsf{S}n) + F'ab(\mathsf{S}(\mathsf{S}n)) && \text{conversion} \end{aligned}$$

∎

Fact 9.2.2 $F(\mathsf{S}(\mathsf{S}n)) = Fn + F(\mathsf{S}n)$.

Proof Immediate with Lemma 9.2.1. ∎

Fact 9.2.3 (Satisfiability) F satisfies Fib.

Proof We prove $Fn = \mathsf{Fib}\,Fn$ with the case analysis $n = 0,\ 1,\ \mathsf{S}(\mathsf{S}n))$. The third case follows with Fact 9.2.2. ∎

The uniqueness proof for fib is analogous to the uniqueness proof for Even (Fact 9.1.2).

Fact 9.2.4 (Uniqueness) Fib is unique.

Proof Let f and g satisfy Fib. We prove

$$\forall n.\, fn = gn \wedge f(\mathsf{S}n) = g(\mathsf{S}n)$$

by induction on n. The base case follows with the specifying equations for 0 and 1. For the successor case we prove $f(\mathsf{S}n) = g(\mathsf{S}n)$ and $f(\mathsf{S}(\mathsf{S}n)) = g(\mathsf{S}(\mathsf{S}n))$. The first equation is exactly the second inductive hypothesis. The second equation follows with the third specifying equation and the two inductive hypotheses. ∎

Corollary 9.2.5 All functions satisfying Fib agree with F.

Proof Follows with Fact 9.2.3 and uniqueness. ∎

Exercise 9.2.6 Prove the following induction lemma for numbers:

$$\forall p^{\mathsf{N}\to\mathbb{T}}.\, p0 \to p1 \to (\forall n.\, pn \to p(\mathsf{S}n) \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.pn$$

Hint: Prove the strengthened conclusion $pn \times p(\mathsf{S}n)$. Note that the lemma follows the recursive structure of the Fibonacci specification. Use the lemma to prove uniqueness of the step functions Even and Fib.

9.3 Functional Extensionality

Functional extensionality is a proposition saying that functions are equal if they agree on all arguments:

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

Functional extensionality is not provable in Coq's type theory but may be assumed consistently. This means in that in Coq's type theory no proposition contradicting functional extensionality can be shown. For instance, one cannot prove $f \neq g$ for two functions f and g that agree on all arguments.

Assuming functional extensionality determines equality for many functions where without functional extensionality equality was not determined (i.e., one can prove neither $f = g$ nor $f \neq g$).

In set theory-based mathematics functional extensionality is a basic fact. In set-theory, extensionality of functions follows from extensionality of sets (sets are equal if they have the same arguments) since functions are obtained as sets of pairs.

It is generally believed that in type theories functional extensionality is beneficial for the way we think about functions.

Functionality extensionality has many nice consequences that are familiar from set-theoretic mathematics. For instance, assuming functional extensionality, a function f satisfies a step function F if and only if f is a fixed point of F (i.e., $Ff = f$). It has been popular to characterize recursive functions as fixed points of their step functions. We remark that recursive functions are often called fixed points in Coq slang.

Exercise 9.3.1 (Fixed point characterization)
Let $F : (X \to Y) \to X \to Y$ and $f : X \to Y$. Convince yourself that functional extensionality implies $(\forall x.\, fx = Ffx) \longleftrightarrow Ff = f$.

9.4 Ackermann Function

The cascaded step function

$$\begin{aligned}
&\mathsf{Ack} : (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}\\
&\mathsf{Ack}\, f\, 0\, y = \mathsf{S}y\\
&\mathsf{Ack}\, f\, (\mathsf{S}x)\, 0 = f\, x\, 1\\
&\mathsf{Ack}\, f\, (\mathsf{S}x)\, (\mathsf{S}y) = f\, x\, (f\, (\mathsf{S}x)\, y)
\end{aligned}$$

formulates a recursive specification that is satisfied by a variant of the prominent Ackermann function (a set-theoretic function). It turns out that the specified recursion can be seen as a structural recursion on the first argument with a nested structural recursion on the second argument. The nested recursion can be encapsulated into a higher-order auxiliary function A' that is called by the main function A:

$$\begin{aligned}
A &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} & A' &: (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N}\\
A\,0 &:= \mathsf{S} & A'h\,0 &:= h\,1\\
A(\mathsf{S}x) &:= A'(Ax) & A'h\,(\mathsf{S}y) &:= h(A'hy)
\end{aligned}$$

Fact 9.4.1 A satisfies Ack.

Proof Case analysis $(x, y) = (0, y)$, $(\mathsf{S}x, 0)$, $(\mathsf{S}x, \mathsf{S}y)$ and computational equality. ∎

Given that the recursion in the specification is structural and nested, uniqueness of the specification can be shown with nested structural induction.

Fact 9.4.2 Ack is unique.

Proof Let f and g satisfy Ack. We prove

$$\forall y.\, fxy = gxy$$

by structural induction on x. The base case $f0y = g0y$ follows with the step equations for f and g. For the successor case

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

we do structural induction on y. After rewriting with the step equations we have $fx1 = gx1$ as base case and $fx(f(\mathsf{S}x)y) = gx(g(\mathsf{S}x)y)$ as successor case. The base case follows with the inductive hypothesis for x. The successor case follows by the inductive hypothesis for y and the inductive hypothesis for x. ∎

The uniqueness proof for Ack is a nice example of a **nested induction** where both the outer and the inner inductive hypothesis are used in the inner successor case. Note that nested inductions come for free in the general setup we are in. While the details of the nested induction in the above proof might be challenging for a beginner on paper, the actual simplicity of the argument becomes apparent when the proof is done with a proof assistant.

Exercise 9.4.3 Prove the following induction principle mimicking the recursive structure of the step function Ack.

$$\begin{aligned}\forall p^{\mathsf{N}\to\mathsf{N}\to\mathbb{T}}.\ & (\forall y.\, p0y) \to \\ & (\forall x.\, px1 \to p(\mathsf{S}x)0) \to \\ & (\forall xy.\, (\forall z.\, pxz) \to p(\mathsf{S}x)y \to p(\mathsf{S}x)(\mathsf{S}y)) \to \\ & \forall xy.\, pxy\end{aligned}$$

Use the induction principles to prove the uniqueness of Ack.

9.5 Summary

We have seen that recursive specifications of functions can be formalized with step functions. Step functions follow the specifying equations but eliminate the recursion by taking a continuation function as argument. We looked at three examples for recursive specifications and in each case showed satisfiability and uniqueness. The proofs turn out to be interesting.

As it comes to existence, the following ideas lead to definitions satisfying the specifications:

- Use boolean negation to define an eveness test with structural recursion.

- Use a bottom-up scheme with two auxiliary arguments (predecessor of predecessor and predecessor) to compute Fibonacci numbers with structural recursion. The bottom-up scheme yields a tail-recursive function that may be realized with a loop.
- Use higher-order nested recursion to compute the Ackermann function.

To show the correctness of the defined functions, one has to verify the specifying equations. For Ackermann this can be done with just computational equality. For evenness the involution law is needed. For Fibonacci, the recursive equation requires an inductive proof of a generalized equation.

For the uniqueness proofs one needs an induction scheme that works for the specification. For evenness and Fibonacci, doing the induction on $\forall n.\ pn \wedge p(\mathsf{S}n)$ instead of $\forall n.pn$ works. For Ackermann, nested induction on the two arguments does the job.

Recursive specifications of functions can also be interpreted in a programming language or in set theory. In contrast to computational type theory, termination is not a primary concern with these systems: nonterminating procedures are fine in programming languages and partial functions are the default case in set theory. There is a first-order computational system of μ-recursive functions where termination is not an issue and where functions are interpreted as partial functions on numbers.

10 Informative Types

Informative types combine computational and propositional information. They are obtained with computational variants of disjunctions and existential quantifications known as sum types ($X+Y$) and sigma types ($\Sigma x.px$). Informative types are a unique feature of Coq's type theory having no equivalent in set-theoretic mathematics. With informative types one can elegantly state computational properties that don't have adequate formal statements in set-theoretic mathematical language.

We explain informative types further with two examples. The informative type

$$\forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y)$$

describes functions that given two numbers decide equality and return a proof certifying the decision. The informative type compares to the propositional type

$$\forall xy^{\mathsf{N}}.\ (x = y) \vee (x \neq y)$$

and to the purely computational type

$$\mathsf{N} \to \mathsf{N} \to \mathsf{B}$$

While the computational type admits all kind of tests, the propositional type agrees with the informative type but for the crucial difference that functions of the propositional type cannot be used for computational decisions because of the elim restriction.

The informative type

$$\forall xy^{\mathsf{N}}\, \Sigma z.\ (x + z = y) \vee (y + z = x)$$

describes functions that given two numbers return the distance between the numbers and a proof certifying the result. The informative type compares to the propositional type

$$\forall xy^{\mathsf{N}}\, \exists z.\ (x + z = y) \vee (y + z = x)$$

and to the purely computational type

$$\mathsf{N} \to \mathsf{N} \to \mathsf{N}$$

While the computational type admits all kind of functions, the propositional type agrees with the informative type except for the difference that for functions of the propositional type the distance cannot be used computationally.

An important application of sum types are decision types

$$\mathcal{D}(P) \;:=\; P + \neg P$$

which may be used to write the types of certifying deciders for a predicate p:

$$\forall x.\, \mathcal{D}(px)$$

Certifying deciders combine computational deciders with correctness proofs. This combination is beneficial for both the use and the construction of computational deciders.

With informative types we can realize functions as typed constants hiding their definitions. We speak of *computational lemmas* and *computational proofs* since the construction generalizes propositional lemmas. Once we have a function of an informative type, we can extract a function of a purely computational type together with a correctness proof. We may see a function of an informative type as a combination of a computational function and a correctness proof.

Constructing functions as computational proofs is appropriate whenever there is a sufficiently complete specification of the function, as in the case of the two motivating examples.

The construction of computational proofs is very similar to the construction of propositional proofs. Constructing computational proofs (i.e., functions) with the abstractions commonly used for propositional proofs (e.g., structural induction and structural case analysis) turns out to be pleasant. As with propositional proofs, the construction is strongly guided by the accompanying types. When working with Coq, computational proofs are best constructed with the tactics known from propositional proofs, which smoothly generalize to computational proofs.

10.1 Sum Types and Sigma Types

We start with a table listing propositional types together with their computational counterparts:

$\forall$	$\to$	$\times$	$\Leftrightarrow$	$+$	Σ	computational types in $\mathbb{T}$
$\forall$	$\to$	$\wedge$	$\longleftrightarrow$	$\vee$	$\exists$	propositional types in $\mathbb{P}$

As it comes to function types ($\forall$, $\to$), the transition from the propositional variant to the computational variant is implicit. For conjunctions we have product types as computational counterpart, and for propositional equivalence we define the computational variant (propositional equivalence of types) as follows:

$$\Leftrightarrow : \mathbb{T} \to \mathbb{T} \to \mathbb{T}$$
$$X \Leftrightarrow Y \;:=\; (X \to Y) \times (Y \to X)$$

Thus an inhabitant of an equivalence $X \Leftrightarrow Y$ is a pair (f, g) of functions $f : X \to Y$ and $g : Y \to X$.

The computational counterparts for disjunctions and existential quantifications are called **sum types** $(X + Y)$ and **sigma types** $(\Sigma x.px)$. Their inductive definitions mimic the inductive definitions of disjunctions and existential quantifications and simply replace the universe $\mathbb{P}$ with the universe $\mathbb{T}$:

$$\begin{aligned} +\,(X : \mathbb{T},\, Y : \mathbb{T}) : \mathbb{T} \;&::=\; \mathsf{L}(X) \mid \mathsf{R}(Y) \\ \mathsf{sig}\,(X : \mathbb{T},\, p : X \to \mathbb{T}) : \mathbb{T} \;&::=\; \mathsf{E}\,(x : X,\, px) \end{aligned}$$

Similar to the notation $\exists x.s$ for propositions $\mathsf{ex}\,(\lambda x.s)$, we shall use the notation $\Sigma x.s$ for sigma types $\mathsf{sig}\,(\lambda x.s)$. The full types of the value constructors for sum and sigma types are as follows:

$$\begin{aligned} \mathsf{L} &: \forall X^{\mathbb{T}} Y^{\mathbb{T}}.\, X \to X + Y \\ \mathsf{R} &: \forall X^{\mathbb{T}} Y^{\mathbb{T}}.\, Y \to X + Y \\ \mathsf{E} &: \forall X^{\mathbb{T}} p^{X \to \mathbb{T}} x^{X}.\, px \to \mathsf{sig}\, X p \end{aligned}$$

We will treat X and Y as implicit arguments.

A value of a sum type $X + Y$ carries a value of X or a value of Y, where the information which alternative is present can be used computationally. The elements of sum types are sometimes called **variants**.

A value of a sigma type $\Sigma x.px$ carries a **witness** $x : X$ and a **certificate** $c : px$ (which often is a proof). The values of sigma types may be called **Σ-pairs** or **dependent pairs** (since the type of the certificate depends on the witness).

While most uses of sum types $X + Y$ and sigma types $\Sigma x.px$ are such that X, Y, and px are propositions, the more general cases matter. They are for instance needed when we nest informative types as in $(P_1 + P_2) + P_3$ or $\Sigma x.\Sigma y.pxy$.

We can define functions

$$\begin{aligned} P + Q &\to P \vee Q \\ (\Sigma x.px) &\to \exists x.px \end{aligned}$$

establishing disjunctions and existential quantifications as consequences of their computational variants. Functions in the other direction cannot be defined because of the elim restriction.

Exercise 10.1.1 Prove the constructor laws for sum types:

a) $\mathsf{L}x \neq \mathsf{R}y$.

b) $\mathsf{L}x = \mathsf{L}x' \to x = x'$.

c) $\mathsf{R}y = \mathsf{R}y' \to y = y'$.

Explain why $\mathsf{L}x \neq \mathsf{R}y$ cannot be shown for disjunctions.

Exercise 10.1.2 (Functional characterisations)
Prove the following equivalences:

a) $X + Y \Leftrightarrow \forall Z^{\mathbb{T}}.\ (X \to Z) \to (Y \to Z) \to Z.$

b) $(\Sigma x.px) \Leftrightarrow \forall Z^{\mathbb{T}}.\ (\forall x.\ px \to Z) \to Z.$

Exercise 10.1.3 Define functions as follows:

a) $\forall b^{\mathsf{B}}.\ (b = \mathbf{T}) + (b = \mathbf{F}).$

b) $\forall n^{\mathsf{N}}.\ (n = 0) + (\Sigma k.\, n = \mathsf{S}k).$

c) $\forall XYZ^{\mathbb{T}}.\ (X \to Z) \to (Y \to Z) \to (X + Y) \to Z.$

d) $\forall XYZ^{\mathbb{T}}.\ (Y \Leftrightarrow Z) \to (X + Y \Leftrightarrow X + Z).$

e) $\forall xy^{\mathsf{B}}.\ x \,\&\, y = \mathbf{F} \Leftrightarrow (x = \mathbf{F}) + (y = \mathbf{F}).$

f) $\forall xy^{\mathsf{B}}.\ x \mid y = \mathbf{T} \Leftrightarrow (x = \mathbf{T}) + (y = \mathbf{T}).$

10.2 Computational Lemmas

To state and demonstrate that a proposition P is provable we typically write something like

Fact P.
Proof. $\cdots$ ∎

The proof part is usually some informal text that explains how a value of type P can be defined. Once the proof is closed the details of the proof don't matter, what matters is simply that an inhabitant of P is definable.

It turns out that the above scheme also works well for informative types. As with propositions, the details of a definition of an inhabitant of an informative type do not matter, but the fact that one can define an inhabitant is essential. We will speak of a **propositional lemma** if P is a proposition, and of a **computational lemma** if P is a type that is not a proposition. Following this line, we speak of **propositional proofs** and of **computational proofs**.

We are now ready to prove two computational lemmas stating the definability of functions satisfying the informative function types explained in the introduction of the chapter.

Fact 10.2.1 $\forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y).$

Proof By induction on x with y quantified, followed by case analysis on y. The interesting case is $(\mathsf{S}x = \mathsf{S}y) + (\mathsf{S}x \neq \mathsf{S}y)$. We do case analysis on the instantiated inductive hypothesis $(x = y) + (x \neq y)$. The second case follows by injectivity of the constructor S. ∎

Note that the proof text agrees with the text we have given in § 6.2 for the corresponding proposition

$$\forall xy^{\mathsf{N}}.\ (x = y) \vee (x \neq y)$$

When we check the proof for the informative type, we have to make sure that the induction on x, the case analysis on y, and the case analysis on the instantiated inductive hypothesis are all admissible in a computational context. As it comes to induction and case analysis on numbers, this is always the case. As it comes to the case analysis on the instantiated inductive hypothesis, there is no problem since the inductive hypothesis is formulated with a sum type rather than a disjunction.

We now come to the distance example from the chapter introduction. We strengthen the informative type by replacing the disjunction with a sum type.

Fact 10.2.2 $\forall xy^{\mathsf{N}}\Sigma z.\ (x + z = y) + (y + z = x)$.

Proof By induction on x with y quantified, followed by case analysis on y in the successor case. The cases where $x = 0$ or $y = 0$ are trivial. The interesting case $\Sigma z.\ (\mathsf{S}x + z = \mathsf{S}y) + (\mathsf{S}y + z = \mathsf{S}x)$ follows by case analysis on the instantiated inductive hypothesis $\Sigma z.\ (x + z = y) + (y + z = x)$. ∎

Exercise 10.2.3 (Eliminator for some types) We define an eliminator for the sum type $X + Y$ as follows:

$$\begin{aligned}
&\mathsf{E}_+ :\ \forall p^{X+Y\to\mathbb{T}}.\ (\forall x.\, p(\mathsf{L}x)) \to (\forall y.\, p(\mathsf{R}y)) \to \forall a.\, pa\\
&\mathsf{E}_+\, pfg\,(\mathsf{L}x) \ :=\ fx\\
&\mathsf{E}_+\, pfg\,(\mathsf{R}y) \ :=\ gy
\end{aligned}$$

Use the eliminator E_+ to prove the following facts:

a) $\mathsf{L}x = \mathsf{L}x' \to x = x'$

b) $\forall a^{X+Y}.\ (\Sigma x.\, a = \mathsf{L}x) + (\Sigma y.\, a = \mathsf{R}y)$

10.3 Projections and Eliminator for Sigma Types

We assume a type function $p : X \to \mathbb{T}$ and define two projections that yield the witness and the certificate of the dependent pairs $a : \mathsf{sig}\, p$ obtained with p:

$$\begin{aligned}
&\pi_1 :\ \mathsf{sig}\, p \to X && \pi_2 :\ \forall a^{\mathsf{sig}\, p}.\ p(\pi_1 a)\\
&\pi_1\,(\mathsf{E}x_) \ :=\ x && \pi_2\,(\mathsf{E}xb) \ :=\ b
\end{aligned}$$

Note that the type of π_2 is given using the function π_1. This expresses the fact that the type of the certificate depends on the witness. Type checking the defining equation of π_2 requires conversion steps unfolding the definition of π_1.

We shall use the projections to define a translation function that, given a function $f : X \to Y$ satisfying $\forall x.\ px(fx)$, yields a certifying function $\forall x\, \Sigma y. pxy$. We say that the translation *merges* the function f and the correctness proof $\forall x.\ px(fx)$ into a single certifying function. We will also define a converse translation function that decomposes a certifying function $\forall x\, \Sigma y. pxy$ into a function $f : X \to Y$ and a correctness proof $\forall x.\ px(fx)$. The definability of the two translations can be stated elegantly using informative types.

Fact 10.3.1 (Skolem equivalence)
$\forall XY^{\mathbb{T}}\ \forall p^{X\to Y\to\mathbb{T}}.\ (\Sigma f\ \forall x.\ px(fx)) \Leftrightarrow (\forall x\, \Sigma y.\ pxy)$.

Proof The translation $\Rightarrow$ can be defined as $\lambda ax.\, E(\pi_1 ax)(\pi_2 ax)$. The converse translation $\Leftarrow$ can be defined as $\lambda F.\, E(\lambda x.\, \pi_1(Fx))(\lambda x.\, \pi_2(Fx))$. ∎

Note that type checking the above proof requires several conversion steps unfolding the definitions of the projections π_1 and π_2.

The next fact says that sum types may be expressed as sigma types.

Fact 10.3.2 (Sum-sigma equivalence)
$\forall XY^{\mathbb{T}}.\ (X + Y) \Leftrightarrow (\Sigma b.\, \text{IF } b \text{ THEN } X \text{ ELSE } Y)$.

Proof Translation $\Rightarrow$: $\lambda a.\, \text{MATCH } a\ [\, \mathsf{L}\, x \Rightarrow \mathsf{E}\, \mathbf{T}\, x \mid \mathsf{R}\, y \Rightarrow \mathsf{E}\, \mathbf{F}\, y\,]$.

Translation $\Leftarrow$: $\lambda a.\, \text{MATCH } a\ [\, \mathsf{E}\, bc \Rightarrow (\text{IF } b \text{ THEN } \mathsf{L} \text{ ELSE } \mathsf{R})\, c\,]$. ∎

Type checking of the translation functions makes again heavy use of conversion. You may want to replace the nested conditional with an application of the boolean eliminator to make the typing constraints more explicit.

Exercise 10.3.3 Try to prove the propositions corresponding to the informative types of Facts 10.3.1 and 10.3.2 informally. Note that some parts of the informal proofs are far from the rigorous formulations given in the proofs of the facts. On the other hand, the informal proofs translate smoothly into Coq proof scripts, which also work for the informative types.

Exercise 10.3.4 (Distance) Let $D : \forall xy^{\mathsf{N}}\, \Sigma k.\ (x + k = y) + (y + k = x)$. Prove the following equations:

a) $\pi_1(Dxy) = (x - y) + (y - x)$.

b) $x - y = \text{IF } \pi_2(Dxy) \text{ THEN } 0 \text{ ELSE } \pi_1(Dxy)$.

Note that the definition of D is not needed for the proofs since all necessary information about D is in the given type. Hint: Destructure Dxy and simplify. What remains are equations involving truncating subtraction only.

Exercise 10.3.5 (Eliminator) We assume a type function $p : X \to \mathbb{T}$ and define an eliminator for dependent pairs $a : \mathsf{sig}\, p$:

$$\begin{aligned} &\mathsf{E}_\Sigma : \ \forall q^{\mathsf{sig}\to\mathbb{T}}.\ (\forall xc.\, q(\mathsf{E}\, xc)) \to \forall a.\, qa \\ &\mathsf{E}_\Sigma\, q\, f\, (\mathsf{E}\, xc) \ := \ fxc \end{aligned}$$

a) Give the types for the variables x and c.

b) Prove the η-law $\mathsf{E}\,(\pi_1 a)(\pi_2 a) = a$ using the eiminator E_Σ.

c) Define the two projection functions with E_Σ and show that these definitions are computationally equal to π_1 and π_2.

Exercise 10.3.6 (Injectivity laws) Consider dependent pairs $a : \mathsf{sig}\, p$ for a type function $p : X \to \mathbb{T}$. One would think that the injectivity laws

$$\begin{aligned} &\mathsf{E}\, xc = \mathsf{E}\, x'c' \ \to \ x = x' \\ &\mathsf{E}\, xc = \mathsf{E}\, xc' \ \to \ c = c' \end{aligned}$$

are both provable. While the first law is easy to prove, the second law cannot be shown in Coq's type theory. This certainly conflicts with intuitions that worked well so far. The problem is with subtleties of dependent type checking that we will not discuss here.

a) Prove the first injectivity law.

b) Try to prove the second injectivity law. If you think you have found a proof with pen and paper, check it with Coq to find out where it breaks. Note that the proof that rewrites $\pi_2(\mathsf{E}\, xc)$ to $\pi_2(\mathsf{E}\, xc')$ does not work since there is no well-typed rewrite predicate validating the rewrite.

10.4 Decision Types and Certifying Deciders

We define **decision types** as follows:

$$\begin{aligned} &\mathcal{D} : \ \mathbb{P} \to \mathbb{T} \\ &\mathcal{D}(P) \ := \ P + \neg P \end{aligned}$$

We call values of decision types **decisions**. A decision of type $\mathcal{D}(P)$ carries either a proof of P or a proof of $\neg P$. Because of the use of a sum type, the information which variant is present can be used computationally.

A **certifying decider** for a predicate $p : X \to \mathbb{P}$ is a function $\forall x.\, \mathcal{D}(px)$. That we can define a certifying decider for a predicate in Coq's type theory means that we can show within Coq's type theory that the predicate is algorithmically decidable (since every function definable in Coq's type theory is algorithmically computable). We say that a predicate is **decidable** if we can define a certifying decider for it.

We remark that the predicate

$$\mathsf{tsat} : \mathsf{N} \to \mathsf{B} := \lambda f.\ \exists n.\ f n = \mathbf{T}$$

is not algorithmically decidable. Hence it cannot be shown in Coq's type theory that tsat is decidable. The predicate says that a boolean test for numbers is satisfiable, that is, yields the boolean $\mathbf{T}$ for at least one number.

A **boolean decider** for a predicate $p : X \to \mathbb{P}$ is a function $f : X \to \mathsf{B}$ such that

$$\forall x.\ \ px \longleftrightarrow fx = \mathbf{T}$$

It turns out that we can define translations between boolean deciders and certifying deciders.

Fact 10.4.1 (Decider equivalence)
$\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}}.\ \ (\forall x. \mathcal{D}(px)) \ \Leftrightarrow\ (\Sigma f\ \forall x.\ \ px \longleftrightarrow fx = \mathbf{T})$.

Proof We prefer to describe the translations informally. Using Coq's tactic language, it is matter of routine to construct formal function definitions following the informal descriptions.

Let $F : \forall x. \mathcal{D}(px)$. We define a boolean decider $fx := \text{IF } Fx \text{ THEN } \mathbf{T} \text{ ELSE } \mathbf{F}$ and prove $\forall x.\ \ px \longleftrightarrow fx = \mathbf{T}$ by fixing x and doing case analysis on Fx.

For the other direction, suppose $\forall x.\ \ px \longleftrightarrow fx = \mathbf{T}$. We fix x and show $\mathcal{D}(px)$ by case analysis on fx. If $fx = \mathbf{T}$, we show px, otherwise we show $\neg px$. ∎

We state the basic propagation laws for decisions. All of them are easy to prove.

Fact 10.4.2 (Propagation of decisions)

1. $\mathcal{D}(\top)$ and $\mathcal{D}(\bot)$.
2. $\forall PQ.\ \ \mathcal{D}(P) \to \mathcal{D}(Q) \to \mathcal{D}(P \to Q)$.
3. $\forall PQ.\ \ \mathcal{D}(P) \to \mathcal{D}(Q) \to \mathcal{D}(P \wedge Q)$.
4. $\forall PQ.\ \ \mathcal{D}(P) \to \mathcal{D}(Q) \to \mathcal{D}(P \vee Q)$.
5. $\forall P.\ \ \mathcal{D}(P) \to \mathcal{D}(\neg P)$.
6. $\forall PQ.\ \ (P \longleftrightarrow Q) \to (\mathcal{D}(P) \Leftrightarrow \mathcal{D}(Q))$.

Exercise 10.4.3 Prove the claims of Fact 10.4.2.

Exercise 10.4.4 Define a function $\forall X^{\mathbb{T}}\ f^{X \to \mathsf{B}}\ x^{X}.\ \ \mathcal{D}(fx = \mathbf{T})$.

Exercise 10.4.5 Define functions as follows.

a) $\forall P^{\mathbb{P}}.\ \ \mathcal{D}(P) \ \Leftrightarrow\ \Sigma b^{\mathsf{B}}.\ \text{IF } b \text{ THEN } P \text{ ELSE } \neg P$.

b) $\forall P^{\mathbb{P}}.\ \ \mathcal{D}(P) \ \Leftrightarrow\ \Sigma b^{\mathsf{B}}.\ P \longleftrightarrow b = \mathbf{T}$.

10.5 Discrete Types

We call a type X **discrete** if we can define a certifying equality decider for it:

$$\forall xy^X.\ \mathcal{D}(x = y)$$

In other words, a type is discrete if its equality predicate is decidable. We define

$$\mathcal{E}(X^{\mathbb{T}}) : \mathbb{T} \ :=\ (\forall xy^X.\ \mathcal{D}(x = y))$$

Note that $\mathcal{E}(X)$ is the type of certifying equality deciders for X.

Fact 10.5.1 (Propagation of equality decisions)

1. $\mathcal{E}(\bot)$, $\mathcal{E}(\top)$, $\mathcal{E}(\mathsf{B})$, $\mathcal{E}(\mathsf{N})$.
2. $\forall XY^{\mathbb{T}}.\ \mathcal{E}(X) \to \mathcal{E}(Y) \to \mathcal{E}(X \times Y)$.
3. $\forall XY^{\mathbb{T}}.\ \mathcal{E}(X) \to \mathcal{E}(Y) \to \mathcal{E}(X + Y)$.
4. $\forall XY^{\mathbb{T}} \forall f^{X \to Y}.\ \mathsf{injective}\, \mathsf{f} \to \mathcal{E}(Y) \to \mathcal{E}(X)$.

Proof $\mathcal{E}(\mathsf{N})$ is immediate from Fact 10.2.1. The other claims all have straightforward proofs. We use $\mathsf{injective}\, f := \forall xx'.\ fx = fx' \to x = x'$. ∎

Exercise 10.5.2 Proof the claims of Fact 10.5.1.

Exercise 10.5.3 Prove that a type has a certifying equality decider if and only if it has a boolean equality decider: $\forall X.\ \mathcal{E}(X) \Leftrightarrow \Sigma f^{X \to X \to \mathsf{B}}.\ \forall xy.\ x = y \longleftrightarrow fxy = \mathbf{T}$.

10.6 Option Types

Given a type X, we may see the sum type $X + \top$ as a type that extends X with one additional element. Such one-element extensions are often useful and are accommodated with dedicated inductive types called **option types**:

$$\mathcal{O}(X : \mathbb{T}) : \mathbb{T} \ ::=\ {}^\circ X \mid \emptyset$$

The types of the constructors introduced by this inductive type definition are as follows:

$$\begin{aligned} \mathcal{O} &: \mathbb{T} \to \mathbb{T} \\ {}^\circ &: \forall X^{\mathbb{T}}.\ X \to \mathcal{O}(X) \\ \emptyset &: \forall X^{\mathbb{T}}.\ \mathcal{O}(X) \end{aligned}$$

As usual, we treat the argument X of the value constructors as implicit argument. Following language from functional programming, we pronounce the constructors ${}^\circ$ and $\emptyset$ as *some* and *none*. We offer the intuition that $\emptyset$ is the new element and that ${}^\circ$ injects the elements of X into $\mathcal{O}(X)$.

Fact 10.6.1 (Constructor laws)
The constructors $\circ$ and $\emptyset$ are disjoint, and that the constructor $\circ$ is injective.

Proof Exercise. ∎

Fact 10.6.2 (Discreteness)
$\forall X^{\mathbb{T}}.\ \mathcal{E}(X) \to \mathcal{E}(\mathcal{O}(X))$.

Proof Exercise. ∎

Exercise 10.6.3 Prove $\forall a^{\mathcal{O}(X)}.\ a \neq \emptyset \Leftrightarrow \Sigma x.\ a = \circ x$.

Exercise 10.6.4 (Truncation flag) Define a recursive function $f : \mathsf{N} \to \mathsf{N} \to \mathcal{O}(\mathsf{N})$ that yields $\emptyset$ if $x - y$ truncates and $\circ(x - y)$ if $x - y$ doesn't truncate. Prove the equation $fxy = \text{IF } y - x \text{ THEN } \circ(x - y) \text{ ELSE } \emptyset$.

Exercise 10.6.5 (Bijectivity) We say that two types X and Y are *in bijection* if we can define functions $f : X \to Y$ and $g : Y \to X$ such that $\forall x.\ g(fx) = x$ and $\forall y.\ f(gy) = y$. Recall that we defined a bijection between N and $\mathsf{N}\times\mathsf{N}$ in Chapter 7. Show that the following types are in bijection:

1. B and $\top + \top$.
2. B and $\mathcal{O}(\mathcal{O}(\bot))$.
3. $\top$ and $\mathcal{O}(\bot)$.
4. $\mathcal{O}(X)$ and $X + \top$.
5. N and $\mathcal{O}(\mathsf{N})$.

Exercise 10.6.6 (Finite size) Note that $\mathcal{O}^n(\bot)$ is a type that has exactly n elements. Given our definitions so far, this is an informal statement. If we want to have a general definition of finite size, we may say that a type has size n if and only if it is in bijection with $\mathcal{O}^n(\bot)$. To justify this definition, we should prove that $\mathcal{O}^m(\bot)$ and $\mathcal{O}^n(\bot)$ are in bijection if and only if $m = n$. We will study this and related questions in a later chapter on finite types.

11 Numbers

Numbers $0, 1, 2, \ldots$ constitute the basic infinite data structure. Starting from the familiar inductive definition, we develop a computational theory of numbers based on type theory. A main topic is the familiar ordering of numbers. The computational results we derive include operators for trichotomy and least witnesses. There is much beauty in developing the theory of numbers from first principles.

11.1 Inductive Definition

Following the informal presentation in Chapter 1, we introduce the type of numbers $0, 1, 2 \ldots$ with an inductive definition

$$\mathsf{N} \ ::= \ 0 \mid \mathsf{S}(\mathsf{N})$$

introducing three constructors:

$$\mathsf{N} : \mathbb{T}, \qquad 0 : \mathsf{N}, \qquad \mathsf{S} : \mathsf{N} \to \mathsf{N}$$

Based on the inductive type definition, we can define functions with equations using exhaustive case analysis and structural recursion. A basic inductive function definition obtains an eliminator E_N providing for inductive proofs on numbers:

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

A discussion of the eliminator appears in § 6.2. Matches for numbers can be obtained as applications of the eliminator where no use of the inductive hypothesis is made. More directly, a specialized elimination function for matches omitting the inductive hypothesis can be defined.

Fact 11.1.1 (Constructors)

1. $\mathsf{S}x \neq 0$ (disjointness)
2. $\mathsf{S}x = \mathsf{S}y \to x = y$ (injectivity)
3. $\mathsf{S}x \neq x$ (progress)

Proof The proofs of (1) and (2) are discussed in § 5.2. Claim 3 follows by induction on x using (1) and (2). ∎

Fact 11.1.2 (Discreteness) N is a discrete type: $\forall xy^{\mathsf{N}}.\, \mathcal{D}(x = y)$.

Proof Fact 10.2.1. ∎

Exercise 11.1.3 Show the constructor laws and discreteness using the eliminator and without using matches.

Exercise 11.1.4 (Double induction) Prove the following double induction principle for numbers (from Smullyan and Fitting [11]):

$$\begin{aligned}
&\forall p^{\mathsf{N}\to\mathsf{N}\to\mathbb{T}}.\\
&(\forall x.\, px0) \to\\
&(\forall xy.\, pxy \to pyx \to px(\mathsf{S}y)) \to\\
&\forall xy.\, pxy
\end{aligned}$$

There is a nice geometric intuition for the truth of the principle: See a pair (x, y) as a point in the discrete plane spanned by N and convince yourself that the two rules are enough to reach every point of the plane.

An interesting application of double induction appears in Exercise 11.5.14.

Hint: First do induction on y with x quantified. In the successor case, first apply the second rule and then prove pxy by induction on x.

11.2 Addition

We accommodate addition of numbers with a recursively defined function:

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

The two most basic properties of addition are **associativity** and **commutativity**.

Fact 11.2.1 $(x + y) + z = x + (y + z)$ 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: $x + y + z \rightsquigarrow (x + y) + z$. Quite often the symmetric versions $x + 0 = x$ and $x + \mathsf{S}y = \mathsf{S}(x + y)$ of the defining equations will be used.

Another important fact about numbers is injectivity, which comes in two flavors.

Fact 11.2.2 (Injectivity) $x + y = x + z \to y = z$ and $x + y = x \to y = 0$.

Proof Both claims follow by induction on x. ∎

Exercise 11.2.3 Prove $x \neq x + \mathsf{S}y$.

11.3 Multiplication

We accommodate addition of numbers with a recursively defined function:

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

The definition is such that the equations

$$0 \cdot y = 0 \qquad 1 \cdot y = y + 0 \qquad 2 \cdot y = y + (y + 0)$$

hold by computational equality.

Proving the familiar properties of multiplication like associativity, commutativity, and distributivity is routine. In contrast to addition, multiplication will play only a minor role in this text.

Exercise 11.3.1 Prove that multiplication is commutative and associative.

11.4 Subtraction

We define (truncating) subtraction of numbers as a total operation that yields 0 whenever the standard subtraction operation for integers yields a negative number:

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

Note that the recursion is on the first argument and that in the successor case there is a case analysis on the second argument. Truncating subtraction plays a major role in our theory of numbers since we shall use it to define the canonical order on numbers.

Fact 11.4.1

1. $x - 0 = x$
2. $x - (x + y) = 0$
3. $x - x = 0$
4. $(x + y) - x = y$

Proof Claim 1 follows by case analysis on x. Claim 2 follows by induction on x. Claim 3 follows with (2) with $y = 0$. Claim 4 follows by induction on x using (1) for the base case. ∎

11.5 Order

We define the order relation on numbers using truncating subtraction:

$$x \le y \;:=\; (x - y = 0)$$

While this definition is nonstandard, it is quite convenient for deriving the basic properties of the order relation. We define the usual notational variants for the order relation:

$$\begin{aligned} x < y &:= \mathsf{S}x \le y \\ x \ge y &:= y \le x \\ x > y &:= y < x \end{aligned}$$

Fact 11.5.1 The following propositions hold by computational equality:

1. $(\mathsf{S}x \le \mathsf{S}y) = (x \le y)$ (shift law)
2. $0 \le x$
3. $0 < \mathsf{S}x$

Fact 11.5.2 (Decidability) $\mathcal{D}(x \le y)$.

Proof Immediate with Fact 11.1.2. ∎

Fact 11.5.3 $x \le y \to x + (y - x) = y$.

Proof By induction on x with y quantified. The base case is immediate with (1) of Fact 11.4.1. In the successor case we proceed with case analysis on y. Case $y = 0$ is contradictory. For the successor case, we exploit the shift law. We assume $x \leq y$ and show $\mathsf{S}(x + (y - x)) = \mathsf{S}y$, which follows by the inductive hypothesis. ∎

Fact 11.5.4 $x \leq y \longleftrightarrow \exists k.\, x + k = y$.

Proof Direction → follows with Fact 11.5.3, direction ← follows with Fact 11.4.1 (2). ∎

Fact 11.5.5

1. $x \leq x + y$
2. $x \leq \mathsf{S}x$
3. $x + y \leq x \to y = 0$
4. $x \leq 0 \to x = 0$
5. $x \leq x$ (reflexivity)
6. $x \leq y \to y \leq z \to x \leq z$ (transitivity)
7. $x \leq y \to y \leq x \to x = y$ (antisymmetry)

Proof Claim 1 follows with Fact 11.4.1 (2). Claim 2 follows from (1). Claim 3 follows with Fact 11.4.1 (4). Claim 4 follows by case analysis on x and constructor disjointness.

Reflexivity follows with Fact 11.4.1 (3).

For transitivity, we assume $x + a = y$ and $y + b = z$ using Fact 11.5.4. Then $z = x + a + b$. Thus $x \leq z$ by (1).

For antisymmetry, we assume $x + a = y$ and $x + a \leq x$ using Fact 11.5.4. By (3) we have $a = 0$, and thus $x = y$. ∎

Fact 11.5.6 (Strict transitivity)

1. $x < y \leq z \to x < z$
2. $x \leq y < z \to x < z$.

Proof We show (1), (2) is similar. Using Fact 11.5.4, the assumptions give us $\mathsf{S}x + a = y$ and $y + b = z$. Thus it suffices to prove $\mathsf{S}x \leq \mathsf{S}x + a + b$, which follows by Fact 11.5.5 (1). ∎

Fact 11.5.7

1. $\neg(x < 0)$
2. $\neg(x + y < x)$ (strictness)
3. $\neg(x < x)$ (strictness)
4. $x \leq y \to x \leq y + z$
5. $x \leq y \to x \leq \mathsf{S}y$

Proof Claim 1 converts to $\mathsf{S}x \neq 0$. For Claim 2 we assume $\mathsf{S}x + y - x = 0$ and obtain the contradiction $\mathsf{S}y = 0$ with Fact 11.4.1 (4). Claim 3 follows from (2). For Claim 4 we assume $x + a = y$ using Fact 11.5.4 and show $x \leq x + a + z$ using Fact 11.5.5 (1). Claim 5 follows from (4). ∎

Fact 11.5.8 $x - y \leq x$

Proof Induction on x with y quantified. The base case follows by conversion. The successor case is done with case analysis on y. If $y = 0$, the claim follows with reflexivity. For the successor case $y = \mathsf{S}y$, we have to show $\mathsf{S}x - \mathsf{S}y \leq \mathsf{S}x$. We have $\mathsf{S}x - \mathsf{S}y = x - y \leq x \leq \mathsf{S}x$ using shift, the inductive hypothesis, and Fact 11.5.5 (2). The claim follows by transitivity. ∎

Next we prove an induction principle known as **complete induction**, which improves on structural induction by providing an inductive hypothesis for every $y < x$, not just the predecessor of x.

Fact 11.5.9 (Complete Induction)
$\forall p^{\mathsf{N}\to\mathbb{T}}.\ (\forall x.\ (\forall y.\ y < x \to py) \to px) \to \forall x.px.$

Proof We assume p and the *step function*

$$F : \forall x.\ (\forall y.\ y < x \to py) \to px$$

and show $\forall x.px$. The trick is now to prove the equivalent claim

$$\forall nx.\ x < n \to px$$

by structural induction on n. For $n = 0$, the claim is trivial. In the successor case, we assume $x < \mathsf{S}n$ and prove px. We apply the step function F, which gives us the assumption $y < x$ and the claim py. By the inductive hypothesis it suffices to show $y < n$, which follows by strict transitivity (Fact 11.5.6). ∎

We will not give examples for the use of complete induction here. Chapter 12 introduces a generalization of complete induction called size recursion and studies an example (Euclidean division) that can be done with complete induction.

Exercise 11.5.10 Prove $y > 0 \to y - \mathsf{S}x < y$.

Exercise 11.5.11 Prove $x + y \leq x + z \to y \leq z$.

Exercise 11.5.12 Define a boolean decider for $x \leq y$ and prove its correctness.

Exercise 11.5.13 Define a function $\forall xy.\ x \leq y \to \Sigma k.\ x + k = y$.

Exercise 11.5.14 Use the double induction operator from Exercise 11.1.4 to prove $\forall xy.\ (x \leq y) + (y < x)$. No further induction or lemma is necessary.

11.6 Trichotomy

We define a trichotomy operator that given two numbers decides how they are ordered.

Fact 11.6.1 (Trichotomy) $\forall x y^{\mathsf{N}}.\ (x < y) + (x = y) + (y < x)$.

Proof By induction on x with y quantified. Both cases need case analysis on y. The inductive hypothesis is needed only in the successor-successor case. Fact 11.5.1 is useful. ∎

Fact 11.6.2 $x \le y \Leftrightarrow (x < y) + (x = y)$.

Proof For direction $\Rightarrow$, we assume $x \le y$ and use the trichotomy operator. If $y < x$, we have $x < x$ by strict transitivity and a contradiction by strictness.

For direction $\Leftarrow$, we assume either $\mathsf{S}x \le y$ or $x = y$. For $\mathsf{S}x \le y$, $x \le y$ follows with transitivity from $x \le \mathsf{S}x$. For $x = y$, the claim holds by reflexivity. ∎

Fact 11.6.3 $(x \le y) + (y < x)$.

Proof By Facts 11.6.1 and 11.6.2. ∎

The following corollaries have the flavor of excluded middle, but do have constructive proofs using the trichotomy operator.

Corollary 11.6.4 (Contraposition) $\neg(x > y) \to x \le y$.

Corollary 11.6.5 (Equality by Contradiction) $\neg(x < y) \to \neg(y < x) \to x = y$.

Proof Follows by contraposition and antisymmetry. ∎

Fact 11.6.6 Bounded quantification preserves decidability:

1. $(\forall x.\, \mathcal{D}(px)) \to \mathcal{D}(\forall x.\, x < k \to px)$.
2. $(\forall x.\, \mathcal{D}(px)) \to \mathcal{D}(\exists x.\, x < k \wedge px)$.

Proof By induction on k and Fact 11.6.2. ∎

Exercise 11.6.7 (Tightness) Prove $x \le y \le \mathsf{S}x \to x = y \vee y = \mathsf{S}x$.

Exercise 11.6.8 Formulate the contraposition corollaries as equivalences and prove them with the trichotomy operator.

Exercise 11.6.9 We define **divisibility** and **primality** as follows:

$$\begin{aligned} k \mid x &:= \exists n.\, x = n \cdot k \\ \mathsf{prime}\, x &:= x \geq 2 \wedge \forall k.\, k \mid x \to k = 1 \vee k = x \end{aligned}$$

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.

11.7 Least Witnesses

Assume a predicate $p : \mathsf{N} \to \mathbb{P}$ on numbers. If the predicate is *satisfied* by some x (i.e., px is provable), we would expect that there is a least x satisfying p. The resulting questions are interesting since we work in an intuitionistic logic and strive for computational results. To get computational results, we will have to assume that p is decidable. Then, given some number satisfying p, we will be able to compute the least number satisfying p. It will be convenient to call numbers satisfying p *witnesses*. We can now say that we want to define a function that given a witness yields the least witness. We call such a function a **least witness operator (LWO)**.

The computational idea for obtaining a least witness is called *linear search*: We check p for $k = 0, 1, 2, \ldots$ until we find the first k satisfying p. If there is a number satisfying p, linear search will terminate with the least such number. If there is no such number, linear search will not terminate. The challenge now is to write with structural recursion on numbers a function that given a witness returns the least witness.

We start with the definition of a **least witness predicate**:

$$\begin{aligned} \mathsf{safe}\, p\, n &:= \forall k.\, pk \to k \geq n \\ \mathsf{least}\, p\, n &:= pn \wedge \mathsf{safe}\, pn \end{aligned}$$

Fact 11.7.1

1. $\mathsf{least}\, p\, n \to \mathsf{least}\, p\, n' \to n = n'$ (uniqueness)
2. $\mathsf{safe}\, p0$
3. $\mathsf{safe}\, pn \to \neg pn \to \mathsf{safe}\, p(\mathsf{S}n)$

Proof Claim 1 follows with antisymmetry. Claim 2 is trivial. For Claim 3 we assume pk and show $k > n$. By contraposition (Fact 11.6.4) we assume $k \leq n$ and derive a contradiction. The first assumption and pk give us $k \geq n$. Thus $n = k$ by antisymmetry, which makes pk contradict $\neg pn$. ∎

We now approach the least witness operator and assume a decider for p. To accommodate the needs of structural recursion, we will work with two arguments n and k, where the recursion is on n and k is incremented when n is decremented. We start with some n and $k = 0$ and decrement n until pk is satisfied. If we reach $n = 0$, we return k without testing p.

$$\begin{aligned} L &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ L\,0\,k &:= k \\ L\,(\mathsf{S}n)\,k &:= \text{IF } \ulcorner pk \urcorner \text{ THEN } k \text{ ELSE } L\,n\,(\mathsf{S}k) \end{aligned}$$

Note the use of the **upper-corner notation** $\ulcorner pk \urcorner$. It acts as a placeholder for an application fk of a decision function f for p (boolean or informative). The use of the upper-corner notation is convenient since it saves us from naming the decision function.

To prove that $Ln0$ computes the least witness if n is a witness, we need to come up with a strong enough invariant for n and k.

Lemma 11.7.2 (Invariant)
$\forall nk.\ p(n+k) \to \mathsf{safe}\,pk \to \mathsf{least}\,p\,(Lnk)$.

Proof By induction on n with k quantified. For $n = 0$ the claim is trivial. For the successor case we assume $p(\mathsf{S}n + k)$, $\mathsf{safe}\,pk$, and show $\mathsf{least}\,p\,(L(Sn)k)$. If pk, it suffices to show $\mathsf{least}\,pk$, which is straightforward. Otherwise we have $\neg pk$ and show $\mathsf{least}\,p\,(Ln(\mathsf{S}k))$. By the inductive hypothesis it suffices to show $p(n+\mathsf{S}k)$ and $\mathsf{safe}\,p(\mathsf{S}k)$. Given the assumptions, the first claim is straightforward. The second claim follows with Fact 11.7.2 (3). ∎

Fact 11.7.3 (Least witness operator)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to \forall n.\ pn \to \Sigma k.\ \mathsf{least}\,pk$.

Proof Lemma 11.7.2 with $k = 0$. ∎

Note that the least witness operator is special in that it takes a proof of px as argument. The proposition px expresses a **precondition** that must be satisfied so that a least witness procedure terminates. This is the first time in this text we have constructed a computational function that takes a proof of a precondition as argument.

Corollary 11.7.4 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to (\exists n.pn) \to (\exists k.\ \mathsf{least}\,pk)$.

Fact 11.7.5 (Decidability)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to (\forall n.\,\mathcal{D}(\mathsf{least}\,pn))$.

Proof We show $\mathcal{D}(\mathsf{least}\,pn)$ for $n : \mathsf{N}$. If $\neg pn$, we have $\neg\mathsf{least}\,pn$. Otherwise we have pn. Thus $\mathsf{least}\,pk$ for some k by Fact 11.7.3. If $n = k$, we are done. If $n \neq k$, we assume $\mathsf{least}\,pn$ and obtain a contradiction with the uniqueness of $\mathsf{least}\,p$ (Fact 11.7.1). ∎

Exercise 11.7.6 Prove that $x - y$ is the least z such that $x \le y + z$:
$x - y = z \longleftrightarrow \mathsf{least}\,(\lambda z.x \le y + z)\,z$.

11.8 Least Witness Operator via Induction

We now give a compact definition of a least witness operator using the operator for structural induction on numbers. The trick is to not define the auxiliary function L explicitly but rather construct a certifying version using structural recursion on numbers.

Fact 11.8.1 (Least witness operator)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\mathcal{D}(pn)) \to \forall n.\ pn \to \Sigma x.\ \mathsf{least}\,px.$

Proof We assume p and a decider $\forall n.\mathcal{D}(pn)$ and prove the more general claim

$$\forall nk.\ p(n+k) \to \mathsf{safe}\,pk \to \Sigma x.\ \mathsf{least}\,px$$

mimicking the invariant lemma. We prove the claim by induction on n with k quantified. If $n = 0$, the claim holds by the definition of least. In the successor case, we assume $H : p(\mathsf{S}n + k)$ and $\mathsf{safe}\,pk$ and prove $\Sigma x.\ \mathsf{least}\,px$. We now use the decider and do case analysis on $pk + \neg pk$. If pk, $x = k$ satisfies the claim. If $\neg pk$, we have $\mathsf{safe}\,p(\mathsf{S}k)$. Since we also have $p(n+\mathsf{S}k)$ by assumption H, the inductive hypothesis yields the claim. ∎

Note that the inductive proof yields an elegant construction of a least witness operator that saves unnecessary details. Induction operators of various kinds will turn out to be the power tool for constructing functions that don't have straightforward definitions with structural recursion. We will usually refer to an induction operator as a recursion operator when using it for computational purposes.

11.9 Least Witnesses and Excluded Middle

The set-theoretic development of numbers comes with the prominent result that every nonempty set of numbers has a least element. Transferring to our type-theoretic setting, we can ask whether every satisfiable predicate on numbers has a least witness. Assuming the law of excluded middle, this is in fact the case. Even better,

we can show that the law of excluded middle is equivalent to the existence of least witnesses for satisfiable predicates on numbers.

We start with a lemma mimicking the invariant lemma 11.7.2 for a logically decidable predicate.

Lemma 11.9.1 $(\forall x.\ px \vee \neg px) \to p(n+k) \to \mathsf{safe}\, pk \to \exists x.\ \mathsf{least}\, px.$

Proof By induction on n with k quantified. The proof follows the proof of Lemma 11.7.2. Logical decidability of p suffices since the claim of the lemma is a proposition rather than a Σ-type. ∎

Fact 11.9.2 $(\forall x.\ px \vee \neg px) \to (\exists x.px) \to \exists x.\ \mathsf{least}\, px.$

Proof Straightforward with Lemma 11.9.1. Eliminating the witness of the existential assumption is fine since the claim is propositional. ∎

Theorem 11.9.3 $(\forall P^{\mathbb{P}}.\ P \vee \neg P) \longleftrightarrow (\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\exists x.px) \to (\exists x.\ \mathsf{least}\, px)).$

Proof Direction $\to$ follows with Fact 11.9.2. For direction $\leftarrow$, let P be a proposition. We define $pn := \textsc{match}\ n\ [\, 0 \Rightarrow P \mid \mathsf{S}\_ \Rightarrow \top\,]$. Since p is satisfiable, the assumption gives us n such that $\mathsf{least}\, pn$. If $n = 0$, we have $p0$ and thus P. If $n = \mathsf{S}k$, we assume P and obtain a contradiction since $\mathsf{safe}\, p(\mathsf{S}k)$ but $p0$. ∎

11.10 Notes

Our definition of the order predicate deviates from Coq's inductive definition. Coq comes with a very helpful automation tactic lia for linear arithmetic that proves almost all of the results in this chapter and that frees the user from knowing the exact definitions and lemmas. All our further Coq developments will rely on lia.

The reader may find it interesting to compare the computational development of the numbers given here with Landau's [8] classical development from 1929.

12 Size Recursion

We define a size recursion operator using structural recursion on numbers. With the operator we can construct functions obeying recursion schemes decreasing the argument size with respect to a numeric size function. We use the size recursion operator to construct functions for Euclidean division, greatest common divisors, and discrete inversion. In each case, we work with informative types stating the totality of relational specifications. The concrete definition of the size recursion operator does not matter since all information needed for its use is transported through informative types.

The size recursion operator formulates a familiar induction principle generalizing complete induction on numbers. The size operator shows once more that in type theory proof methods and computational methods come hand in hand.

The size recursion operator also shows that in a dependently typed high-order setting more general forms of recursion can be defined with structural recursion.

12.1 Size Recursion Operator

A basic intuition about recursion says that when we compute fx we may obtain fy by recursion for every y smaller than x. Similarly, when we prove px, we may assume a proof for py for every y smaller than x. Formally, we may provide for this intuition with a **size recursion operator**

$$\begin{aligned}&\forall X^{\mathbb{T}}\,\forall \sigma^{X\to\mathsf{N}}\,\forall p^{X\to\mathbb{T}}.\\ &\quad(\forall x.\,(\forall y.\,\sigma y<\sigma x\to py)\to px)\to\\ &\quad\forall x.\,px\end{aligned}$$

where $\sigma y<\sigma x$ formalizes that y is smaller than x using a **size function** σ. From the type of the size recursion operator we see that the operator obtains a **target function** $\forall x.px$ from a **step function**

$$\forall x.\,(\forall y.\,\sigma y<\sigma x\to py)\to px$$

The step function says how for x a px is computed, where for every y smaller than x a py is provided by a **continuation function**

$$\forall y.\,\sigma y<\sigma x\to py$$

It is helpful to see a size recursion operator as a generalisation of a structural recursion operator for numbers:

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

Generalising structural recursion, size recursion works on arbitrary types and provides recursion for every y smaller than x, not just the predecessor. Structural recursion comes with a base value $p0$ and a step function

$$\forall x.\ px \to p(\mathsf{S}x)$$

saying how for every number x a value of $p(\mathsf{S}x)$ can be obtained from a value of px.

The special case of size recursion where X is N, p is a predicate, and σ is the identity function is known as *complete induction* in mathematical reasoning (Fact 11.5.9).

It turns out that a size recursion operator can be defined with structural recursion. Given the step function, one can define an auxiliary function

$$\forall nx.\ \sigma x < n \to px$$

by structural recursion on the numeric argument n. By using the auxiliary function with $n = \mathsf{S}(\sigma x)$ one then obtains the target function $\forall x.px$.

Lemma 12.1.1 Let $X : \mathbb{T}$, $\sigma : X \to \mathsf{N}$, $p : X \to \mathbb{T}$, and

$$F : \forall x.\ (\forall y.\ \sigma y < \sigma x \to py) \to px$$

Then there is a function $\forall nx.\ \sigma x < n \to px$.

Proof We define the asserted function by structural recursion on n:

$$\begin{aligned} R :&\ \forall nx.\ \sigma x < n \to px \\ R0xh &:= \text{MATCH } \ulcorner\bot\urcorner\ [] && h : \sigma x < 0 \\ R(\mathsf{S}n)xh &:= Fx(\lambda y h'.\ Rny\ulcorner\sigma y < n\urcorner) && h : \sigma x < \mathsf{S}n,\ \ h' : \sigma y < \sigma x \end{aligned}$$

We can also phrase the definition of R as an inductive proof. While more verbose, the inductive proof formulation is easier to read and to write for humans.

We prove $\forall nx.\ \sigma x < n \to px$ by induction on n. If $n = 0$, we can assume $\sigma x < 0$, which is contradictory. For the inductive step, we assume $\sigma x < \mathsf{S}n$ and construct a value of px. Using the step function F, it suffices to construct a continuation function $\forall y.\ \sigma y < \sigma x \to py$. So we assume $\sigma y < \sigma x$ an prove py. Since $\sigma y < n$ by the assumptions $\sigma y < \sigma x < \mathsf{S}n$, the inductive hypothesis yields py. ∎

Theorem 12.1.2 (Size Recursion)
$\forall X^{\mathbb{T}}\, \forall \sigma^{X\to \mathsf{N}}\, \forall p^{X\to\mathbb{T}}.\ (\forall x.\ (\forall y.\ \sigma y < \sigma x \to py) \to px) \ \to\ \forall x.px.$

Proof Straightforward with Lemma 12.1.1. ∎

We have now seen a definition of a size recursion operator in Coq's type theory using structural recursion. We will see several applications of size recursion, both to construct functions and to prove theorems. For instance, we will use size induction to show that for two numbers x and y there always exist numbers a and b such that $x = a \cdot \mathsf{S}y + b$ and $b \le y$.

The size recursion theorem does not expose the definition of the recursion operator and we will not use the defining equations of the operator. When we use the size recursion operator to construct a function $f : \forall x.px$, we make sure that px is an informative type giving us all the information we need about fx.

The accompanying Coq development gives a transparent definition of the size recursion operator. This way we can actually compute with the functions defined with the recursion operator. This makes it possible to prove concrete equations by computational equality.

Exercise 12.1.3 Define operators for structural recursion on numbers

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

and for complete recursion on numbers (Fact 11.5.9)

$$\forall p^{\mathsf{N}\to\mathbb{T}}.\ (\forall x.\ (\forall y.\ y < x \to py) \to px) \to \forall x.px$$

using the size recursion operator.

Exercise 12.1.4 (Binary size recursion) Define a binary size recursion operator

$$\begin{aligned}&\forall XY^{\mathbb{T}}\, \forall \sigma^{X\to Y\to \mathsf{N}}\, \forall p^{X\to Y\to\mathbb{T}}.\\ &\quad (\forall xy.\ (\forall x'y'.\ \sigma x'y' < \sigma xy \to px'y') \to pxy) \to\\ &\quad \forall xy.\ pxy\end{aligned}$$

using the size recursion operator on the product type $X \times Y$.

12.2 Least Witness Operator Revisited

Recall from § 11.7 that a least witness operator (LWO) is a function

$$\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to \forall n.\ pn \to \Sigma x.\ \mathsf{least}\, px$$

It turns out that size recursion provides for a particularly elegant construction of an LWO using the ideas discussed in § 11.7.

Fact 12.2.1 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to \forall n.\ pn \to \Sigma x.\ \mathsf{least}\,px.$

Proof We assume pn and a decider for p. Since $\mathsf{safe}\,p0$, it suffices to prove

$$\forall k.\ \mathsf{safe}\,pk \to \Sigma x.\ \mathsf{least}\,px$$

by size induction on $n-k$. If pk, we are done. Otherwise, we have $\mathsf{safe}\,p(\mathsf{S}k)$ and the claim follows with the inductive hypothesis since $n-\mathsf{S}k < n-k$ since pn yields $\mathsf{S}k \le n$. ∎

12.3 Relational Specifications

We now sketch the method we will use in the following to construct functions with the size recursion operator. The method requires a relational specification

$$\varphi : X \to Y \to \mathbb{P}$$

of the function we want to construct. Given φ, we want a function

$$f : X \to Y$$

such that

$$\forall x.\,\varphi x(fx)$$

We will obtain such a function f by constructing a certifying function

$$F : \forall x\,\Sigma y.\,\varphi xy$$

The function F may be seen as a computational proof of the totality of the relation φ. In the examples we will consider in this chapter, we will construct F with size recursion using the return type function

$$px := \Sigma y.\,\varphi xy$$

and a suitable size function for the argument type X.

Note that **Σ-totality** of the specification predicate φ says that the specification φ is satisfiable. In practice, we will always work with **functional** specification predicates:

$$\forall xyy'.\,\varphi xy \to \varphi xy' \to y = y'$$

Functionality means that a satisfiable specification uniquely determines a function up to functional extensionality.

12.4 Euclidean Division

We will define a division operation inverting multiplication. There is the complication that inversion of multiplication is neither functional nor total on all numbers. The problem can be solved with a relational specification subsuming inversion of multiplication.

Relational Specification

We define the **discrete quotient** of two numbers x and $\mathsf{S}y$ as the unique number z such that

$$\delta xyz \;:=\; z \cdot \mathsf{S}y \le x < \mathsf{S}z \cdot \mathsf{S}y$$

For instance, the discrete quotient of 8 and 4 is 2, and the discrete quotient of 12 and 4 is 3, as is the discrete quotient of 15 and 4.

Fact 12.4.1 (Functionality) The predicate δ is functional. Thus the discrete quotient of two numbers is unique.

Proof Let δxyz and $\delta xyz'$. We show $z = z'$ by doing case analysis on the disjunction $z < z' \vee z = z' \vee z' < z$ (trichotomy, Fact 11.6.1). Wlog we assume $z < z'$ and derive a contradiction. We have $\mathsf{S}z \le z'$ and hence $x < \mathsf{S}z \cdot \mathsf{S}y \le z' \cdot \mathsf{S}y \le x$. ∎

Together with functionality, the following fact ensures that δ captures inversion of multiplication.

Fact 12.4.2 (Agreement) $x = z \cdot \mathsf{S}y \to \delta xyz$.

Proof Straightforward. ∎

Procedure

Our goal is the construction of a function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that

$$\forall xy.\ \delta xy(fxy)$$

The computational idea for f is to subtract $\mathsf{S}y$ from x as often as it is possible without truncation and take the number of subtractions as the discrete quotient of x and y. We formulate this computational idea with the recursive specification shown in Figure 12.1. The procedure described by the specification terminates for all arguments since every recursion step decreases the first argument x.

While the specification in Figure 12.1 can be realized easily in an execution-oriented programming language with a recursive procedure, more effort is needed to realize the specification with a function in type theory. The reason for the extra effort is the fact that the recursion in the specification is not structural. Given that

$$f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$
$$fxy \;=\; \begin{cases} 0 & \text{if } x \le y \\ \mathsf{S}(f(x - \mathsf{S}y)y) & \text{if } x > y \end{cases}$$

Figure 12.1: Recursive specification of a Euclidean division function

the recursion reduces the size of the argument x, we can use size recursion to construct a function satisfying the specification in type theory. The easiest way to do this consists in proving that the specification δ is Σ-total and functional.

To prove Σ-totality of δ, we will use size recursion. The following fact provides so-called recursion rules that may be seen as certifying versions of the equations of the specification in Figure 12.1. The recursion rules will be used for the construction of the (certifying) step function providing for the proof of Σ-totality.

Fact 12.4.3 (Recursion Rules)

1. $x \le y \to \delta xy0$.
2. $x > y \to \delta(x - \mathsf{S}y)yz \to \delta xy(\mathsf{S}z)$.

Proof Straightforward. ∎

Fact 12.4.4 There is a function $\mathsf{Div} : \forall xy\Sigma z.\, \delta xyz$.

Proof By size recursion on x using the recursion rules from Fact 12.4.3. ∎

Corollary 12.4.5 There is a function $D : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that $\forall xy.\, \delta xy(Dxy)$.

Recursive Specification

We now show that a function satisfies the specification δ if and only if it satisfies the recursive specification in Figure 12.1.

Fact 12.4.6 A function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ satisfies $\forall xy.\, \delta xy(fxy)$ if and only if it satisfies the recursive specification in Figure 12.1.

Proof Both directions of the proof are interesting.

Assume $\forall xy.\, \delta xy(fxy)$. We need to show that f satisfies the two equations of the recursive specification. For the second equation, we assume $x > y$ and show $fxy = \mathsf{S}(f(x - \mathsf{S}y)y)$. By the assumption and the recursion rule from Fact 12.4.3 we have $\delta xy(\mathsf{S}(f(x - \mathsf{S}y)y))$. Now the claim follows with the assumption and the functionality of δ (Fact 12.4.1).

For the other direction, assume f satisfies the recursive specification. We can now prove $\forall xy.\, \delta xy(fxy)$ by complete induction on x using case analysis on $x \le y \vee x > y$ and the recursion rules from Fact 12.4.3. ∎

Corollary 12.4.7 The function D satisfies the recursive specification in Figure 12.1.

We remark that in this section y is never modified. Thus y can be pulled out as a parameter. We have chosen this design in the Coq development.

Exercise 12.4.8 Use D to define a function $f : \mathsf{N} \to \mathsf{N}$ satisfying the equations

$$\begin{aligned} f\,0 &= 0 \\ f\,1 &= 0 \\ f(\mathsf{S}(\mathsf{S}n)) &= \mathsf{S}(fn) \end{aligned}$$

Prove that your function is correct.

Recall that the construction of a function satisfying the equations was already asked for in Exercise 1.5.2. There a construction with an extra boolean argument is suggested. Prove this construction correct.

12.5 Euclidean Division Theorem

We prove a basic representation theorem for numbers using the results we have obtained for discrete quotients.

Theorem 12.5.1 (Euclidean Division) Given two numbers x and y, there exist unique numbers a and b such that $x = a \cdot \mathsf{S}y + b$ and $b \le y$.

In fact, we will see that $a = Dxy$ and $b = x - Dxy \cdot \mathsf{S}y$. Thus both a and b can be computed. We define a function M that yields **remainders**:

$$Mxy \;:=\; x - Dxy \cdot \mathsf{S}y$$

Fact 12.5.2 (Existence) $x = Dxy \cdot \mathsf{S}y + Mxy$ and $Mxy \le y$.

Proof By linear arithmetic it suffices to show $x \ge Dxy \cdot \mathsf{S}y$, which follows with Corollary 12.4.5 and linear arithmetic. ∎

Fact 12.5.3 (Uniqueness) Let $x = a \cdot \mathsf{S}y + b$ and $b \le y$.
Then $a = Dxy$ and $b = Mxy$.

Proof By linear arithmetic it suffices to show $a = Dxy$. Since δ is functional (Fact 12.4.1), it suffices to show δxya and $\delta xy(Dxy)$. The first obligation follows with linear arithmetic and the second obligation is Corollary 12.4.5. ∎

Corollary 12.5.4 (Uniqueness) Let $a \cdot \mathsf{S}y + b = a' \cdot \mathsf{S}y + b'$ and $b, b' \le y$.
Then $a = a'$ and $b = b'$.

We now have a complete proof of Theorem 12.5.1.

Exercise 12.5.5 Show that M satisfies the equation

$$Mxy = \begin{cases} x & \text{if } x \leq y \\ M(x - \mathsf{S}y)y & \text{if } x > y \end{cases}$$

Exercise 12.5.6 Prove $\forall nx.\ \mathcal{D}(n \mid x)$. Hint: Show $\mathsf{S}k \mid x \longleftrightarrow Mxk = 0$ first.

Exercise 12.5.7 Let $\mathsf{even}\, n := \exists k.\ n = k \cdot 2$. Prove the following:

a) $\mathcal{D}(\mathsf{even}\, n)$.

b) $\mathsf{even}\, n \to \neg\mathsf{even}\, (\mathsf{S}n)$.

c) $\neg\mathsf{even}\, n \to \mathsf{even}\, (\mathsf{S}n)$.

Exercise 12.5.8 Prove $\mathsf{S}(2 \cdot x) \neq 2 \cdot y$ using uniqueness of Euclidean division. 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 12.5.9 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:

$$\begin{aligned} &D : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &\quad Dxy0 := 0 \\ &Dxy(\mathsf{S}z) := 0 && \text{if } x \leq y \\ &Dxy(\mathsf{S}z) := \mathsf{S}(D(x - \mathsf{S}y)yz) && \text{if } x > y \end{aligned}$$

$$\begin{aligned} &M : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &\quad Mxy0 := 0 \\ &Mxy(\mathsf{S}z) := x && \text{if } x \leq y \\ &Mxy(\mathsf{S}z) := M(x - \mathsf{S}y)yz && \text{if } x > y \end{aligned}$$

a) Prove $x < z \to x = Dxyz \cdot \mathsf{S}y + Mxyz$.

b) Prove $x = Dxy(\mathsf{S}x) \cdot \mathsf{S}y + Mxy(\mathsf{S}x)$.

Exercise 12.5.10 We consider two specifications of two functions $D, M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$. The *liberal specification* is

$$\begin{aligned} &Mxy \leq y \\ &x = Dxy \cdot \mathsf{S}y + Mxy \end{aligned}$$

The *recursive specification* is

$$Mxy = \begin{cases} x & \text{if } x \le y \\ M(x - \mathsf{S}y)y & \text{if } x > y \end{cases} \qquad Dxy = \begin{cases} 0 & \text{if } x \le y \\ \mathsf{S}(D(x - \mathsf{S}y)y) & \text{if } x > y \end{cases}$$

Show that the two specifications are equivalent and unique up to functional extensionality. Use results we have shown before.

12.6 Greatest Common Divisors

The techniques we have used for Euclidean division generalize to all situations where the function we want to construct can be computed with size recursion and, in addition, can be specified independently. For our second example we consider the construction of a function computing greatest comon divisors with repeated subtraction. This time we will need a size function depending on two arguments.

Relational Specification

We define the **divisors** of a number x as follows:

$$n \mid x \;:=\; \exists k.\; x = k \cdot n \qquad\qquad n \textbf{ divides } x$$

We will construct a function that given two numbers x and y computes a number z such that the divisors of z are exactly the common divisors of x and y. We will call z the **gcd** of x and y. Provided x and y are not both 0, the number z is in fact the greatest common divisor of x and y. We specify gcds with the following predicate:

$$yxyz \;:=\; \forall n.\; n \mid z \longleftrightarrow n \mid x \wedge n \mid y \qquad\qquad \textbf{gcd of } x \textbf{ and } y \textbf{ is } z$$

We start with some facts about divisibility.

Fact 12.6.1

1. $n \mid 0$.
2. $x \le y \to n \mid x \to (n \mid y \longleftrightarrow n \mid y - x)$.
3. $x > 0 \to n \mid x \to n \le x$.
4. $x > 0 \to y > 0 \to x \mid y \to y \mid x \to x = y$.
5. $(\forall n.\; n \mid x \longleftrightarrow n \mid y) \to x = y$.

Proof The first three claims have straightforward proofs unfolding the existential definition of divisibility. Claim (4) follows with (3) using antisymmetry. For Claim (5) we first destructure x and y. In case $x = 0$ and $y > 0$, we obtain with (1) that $\mathsf{S}y \mid y$, which is contradictory by (3). In case $x, y > 0$, we obtain $x = y$ with (4). ∎

Fact 12.6.2 The predicate γ is functional. Thus the gcd of two numbers is unique.

Proof We need to show $\forall xyzz'.\ \gamma xyz \to \gamma xyz' \to z = z'$. Follows with Fact 12.6.1 (5). ∎

Procedure

Our goal is the construction of a function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that

$$\forall xy.\ \gamma xy(fxy)$$

The underlying algorithm exploits the facts that the gcd of two numbers x and y is y if $x = 0$, and is the gcd of $x - y$ and y if $x \geq y$. Thus it suffices to subtract the smaller number from the larger number until one of the numbers is 0. The algorithm terminates since the sum of the two numbers is decreased.

Fact 12.6.3 (Recursion Rules)

1. $\gamma 0yy$.
2. $x \geq y \to \gamma(x - y)yz \to \gamma xyz$.
3. $\gamma xyz \to \gamma yxz$.
4. $\gamma x0x$.
5. $y \geq x \to \gamma x(y - x)z \to \gamma xyz$.

Proof Claims (1) and (2) follow with (1) and (2) of Fact 12.6.1. Claim (3) is obvious. Claims (4) and (5) follow with (1) and (2) and the symmetry rule (3). ∎

Fact 12.6.4 There is a function $\mathsf{Gcd} : \forall xy\,\Sigma z.\ \gamma xyz$.

Proof By binary size recursion on $x + y$ (Exercise 12.1.4) using the recursion rules from Fact 12.6.3 and case analysis on x, y, and $x \geq y \vee y > x$ as in Figure 12.2. ∎

Corollary 12.6.5 There is a function gcd such that $\gamma xy(\mathsf{gcd}\,xy)$.

Recursive Specification

Fact 12.6.6 A function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ satisfies $\forall xy.\ \gamma xy(fxy)$ if and only if it satisfies the recursive specification in Figure 12.2.

Proof Assume $\forall xy.\ \gamma xy(fxy)$. We need to show that f satisfies the four equations of the recursive specification. We show the final equation, the proofs of the other equations are similar. Let $y > x$. We show $f(\mathsf{S}x)(\mathsf{S}y) = f(\mathsf{S}x)(y - x)$. By functionality of γ (Fact 12.6.2) and the assumption it suffices to show

$$\gamma(\mathsf{S}x)(\mathsf{S}y)(f(x - y)(\mathsf{S}y))$$

$$
\begin{aligned}
f &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
f\,0\,y &= y \\
f\,(\mathsf{S}x)\,0 &= \mathsf{S}x \\
f\,(\mathsf{S}x)\,(\mathsf{S}y) &= \begin{cases} f\,(x-y)\,(\mathsf{S}y) & \text{if } x \geq y \\ f\,(\mathsf{S}x)\,(y-x) & \text{if } y > x \end{cases}
\end{aligned}
$$

Figure 12.2: Recursive specification of a gcd function

Follows by recursion rule (5) of Fact 12.6.3 and the assumption.

Assume f satisfies the recursive specification in Figure 12.2. We prove $\forall xy.\ \gamma xy(fxy)$ by binary size induction on $x + y$ and case analysis on x, y, and $x \geq y \vee y > x$. The proof obligations follow with the recursion rules from Fact 12.6.3 and the inductive hypothesis. ∎

Corollary 12.6.7 The function gcd satisfies the recursive specification in Figure 12.2.

Exercise 12.6.8 Specify, define and verify a function $\mathcal{L}(\mathsf{N}) \to \mathsf{N}$ that for a list of numbers yields a number whose divisors are the common divisors of the numbers in the list.

Exercise 12.6.9 Show that $\gamma x(\mathsf{S}y)z$ if and only if z is the greatest common divisor of x and $\mathsf{S}y$.

Exercise 12.6.10 There is a straightforward algorithm computing gcds with the remainder function. Let M be the remainder function from § 12.5. Prove the following:

a) $\gamma(Sy)(Mxy)z \to \gamma x(\mathsf{S}y)z$.

b) A function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfies $\forall xy.\ \gamma xy(fxy)$ if and only if it satisfies the equations

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

12.7 Discrete Inversion

For our third and final example we start from an informal specification and a straightforward recursive procedure solving the problem. We develop a formal relational specification and construct a function satisfying the relational specification

$$g : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$
$$gxk := \begin{cases} k & \text{if } x < f(\mathsf{S}k) \\ gx(\mathsf{S}k) & \text{if } f(\mathsf{S}k) \le x \end{cases}$$

Figure 12.3: Recursive specification of a discrete inversion function

using the recursion rules underlying the recursive procedure. As before we show that the relational specification and the recursive specification are equivalent.

Given a strictly increasing function $f : \mathsf{N} \to \mathsf{N}$

$$f0 < f1 < f2 < \cdots < fn < f(\mathsf{S}n) < \cdots$$

we will define an inverse function $g : \mathsf{N} \to \mathsf{N}$ such that $g(fn) = n$ for all n. This way we can invert functions like $\lambda n.2n$, $\lambda n.n^2$, and $\lambda n.2^n$ and thus compute discrete quotients, discrete roots, and discrete logarithms.

Given x, we want to compute the unique n such that $fn \le x < f(\mathsf{S}n)$. The number n exists if $f0 \le x$. We will compute n by incrementing a counter $k = 0, 1, 2, \ldots$ until $x < f(\mathsf{S}k)$. The final k is then the n we are looking for. We capture this method with the recursive specification shown in Figure 12.3. The recursion in the specification terminates since every recursion step decreases $x - k$. Note that the recursion does not change x.

We fix a function $f : \mathsf{N} \to \mathsf{N}$ that is strictly increasing (i.e., $\forall n,\ fn < f(\mathsf{S}n)$). We start by proving basic facts about f.

Fact 12.7.1 (Monotonicity)

1. $n < n' \to fn < fn'$.
2. $n < n' \longleftrightarrow fn < fn'$.
3. $n \le n' \longleftrightarrow fn \le fn'$.
4. $fn \le x \to f0 \le x$.
5. $n \le fn$.
6. $f(\mathsf{S}n) \le x \to x - \mathsf{S}n < x - n$. *termination property*

Proof Claim (1) follows by induction on n'. Both directions of Claim (2) follow with Claim (1), where direction $\leftarrow$ uses case analysis on $n < n' \vee n = n' \vee n' < n$. Claim (3) follows with Claim (1) and case analysis. Claim (4) follows with Claim (3). Claim (5) follows by induction on n. Claim (6) follows with Claim (5). ∎

Claim (6) of the fact provides the termination argument for the recursive specification in Figure 12.3.

We define a predicate

$$\varphi x n \;:=\; f n \le x < f(\mathsf{S} n)$$

Fact 12.7.2

1. $\varphi(f n)n$.
2. $\varphi(f n)k \to n = k$.
3. $\varphi x n \to \varphi x n' \to n = n'$. *functionality*
4. $\varphi x n \to f 0 \le x$.

Proof Claim (1) is trivial. Claim (2) follows with the monotonicity statements of Fact 12.7.1. Claim (3) follows by case analysis on $n = n' \vee \mathsf{S} n \le n' \vee \mathsf{S} n' \le n$ and direction $\to$ of Fact 12.7.1 (3). Claim (4) follows with Fact 12.7.1 (4). ∎

We define the explicit specification corresponding to the recursive specification in Figure 12.3 as follows:

$$\psi x k n \;:=\; \text{IF } \ulcorner x < f(\mathsf{S} k) \urcorner \text{ THEN } k = n \text{ ELSE } \varphi x n$$

We will show $\psi x k (g k)$ for every function g satisfying the recursive specification in Figure 12.3.

Fact 12.7.3

1. $\psi x k n \to \psi x k n' \to n = n'$. *functionality*
2. $f k \le x \to \psi x k n \to \varphi x n$.
3. $x < f(\mathsf{S} k) \to \psi x k k$. *base rule*
4. $f(\mathsf{S} k) \le x \to \psi x (\mathsf{S} k) n \to \psi x k n$. *step rule*

Proof Claim (1) follows with the functionality of φ. Claims (2) and (3) are straightforward. Claim (4) follows withClaim (2). ∎

Fact 12.7.4 There is a function $\mathsf{Inv} : \forall k\, \Sigma n.\ \psi x k n$.

Proof By size recursion on $x - k$ using the base and step rule of Fact 12.7.3 and the termination property of Fact 12.7.1. ∎

Corollary 12.7.5 There is a function $\mathsf{inv} : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that $\forall x k.\ \psi x k (\mathsf{inv}\, x k)$.

Fact 12.7.6 $\varphi x n \;\longleftrightarrow\; \mathsf{inv}\, x 0 \wedge f 0 \le x$.

Proof Follows with Corollary 12.7.5, Fact 12.7.1 (4) and Fact 12.7.2 (3). ∎

Corollary 12.7.7 (Inversion) $\mathsf{inv}\,(f n) 0 = n$.

Proof Follows with Fact 12.7.6 and Fact 12.7.2 (1). ∎

Fact 12.7.8 (Recursive Specification) A function g satisfies $\forall xk.\ \psi xk(gxk)$ if and only if it satisfies the recursive specification in Figure 12.3.

Proof Assume $\forall xk.\ \psi xk(gxk)$. We need to show that g satisfies the equation of the recursive specification. By the functionality of ψ it suffices to show $\psi xk(\mathsf{INV}xkg)$ where INV is the step function for the recursive specification. Follows with case analysis and the base and step rule of Fact 12.7.3.

Assume g satisfies the recursive specification in Figure 12.1. We prove $\forall k.\ \psi xk(gxk)$ by size induction on $x - k$ and case analysis using the base and step rule of Fact 12.7.3. The termination property of Fact 12.7.1 is needed for the inductive hypothesis. ∎

Exercise 12.7.9 Define functions g as follows. In each case prove correctness.

a) $\forall n.\ g(n^3) = n$. *discrete square roots*

b) $\forall n.\ g(3^n) = n$. *discrete logarithms*

c) $\forall n.\ g(3 \cdot n) = n$. *discrete divisors*

12.8 Notes

We have studied three computational problems for numbers. In each case, we have constructed a function satisfying a relational specification using size recursion. Size recursion was used since natural formulations with structural recursion do not exist.

In each case, we have used size recursion with an informative return type. This way we could uses the size recursion operator without knowing its defining equations.

In each case, there was a natural recursion scheme providing for the construction of the function satisfying the relational specification using the size recursion operator. In contrast to execution-oriented programming languages, where arbitrary recursive procedures can be constructed without specifications, we had to come up with specifications before we could construct the desired functions.

In type theory, terminating recursive procedures can always be captured with equational specifications of functions. While equational specifications capture the recursion scheme, they do not provide for the existence of the specified function.

We mention that type theory can describe arbitrary recursive procedures with indexed inductive predicates. Termination is not an issue with this method since predicates may be partial.

As a take-home message we conclude that the construction of functions in type theory may take work and insight, although we know a terminating recursive procedure computing the function. Work is necessary if the recursion underlying the procedure is not structural. If the termination of the procedure can be argued with a size function, and the function computed by the procedure can be specified independently in type theory, the function can be constructed using size recursion as explained in this chapter.

13 Existential Witness Operators

In this chapter, we will define an existential witness operator that for a decidable predicate on numbers obtains a satisfying number given a satisfiability proof:

$$W : \forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\ \mathcal{D}(pn)) \to (\exists n.\ pn) \to (\Sigma n.\ pn)$$

The interesting point about an existential witness operator is the fact that it obtains a witness that can be used computationally from a propositional satisfiability proof. Existential witness operators are required for various computational constructions.

Given that the elim restriction disallows computational access to the witness of an existential proof, the definition of a witness operator is not obvious. Our definition will in fact rely on higher-order structural recursion, a feature of inductive definitions we have not used before. The key idea is the use of an inductive transfer predicate

$$T(n : \mathsf{N}) : \mathbb{P} \ ::= \ C\,(\neg pn \to T(\mathsf{S}n))$$

featuring a recursion through the right-hand side of a function type. Derivations of Tn thus carry a continuation function $\varphi : \neg pn \to T(\mathsf{S}n)$ providing a structurally smaller derivation $\varphi h : T(\mathsf{S}n)$ for every proof $h : \neg pn$. By recursing on a derivation of $T0$ we will be able to define a function performing a linear search $n = 0, 1, 2, \ldots$ until pn holds. Since $T0$ is a proposition, we can construct a derivation of $T0$ in propositional mode using the witness from the proof of $\exists n.pn$.

13.1 Recursive Transfer Predicate

Recall from § 6.4 that a transfer predicate is an inductive predicate exempted from the elim restriction. Derivations of propositions obtained with a transfer predicate can be decomposed in computation mode although they have been constructed in proof mode. Recursive transfer predicates thus provide for computational recursion.

We fix a predicate $p : \mathsf{N} \to \mathbb{P}$ and define a inductive predicate T as follows:

$$T(n : \mathsf{N}) : \mathbb{P} \ ::= \ C\,(\neg pn \to T(\mathsf{S}n))$$

The argument of the single proof constructor C is a function

$$\varphi : \neg pn \to T(\mathsf{S}n)$$

counting as a proof since T is a predicate. Thus T is in fact a transfer predicate. We will refer to φ as the *continuation function* of a derivation. The important point now is the fact that the continuation function of a derivation of type Tn yields a structurally smaller derivation $\varphi h : T(\mathsf{S}n)$ for every proof $h : \neg pn$. Since the recursion passes through the right constituent of a function type we speak of a **higher-order recursion**. It is the flexibility coming with higher-order recursion that makes the definition of a witness operator possible. We remark that Coq's type theory admits recursion only through the right-hand side of function types, a restriction known as **strict positivity condition**.

We remark that the parameter n of T is **nonuniform**. While T can be defined with the parameter p abstracted out (e.g., as a section variable in Coq), the parameter n cannot be abstracted out since the application $T(\mathsf{S}n)$ appears in the argument type of the proof constructor.

Exercise 13.1.1 (Padding) Given a derivation $d : Tn$, we can always obtain a derivation of Tn that starts with as many constructors we would like to have. The basic padding step is

$$\begin{aligned} Dd &:= \text{LET } (\varphi) := d \text{ IN } \varphi \\ \mathsf{pad}\ d &:= C(\lambda a.\, Dda) \end{aligned}$$

Define a function $\mathsf{pad} : \mathsf{N} \to \forall n.\, Tn \to Tn$ such that $\mathsf{pad}\, k$ pads a derivation with k constructors. For instance,

$$\mathsf{pad}\, 3\, d = C(\lambda a.\, C(\lambda b.\, C(\lambda c.\, D(D(Dda)b)c)))$$

should hold by definitional equality. Note that n is treated as an implicit argument of the constructor C. That padding is possible appears to be a distinctive feature of higher-order recursion.

13.2 Definition of Existential Witness Operator

We now assume that p is a decidable predicate on numbers. We will define an existential witness operator

$$W : (\exists n.pn) \to \Sigma n.\, pn$$

using two functions

$$\begin{aligned} W' &: \ \forall n.\, Tn \to \Sigma n.\, pn \\ V &: \ \forall n.\, pn \to T0 \end{aligned}$$

The idea is to first obtain a derivation $d : T0$ using V and the witness of the proof of $\exists n.pn$, and then obtain a computational witness using W' and the derivation $d : T0$.

We define W' by recursion on Tn:

$$W' : \forall n.\, Tn \to \Sigma k.pk$$

$$W'\, n\, (C\varphi) \;:=\; \begin{cases} \mathsf{E}\, n\, h & \text{if } h : pn \\ W'\, (\mathsf{S}n)\, (\varphi h) & \text{if } h : \neg pn \end{cases}$$

Note that the defining equation of W' makes use of the higher-order recursion coming with Tn. The recursion is admissible since every derivation φh counts as structurally smaller than the derivation $C\varphi$. Coq's type theory is designed such that higher-order structural recursion always terminates.

It remains to define a function $V : \forall n.\, pn \to T0$. Given the definition of T, we have

$$\forall n.\, pn \to Tn \tag{13.1}$$

$$\forall n.\, T(\mathsf{S}n) \to Tn \tag{13.2}$$

Using recursion on n, function (13.2) yields a function

$$\forall n.\, Tn \to T0 \tag{13.3}$$

Using function (13.1), we have a function $V : \forall n.\, pn \to T0$ as required.

Theorem 13.2.1 (Existential witness operator)
There is a function $W : \forall p^{\mathsf{N}\to\mathbb{P}}.\, (\forall n.\, \mathcal{D}(pn)) \to (\exists n.\, pn) \to (\Sigma n.\, pn)$.

Proof Using V we obtain a derivation $d : T0$ from the witness of the proof of $\exists n.pn$. There is no problem with the elim restriction since $T0$ is a proposition. Now W' yields a computational witness for p. ∎

Exercise 13.2.2 Point out where in the defining equation of W' it is exploited that T is a transfer predicate (i.e., the elim restriction does not apply to T).

Exercise 13.2.3 Define W' with FIX and MATCH. Note that FIX must be given a leading argument n so that the recursive function can receive the type $\forall n.\, Tn \to \Sigma k.\, pk$ accommodating the recursive application for $\mathsf{S}n$.

13.3 More Existential Witness Operators

Fact 13.3.1 (Existential least witness operator)
There is a function $\forall p^{\mathsf{N}\to\mathbb{P}}.\, (\forall n.\, \mathcal{D}(pn)) \to (\exists n.\, pn) \to (\Sigma n.\, \mathsf{least}\, pn)$.

Proof There are two ways to construct the operator using W. Either we use Fact 11.7.3 that gives us a least witness for a witness, or Fact 11.7.5 and Corollary 11.7.4 that tell us that $\mathsf{least}\, p$ is a decidable and satisfiable predicate. ∎

Corollary 13.3.2 (Binary existential witness operator)
There is a function $\forall p^{\mathsf{N}\to\mathsf{N}\to\mathbb{P}}.\ (\forall xy.\mathcal{D}(pxy)) \to (\exists xy.pxy) \to (\Sigma xy.pxy)$.

Proof Follows with W and the paring bijection from Chapter 7. The trick is to use W with $\lambda n.\ p(\pi_1(Dn))(\pi_2(Dn))$. ∎

Corollary 13.3.3 (Disjunctive existential witness operator)
Let p and q be decidable predicates on numbers.
Then there is a function $(\exists n.pn) \vee (\exists n.qn) \to (\Sigma n.pn) + (\Sigma n.qn)$.

Proof Use W with the predicate $\lambda n.pn \vee qn$. ∎

The following fact was discovered by Andrej Dudenhefner in March 2020.

Fact 13.3.4 (Discreteness via step-indexed equality decider)
Let $f^{X\to X\to\mathsf{N}\to\mathsf{B}}$ be a function such that $\forall xy.\ x = y \longleftrightarrow \exists n.\ fxyn = \mathbf{T}$.
Then X has an equality decider.

Proof We prove $\mathcal{D}(x = y)$ for fixed $x, y : X$. Using the witness operator we obtain n such that $fxxn = \mathbf{T}$. If $fxyn = \mathbf{T}$, we have $x = y$. If $fxyn = \mathbf{F}$, we have $x \neq y$. ∎

Exercise 13.3.5 (Infinite path)
Let $p : \mathsf{N} \to \mathsf{N} \to \mathbb{P}$ be a decidable predicate that is total: $\forall x\exists y.\ pxy$.

a) Define a function $f : \mathsf{N} \to \mathsf{N}$ such that $\forall x.\ px(fx)$.

b) Given x, define a function $f : \mathsf{N} \to \mathsf{N}$ such that $f0 = x$ and $\forall n.\ p(fn)(f(\mathsf{S}n))$. We may say that f describes an infinite path starting from x in the graph described by the edge predicate p.

Exercise 13.3.6 Let $f : \mathsf{N} \to \mathsf{B}$. Prove the following:

a) $(\exists n.\ fn = \mathbf{T}) \Leftrightarrow (\Sigma n.\ fn = \mathbf{T})$.

b) $(\exists n.\ fn = \mathbf{F}) \Leftrightarrow (\Sigma n.\ fn = \mathbf{F})$.

Exercise 13.3.7 Let p be a decidable predicate on numbers. Define a function $\forall n.\ Tn \to \Sigma k.\ k \geq n \wedge pk$.

Exercise 13.3.8 Construct a witness operator $(\exists x.\ px) \to (\Sigma x.\ px)$ for decidable predicates p on booleans. No transfer predicate is needed since there are only two candidates for a witness.

13.4 Eliminator and Existential Characterization

We define an eliminator for the transfer predicate:

$$E_T : \forall q^{\mathsf{N}\to\mathbb{T}}.\ (\forall n.\ (\neg pn \to q(\mathsf{S}n)) \to qn) \to \forall n.\ Tn \to qn$$
$$E_T\, q\, f\, n\, (C\varphi) \ :=\ f n (\lambda h.\, E_T\, q\, f\, (\mathsf{S}n)\, (\varphi h))$$

The eliminator provides for inductive proofs on derivations of T. That the inductive hypothesis $q(\mathsf{S}n)$ in the type of f is guarded by $\neg pn$ ensures that it can be obtained with recursion through φ.

We remark that when translating the equational definition of E_T to a computational definition with FIX and MATCH, the recursive abstraction with FIX must be given a leading argument n so that the recursive function can receive the type $\forall n.\ Tn \to pn$, which is needed for the recursive application, which is for $\mathsf{S}n$ rather than n.

Exercise 13.4.1 Define W' with the eliminator E_T for T.

Exercise 13.4.2 (Existential characterization) Prove the following facts about the transfer predicate T.

a) $pn \to Tn$.

b) $T(\mathsf{S}n) \to Tn$.

c) $T(k+n) \to Tn$.

d) $Tn \to T0$.

e) $pn \to T0$.

f) $Tn \longleftrightarrow \exists k.\ k \geq n \wedge pk$.

Hints: Direction $\to$ of (f) follows with induction on T using the eliminator E_T. Part (c) follows with induction on k. The rest follows without inductions, mostly using previously shown claims.

Exercise 13.4.3 The eliminator we have defined for T is not the strongest one. One can define a stronger eliminator where the target type depends on both n and a derivation $d : Tn$. This eliminator makes it possible to prove properties of a linear search function $\forall n.\ Tn \to \mathsf{N}$ with a noninformative target type.

13.5 Notes

With the transfer predicate T we have seen an inductive predicate that goes far beyond the inductive definitions we have seen so far. The proof constructor of T employs higher-order structural recursion through the right-hand side of a function

type. Higher-order structural recursion greatly extends the power of structural recursion. Higher-order structural recursion means that an argument of a recursive constructor is a function that yields a structurally smaller value for every argument. That higher-order structural recursions always terminates is a basic design feature of Coq's type theory.

14 Lists

We study inductive list types providing a recursive representation for finite sequences over a base type. Besides numbers, lists are the most important recursive 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 recursive predicates for membership and disjointness of lists, and also for repeating and nonrepeating lists. We will study nonrepeating lists and relate non-repetition to cardinality of lists.

14.1 Inductive Definition

A list represents a finite sequence $[x_1, \ldots, x_n]$ 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 $[x_1, \ldots, x_n]$ the list $[x, x_1, \ldots, x_n]$. Given a list cons x A, we call x the **head** and A the **tail** of the list. Given a list $[x_1, \ldots, x_n]$, we call n the **length** of the list and $x_1, \ldots, x_n$ 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

$$\mathcal{L}(X : \mathbb{T}) : \mathbb{T} \ ::= \ \mathsf{nil} \mid \mathsf{cons}\,(X, \mathcal{L}(X))$$

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 omit 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 similar to what we have seen for numbers. We define the standard **eliminator for lists** as follows:

$$\begin{aligned} \mathsf{E}_{\mathcal{L}} :\ & \forall X^{\mathbb{T}}\, p^{\mathcal{L}(X)\to\mathbb{T}}.\ p\,[] \to (\forall x A.\ pA \to p(x :: A)) \to \forall A.\, pA \\ \mathsf{E}_{\mathcal{L}}\, X p a f\, [] &:= a \\ \mathsf{E}_{\mathcal{L}}\, X p a f\, (x :: A) &:= f x A (\mathsf{E}_{\mathcal{L}}\, X p a f A) \end{aligned}$$

The eliminator provides for inductive proofs, recursive function definitions, and structural case analysis.

Fact 14.1.1 (Constructor laws)

1. $[] \neq x :: A$ (disjointness)
2. $x :: A = y :: B \to x = y$ (injectivity)
3. $x :: A = y :: B \to A = B$ (injectivity)
4. $x :: A \neq A$ (progress)

Proof The proofs are similar to the corresponding proofs for numbers (Fact 11.1.1). Claim (4) corresponds to $\mathsf{S}n \neq n$ and follows by induction on A with x quantified. ∎

Fact 14.1.2 (Discreteness) If X is a discrete type, then $\mathcal{L}(X)$ is a discrete type: $\mathcal{E}(X) \to \mathcal{E}(\mathcal{L}(X))$.

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 14.1.1. In case both lists are nonempty with heads x and y, an additional case analysis on $x = y$ is needed. ∎

Exercise 14.1.3 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ \mathcal{D}(A = [])$.

Exercise 14.1.4 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ (A = []) + \Sigma x B.\ A = x :: B$.

14.2 Basic Operations

We introduce three basic operations on lists, which yield the length of a list, concatenate two lists, and apply a function to every position of a list:

$$\begin{aligned} \mathsf{len}\,[x_1,\ldots,x_n] &= n && \textbf{length}\\ [x_1,\ldots,x_m] +\!\!+ [y_1,\ldots,y_n] &= [x_1,\ldots,x_m,y_1,\ldots,y_n] && \textbf{concatenation}\\ f@\,[x_1,\ldots,x_n] &= [f@x_1,\ldots,f@x_n] && \textbf{map} \end{aligned}$$

Formally, we define the operations as recursive functions:

$$\begin{aligned} \mathsf{len} :\ & \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathsf{N}\\ \mathsf{len}\,[]\ &:=\ 0\\ \mathsf{len}\,(x :: A)\ &:=\ S\,(\mathsf{len}\,A) \end{aligned}$$

$$\begin{aligned} +\!\!+ :\ & \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathcal{L}(X) \to \mathcal{L}(X)\\ []\ +\!\!+\ B\ &:=\ B\\ (x :: A)\ +\!\!+\ B\ &:=\ x :: (A +\!\!+ B) \end{aligned}$$

$$\begin{aligned} @ :\ & \forall XY^{\mathbb{T}}.\ (X \to Y) \to \mathcal{L}(X) \to \mathcal{L}(Y)\\ f@\,[]\ &:=\ []\\ f@(x :: A)\ &:=\ fx :: (f@A) \end{aligned}$$

Note that in all three definitions we accommodate X as an implicit argument for readability.

Fact 14.2.1

1. $A +\!\!+ (B +\!\!+ C) = (A +\!\!+ B) +\!\!+ C$ (associativity)
2. $A +\!\!+ [] = A$
3. $\mathsf{len}\,(A +\!\!+ B) = \mathsf{len}\,A + \mathsf{len}\,B$
4. $\mathsf{len}\,(f@A) = \mathsf{len}\,A$
5. $\mathsf{len}\,A = 0 \longleftrightarrow A = []$

Proof The equations follow by induction on A. The equivalence follows by case analysis on A. ∎

14.3 Membership

Informally, we may characterize **membership** in lists with the equivalence

$$x \in [x_1\,,\ \ldots\ ,\ x_n] \ \longleftrightarrow\ \ x = x_1 \vee \cdots \vee x = x_n \vee \bot$$

Formally, we define the **membership predicate** by structural recursion on lists:

$$\begin{aligned}(\in) :\ & \forall X^{\mathbb{T}}.\ X \to \mathcal{L}(X) \to \mathbb{P}\\ (x \in [])\ &:=\ \bot\\ (x \in y :: A)\ &:=\ (x = y \vee x \in A)\end{aligned}$$

We treat the type argument X of the membership predicate as implicit argument. If $x \in A$, we say that x is an **element** of A.

Fact 14.3.1 (Decidable Membership)
Membership in lists over discrete types is decidable:
$\forall X^{\mathbb{T}}.\ \mathcal{E}(X) \to \forall x^X \forall A^{\mathcal{L}(X)}.\ \mathcal{D}(x \in A)$.

Proof By induction on A. ∎

Recall that bounded quantification over numbers preserves decidability (Fact 11.6.6). Similarly, quantification over the elements of a list preserves decidability.

Fact 14.3.2 (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. $(\forall x.\ \mathcal{D}(px)) \to \mathcal{D}(\exists x.\ x \in A \wedge px)$.
3. $(\forall x.\ \mathcal{D}(px)) \to (\Sigma x.\ x \in A \wedge px) + (\forall x.\ x \in A \to \neg px)$.

Proof By induction on A. ∎

Fact 14.3.3 (Membership laws)

1. $x \in A + B \ \longleftrightarrow\ x \in A \vee x \in B$.
2. $x \in f@A \ \longleftrightarrow\ \exists a.\ a \in A \wedge x = fa$.

Proof By induction on A. ∎

Membership can also be characterized with existential quantification and concatenation. We speak of the explicit characterization of list membership.

Fact 14.3.4 (Explicit Characterization)
$x \in A \ \longleftrightarrow\ \exists A_1 A_2.\ A = A_1 + x :: A_2$.

Proof Direction → follows by induction on A. Direction ← follows by induction on A_1. ∎

Fact 14.3.5 (Factorization)
For every discrete type X there is a function
$\forall x^X A^{\mathcal{L}(X)}.\ x \in A \to \Sigma A_1 A_2.\ A = A_1 + x :: A_2$.

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 14.3.6
Define a function $\delta : \mathcal{L}(\mathcal{O}(X)) \to \mathcal{L}(X)$ such that $x \in \delta A \longleftrightarrow {}^\circ x \in A$.

Exercise 14.3.7 (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:

$$\mathsf{sum}\, A > \mathsf{len}\, A \to \Sigma x.\, x \in A \wedge x \geq 2$$

First define the function sum.

14.4 List Inclusion and List Equivalence

We may see a list as a representation of a finite set. 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:

$$\begin{aligned} A \subseteq B &:= \forall x.\, x \in A \to x \in B \\ A \equiv B &:= A \subseteq B \wedge B \subseteq A \end{aligned}$$

Note that two lists are equivalent if and only if they represent the same set.

Fact 14.4.1 List inclusion $A \subseteq B$ is reflexive and transitive. List equivalence $A \equiv B$ is reflexive, symmetric, and transitive.

Fact 14.4.2 We have the following properties for membership, inclusion, and equivalence of lists.

$$x \notin [] \qquad x \in [y] \longleftrightarrow x = y$$
$$[] \subseteq A \qquad A \subseteq [] \to A = []$$
$$x \in y :: A \to x \neq y \to x \in A \qquad x \notin y :: A \to x \neq y \wedge x \notin A$$
$$A \subseteq B \to x \in A \to x \in B \qquad A \equiv B \to x \in A \longleftrightarrow x \in B$$
$$A \subseteq B \to x :: A \subseteq x :: B \qquad A \equiv B \to x :: A \equiv x :: B$$
$$A \subseteq B \to A \subseteq x :: B \qquad x :: A \subseteq B \longleftrightarrow x \in B \wedge A \subseteq B$$
$$x :: A \subseteq x :: B \to x \notin A \to A \subseteq B \qquad x :: A \subseteq [y] \longleftrightarrow x = y \wedge A \subseteq [y]$$
$$x :: A \equiv x :: x :: A \qquad x :: y :: A \equiv y :: x :: A$$
$$x \in A \to A \equiv x :: A$$
$$x \in A +\!\!+ B \longleftrightarrow x \in A \vee x \in B$$
$$A \subseteq A' \to B \subseteq B' \to A +\!\!+ B \subseteq A' +\!\!+ B' \qquad A +\!\!+ B \subseteq C \longleftrightarrow A \subseteq C \wedge B \subseteq C$$

Proof Except for the membership fact for concatenation, which already appeared as Fact 14.3.3, all claims have straightforward proofs not using induction. ∎

Fact 14.4.3 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 14.3.1) and bounded quantification preserves decidability (Fact 14.3.2). ∎

14.5 Setoid Rewriting

It is possible to rewrite a claim or an assumption in a proof goal with a propositional equivalence $P \longleftrightarrow P'$ 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 14.5.1 (Compatibility laws for propositional equivalence)
Let $P \longleftrightarrow P'$ and $Q \longleftrightarrow Q'$. Then:

$$P \wedge Q \longleftrightarrow P' \wedge Q' \qquad P \vee Q \longleftrightarrow P' \vee Q' \qquad (P \to Q) \longleftrightarrow (P' \to Q')$$
$$\neg P \longleftrightarrow \neg P' \qquad (P \longleftrightarrow Q) \longleftrightarrow (P' \longleftrightarrow Q')$$

Fact 14.5.2 (Compatibility laws for list equivalence)
Let $A \equiv A'$ and $B \equiv B'$. Then:

$$x \in A \longleftrightarrow x \in A' \qquad A \subseteq B \longleftrightarrow A' \subseteq B' \qquad A \equiv B \longleftrightarrow A' \equiv B'$$
$$A +\!\!+ B \equiv A' +\!\!+ B' \qquad f@A \equiv f@A' \qquad A \mid f \equiv A' \mid f$$
$$x :: A \equiv x :: A'$$

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 14.5.3 Which of the compatibility laws are needed to justify rewriting the claim $\neg(x \in y :: (f@A) +\!\!+ B)$ with the equivalence $A \equiv A'$?

14.6 Element Removal

Element removal for lists is an important operation that we will need for results about nonrepeating lists and cardinality. We assume a discrete type X and define a function $A \setminus x$ for **element removal** as follows:

$$\begin{aligned} \setminus :\ & \mathcal{L}(X) \to X \to \mathcal{L}(X) \\ & [] \setminus \_ := [] \\ & (x :: A) \setminus y := \text{IF } \ulcorner x = y \urcorner \text{ THEN } A \setminus y \text{ ELSE } x :: (A \setminus y) \end{aligned}$$

Fact 14.6.1

1. $x \in A \setminus y \longleftrightarrow x \in A \wedge x \neq y$
2. $\mathsf{len}\,(A \setminus x) \leq \mathsf{len}\,A$
3. $x \in A \to \mathsf{len}\,(A \setminus x) < \mathsf{len}\,A$.
4. $x \notin A \to A \setminus x = A$

Proof By induction on A. ∎

Exercise 14.6.2 Prove $x \in A \to A \equiv x :: (A \setminus x)$.

Exercise 14.6.3 Prove the following equations, which are useful in proofs:

1. $(x :: A) \setminus x = A \setminus x$
2. $x \neq y \to (y :: A) \setminus x = y :: (A \setminus x)$

14.7 Nonrepeating 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 nonrepeating. Formally, we define **repeating lists** over a base type X with a recursive predicate:

$$\begin{aligned} \mathsf{rep} : \mathcal{L}(X) &\to \mathbb{P} \\ \mathsf{rep}\,[] &:= \bot \\ \mathsf{rep}\,(x :: A) &:= x \in A \vee \mathsf{rep}\,A \end{aligned}$$

Fact 14.7.1 (Characterization)
For every list A over a discrete type we have:
$\mathsf{rep}\,A \longleftrightarrow \exists x A_1 A_2.\ A = A_1 +\!\!+ x :: A_2 \wedge x \in A_2$.

Proof By induction on $\mathsf{rep}\,A$ using Fact 14.3.4. ∎

We also define a recursive predicate for nonrepeating lists over a base type X:

$$\begin{aligned} \mathsf{nrep} : \mathcal{L}(X) &\to \mathbb{P} \\ \mathsf{nrep}\,[] &:= \top \\ \mathsf{nrep}\,(x :: A) &:= x \notin A \wedge \mathsf{nrep}\,A \end{aligned}$$

Theorem 14.7.2 (Partition) Let A be a list over a discrete type. Then:

1. $\mathsf{rep}\,A \to \mathsf{nrep}\,A \to \bot$ (disjointness)
2. $\mathsf{rep}\,A + \mathsf{nrep}\,A$ (exhaustiveness)

Proof Both claims follow by induction on A. Discreteness is only needed for the second claim. The second claim needs decidability of membership (Fact 14.3.1) for the cons case. ∎

Corollary 14.7.3 Let A be a list over a discrete type. Then:

1. $\mathcal{D}(\mathsf{rep}\,A)$ and $\mathcal{D}(\mathsf{nrep}\,A)$.
2. $\mathsf{rep}\,A \longleftrightarrow \neg\mathsf{nrep}\,A$ and $\mathsf{nrep}\,A \longleftrightarrow \neg\mathsf{rep}\,A$.

Fact 14.7.4 (Equivalent nonrepeating list)
For every list over a discrete type one can obtain an equivalent nonrepeating list:
$\forall A\, \Sigma B.\ B \equiv A \wedge \mathsf{nrep}\,B$.

Proof By induction on A. For $x :: A$, let B be the list obtained for A with the inductive hypothesis. If $x \in A$, B has the required properties for $x :: A$. If $x \notin A$, $x :: B$ has the required properties for $x :: A$. ∎

The next fact formulates a key property concerning the cardinality of lists (number of different elements). It is carefully chosen so that it provides a powerful building block for further results (Corollary 14.7.6). Finding this fact took effort. To get the taste of it, try to prove that equivalent nonrepeating lists have equal length without looking at our development.

Fact 14.7.5 (Discriminating element)
Every nonrepeating list over a discrete type contains for every shorter list an element not in the shorter list: $\forall AB.\ \mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \Sigma z.\ z \in A \wedge z \notin B$.

Proof By induction on A with B quantified. For $x :: A$ and $x \in B$, one uses the inductive hypothesis for A and $B \setminus x$, as justified by Fact 14.6.1 (3). ∎

Corollary 14.7.6 Let A and B be lists over a discrete type X. Then:

1. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,A \le \mathsf{len}\,B$.
2. $\mathsf{nrep}\,A \to \mathsf{nrep}\,B \to A \equiv B \to \mathsf{len}\,A = \mathsf{len}\,B$.
3. $A \subseteq B \to \mathsf{len}\,B < \mathsf{len}\,A \to \mathsf{rep}\,A$.
4. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \le \mathsf{len}\,A \to \mathsf{nrep}\,B$.
5. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \le \mathsf{len}\,A \to B \equiv A$.

Proof Interestingly, all claims follow without induction from Facts 14.7.5, 14.7.1, and 14.7.3.

For (1), assume $\mathsf{len}\,A > \mathsf{len}\,B$ and derive a contradiction with Fact 14.7.5.

Claims (2) and (3) follow from Claim (1), where for (3) we assume $\mathsf{nrep}\,A$ and derive a contradiction (justified by Corollary 14.7.3).

For (4), we assume $\mathsf{rep}\,B$ and derive a contradiction (justified by Corollary 14.7.3). By Fact 14.7.1, We obtain a list B' such that $A \subseteq B'$ and $\mathsf{len}\,B' < \mathsf{len}\,A$. Contradiction with (1).

For (5), it suffices to show $B \subseteq A$. We assume $x \in B$ and $x \notin A$ and derive a contradiction with $B \setminus x$ and Fact 14.7.5. ∎

Exercise 14.7.7 Prove the following facts about map and nonrepeating lists:

a) $\mathsf{injective}\,f \to \mathsf{nrep}\,A \to \mathsf{nrep}\,(f@A)$.

b) $\mathsf{nrep}\,(f@A) \to x \in A \to x' \in A \to fx = fx' \to x = x'$.

Exercise 14.7.8 (Injectivity-surjectivity agreement) Let X be a discrete type and A be a list containing all elements of X. Prove that a function $X \to X$ is surjective if and only if it is injective.

This is an interesting exercise. It can be stated as soon as membership in lists is defined. To solve it, however, one needs properties of length, map, element removal, and nonrepeating lists. If one doesn't know these notions, the exercise

makes an interesting project since one has to invent these notions. Our solution uses Corollary 14.7.6 and Exercise 14.7.7.

Exercise 14.7.9 Let A be a list over a discrete type.
Prove $\mathsf{rep}\,A \to \Sigma x A_1 A_2 A_3.\ A = A_1 +\!\!+ x :: A_2 +\!\!+ x :: A_3$.

Exercise 14.7.10 (Partition) The proof of Corollary 14.7.3 is straightforward and follows a general scheme. Let P and Q be propositions such that $P \to Q \to \bot$ and $P + Q$. Prove $\mathsf{dec}\,P$ and $P \longleftrightarrow \neg Q$. Note that $\mathsf{dec}\,Q$ and $Q \longleftrightarrow \neg P$ follow by symmetry.

Exercise 14.7.11 (Even and Odd) Define recursive predicates even and odd on numbers and show that they partition the numbers: $\mathsf{even}\,n \to \mathsf{odd}\,n \to \bot$ and $\mathsf{even}\,n + \mathsf{odd}\,n$.

Exercise 14.7.12 Define a function $\mathsf{seq} : \mathsf{N} \to \mathsf{N} \to \mathcal{L}(\mathsf{N})$ for which you can prove the following:

a) $\mathsf{seq}\,2\,5 = [2,3,4,5,6]$

b) $\mathsf{seq}\,n\,(\mathsf{S}k) = n :: \mathsf{seq}\,(\mathsf{S}n)\,k$

c) $\mathsf{len}\,(\mathsf{seq}\,nk) = k$

d) $x \in \mathsf{seq}\,nk \longleftrightarrow n \le x < n + k$.

e) $\mathsf{nrep}\,(\mathsf{seq}\,nk)$

14.8 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. Formally, we may say that the cardinality of a list is the length of an equivalent nonrepeating list. This characterization is justified since equivalent nonrepeating lists have equal length (Corollary 14.7.6 (3)), and every list is equivalent to a non-repeating list (Fact 14.7.4).

We assume that lists are taken over a discrete type X and define a **cardinality function** as follows:

$$\begin{aligned}
&\mathsf{card} : \mathcal{L}(X) \to \mathsf{N}\\
&\qquad \mathsf{card}\,[] \;:=\; 0\\
&\mathsf{card}(x :: A) \;:=\; \text{IF } \ulcorner x \in A \urcorner \text{ THEN } \mathsf{card}\,A \text{ ELSE } \mathsf{S}(\mathsf{card}\,A)
\end{aligned}$$

Note that we write $\ulcorner x \in A \urcorner$ for the application of the membership decider provided by Fact 14.3.1. We prove that the cardinality function agrees with the cardinalities provided by equivalent nonrepeating lists.

Fact 14.8.1 (Cardinality)

1. $\forall A\,\Sigma B.\ B \equiv A \wedge \mathsf{nrep}\,B \wedge \mathsf{len}\,B = \mathsf{card}\,A$.
2. $\mathsf{card}\,A = n \longleftrightarrow \exists B.\ B \equiv A \wedge \mathsf{nrep}\,B \wedge \mathsf{len}\,B = n$.

Proof Claim 1 follows by induction on A. Claim 2 follows with Claim 2 and Corollary 14.7.6 (3). ∎

Corollary 14.8.2

1. $\mathsf{card}\,A \le \mathsf{len}\,A$
2. $A \subseteq B \to \mathsf{card}\,A \le \mathsf{card}\,B$
3. $A \equiv B \to \mathsf{card}\,A = \mathsf{card}\,B$.
4. $\mathsf{rep}\,A \longleftrightarrow \mathsf{card}\,A < \mathsf{len}\,A$ (pigeonhole)
5. $\mathsf{nrep}\,A \longleftrightarrow \mathsf{card}\,A = \mathsf{len}\,A$
6. $x \in A \longleftrightarrow \mathsf{card}\,A = \mathsf{S}(\mathsf{card}(A \setminus x))$

Proof All facts follow without induction from Fact 14.8.1, Corollary 14.7.6, and Corollary 14.7.3. ∎

Exercise 14.8.3 (Cardinality predicate) We define a recursive cardinality predicate:

$$\begin{aligned}
\mathsf{Card}:\ & \mathcal{L}(X) \to X \to \mathbb{P} \\
\mathsf{Card}\,[]\,0 &:= \top \\
\mathsf{Card}\,[]\,(\mathsf{S}n) &:= \bot \\
\mathsf{Card}\,(x :: A)\,0 &:= \bot \\
\mathsf{Card}\,(x :: A)\,(\mathsf{S}n) &:= \textsc{if}\ \ulcorner x \in A \urcorner\ \textsc{then}\ \mathsf{Card}\,A\,(\mathsf{S}n)\ \textsc{else}\ \mathsf{Card}\,A\,n
\end{aligned}$$

Prove that the cardinality predicate agrees with the cardinality function:
$\forall An.\ \mathsf{Card}\,An \longleftrightarrow \mathsf{card}\,A = n$.

Exercise 14.8.4 (Disjointness predicate) We define **disjointness** of lists as follows:

$$\mathsf{disjoint}\,A\,B \ :=\ \neg\exists x.\ x \in A \wedge x \in B$$

Define a recursive predicate $\mathsf{Disjoint} : \mathcal{L}(X) \to \mathcal{L}(X) \to \mathbb{P}$ in the style of the cardinality predicate and verify that it agrees with the above predicate disjoint.

14.9 Position-Element Mappings

The positions of a list $[x_1, \ldots, x_n]$ are the numbers $0, \ldots, n-1$. More formally, a number n is a **position** of a list A if $n < \mathsf{len}\,A$. If a list is nonrepeating, we have a

bijective relation between the positions and the elements of the list. For instance, the list $[7,8,5]$ gives us the bijective relation

$$0 \leftrightarrow 7, \quad 1 \leftrightarrow 8, \quad 2 \leftrightarrow 5$$

It turns out that for a discrete type X we can define two functions

$$\begin{aligned} \mathsf{pos} &: \mathcal{L}(X) \to X \to \mathsf{N} \\ \mathsf{sub} &: X \to \mathcal{L}(X) \to \mathsf{N} \to X \end{aligned}$$

realizing the position-element bijection.

$$\begin{aligned} x \in A &\to \mathsf{sub}\, y\, A\, (\mathsf{pos}\, A\, x) = x \\ \mathsf{nrep}\, A \to n < \mathsf{len}\, A &\to \mathsf{pos}\, A\, (\mathsf{sub}\, y\, A\, n) = n \end{aligned}$$

The function pos use 0 as escape value for positions, and the function sub uses a given y^X as escape value for elements of X. The name sub stands for subscript. The functions pos and sub will be used in Chapter 17 for constructing injections and bijections on finite types.

Here are the definitions of pos and sub we will use:

$$\begin{aligned} \mathsf{pos} &: \mathcal{L}(X) \to X \to \mathsf{N} \\ \mathsf{pos}\, []\, x &:= 0 \\ \mathsf{pos}\, (a :: A)\, x &:= \text{IF } \ulcorner a = x \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{pos}\, A\, x) \end{aligned}$$

$$\begin{aligned} \mathsf{sub} &: X \to \mathcal{L}(X) \to \mathsf{N} \to X \\ \mathsf{sub}\, y\, []\, n &:= y \\ \mathsf{sub}\, y\, (a :: A)\, 0 &:= a \\ \mathsf{sub}\, y\, (a :: A)\, (\mathsf{S}n) &:= \mathsf{sub}\, y\, A\, n \end{aligned}$$

Fact 14.9.1 Let A be a list over a discrete type. Then:

1. $x \in A \to \mathsf{sub}\, a\, A\, (\mathsf{pos}\, A\, x) = x$
2. $x \in A \to \mathsf{pos}\, A\, x < \mathsf{len}\, A$
3. $n < \mathsf{len}\, A \to \mathsf{sub}\, a\, A\, n \in A$
4. $\mathsf{nrep}\, A \to n < \mathsf{len}\, A \to \mathsf{pos}\, A\, (\mathsf{sub}\, a\, A\, n) = n$

Proof All claims follow by induction on A. For (3), the inductive hypothesis must quantify n and the cons case needs case analysis on n. ∎

Exercise 14.9.2 Prove $(\forall X^{\mathbb{T}}.\, \mathcal{L}(X) \to \mathsf{N} \to X) \to \bot$.

Exercise 14.9.3 One can realize pos and sub with option types

$$\mathsf{pos} : \mathcal{L}(X) \to X \to \mathcal{O}(\mathsf{N})$$
$$\mathsf{sub} : \mathcal{L}(X) \to \mathsf{N} \to \mathcal{O}(X)$$

and this way avoid the use of escape values. Define pos and sub with option types for a discrete base type X and verify the following properties:

a) $x \in A \to \Sigma n.\ \mathsf{pos}\, A x = {}^\circ n$

b) $n < \mathsf{len}\, A \to \Sigma x.\ \mathsf{sub}\, A n = {}^\circ x$

c) $\mathsf{pos}\, A x = {}^\circ n \to \mathsf{sub}\, A n = {}^\circ x$

d) $\mathsf{nrep}\, A \to \mathsf{sub}\, A n = {}^\circ x \to \mathsf{pos}\, A x = {}^\circ n$

e) $\mathsf{sub}\, A n = {}^\circ x \to x \in A$

f) $\mathsf{pos}\, A x = {}^\circ n \to n < \mathsf{len}\, A$

15 Case Study: Expression Compiler

We verify a compiler translating arithmetic expressions into code for a stack machine. We use a reversible compilation scheme and verify a decompiler reconstructing expressions from their codes. The example hits a sweet spot of computational type theory: Inductive types provide a perfect representation for abstract syntax, and structural recursion on the abstract syntax provides for the definitions of the necessary functions (evaluation, compiler, decompiler). The correctness conditions for the functions can be expressed with equations, and generalized versions of the equations can be verified with structural induction.

This is the first time in our text we see an inductive type with binary recursion and two inductive hypotheses. Moreover, we see a notational convenience for function definitions known as catch-all equations.

15.1 Expressions and Evaluation

We will consider expressions for numbers that are obtained with constants, addition, and subtraction. Informally, we describe the abstract syntax of expressions with a scheme known as BNF:

$$e : \mathsf{exp} ::= x \mid e_1 + e_2 \mid e_1 - e_2 \qquad (x : \mathsf{N})$$

Following the BNF, we represent **expressions** with the inductive type

$$\mathsf{exp} : \mathbb{T} ::= \mathsf{con}(\mathsf{N}) \mid \mathsf{add}(\mathsf{exp}, \mathsf{exp}) \mid \mathsf{sub}(\mathsf{exp}, \mathsf{exp})$$

The ease our presentation, we will write the formal expressions provided by the inductive type exp using the notation suggested by the BNF. For instance:

$$e_1 + e_2 - e_3 \quad \rightsquigarrow \quad \mathsf{sub}(\mathsf{add}\, e_1 e_2) e_3$$

We can now define an **evaluation function** computing the values of expressions:

$$\begin{aligned} E &: \mathsf{exp} \to \mathsf{N} \\ E\, x &:= x \\ E\,(e_1 + e_2) &:= E\, e_1 + E\, e_2 \\ E\,(e_1 - e_2) &:= E\, e_1 - E\, e_2 \end{aligned}$$

Note that E is defined with binary structural recursion. Moreover, E is executable. For instance, $E(3+5-2)$ reduces to 6, and the equation $E(3+5-2) = E(2+3+1)$ follows by computational equality.

Exercise 15.1.1 Do the reduction $E(3+5-2) \succ^* 6$ step by step (at the equational level).

Exercise 15.1.2 Prove some of the constructor laws for expressions. For instance, show that con is injective and that add and sub are disjoint.

Exercise 15.1.3 Define an eliminator for expressions providing for structural induction on expressions. As usual the eliminator has a clause for each of the three constructors for expression. Since additions and subtractions have two subexpressions, the respective clauses of the eliminator have two inductive hypotheses.

15.2 Code and Execution

We will compile expressions into lists of numbers. We refer to the list obtained for an expression as the **code** of the expression. The compilation will be such that an expression can be reconstructed from its code, and that execution of the code yields the same value as evaluation of the expression.

Code is executed on a stack and yields a stack, where **stacks** are list of numbers. We define an **execution function** RCA executing a code C and a stack A as follows:

$$\begin{aligned}
R : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N}) & \\
R\ []\ A &:= A \\
R\ (0 :: x :: C)\ A &:= R\ C\ (x :: A) \\
R\ (1 :: C)\ (x_1 :: x_2 :: A) &:= R\ C\ (x_1 + x_2 :: A) \\
R\ (2 :: C)\ (x_1 :: x_2 :: A) &:= R\ C\ (x_1 - x_2 :: A) \\
R\ _\ _ &:= []
\end{aligned}$$

Note that the function R is defined by recursion on the first argument (the code) and by case analysis on the second argument (the stack). From the equations defining R you can see that the first number of the code determines what is done:

- 0: put the next number in the code on the stack.
- 1: take two numbers from the stack and put their sum on the stack.
- 2: take two numbers from the stack and put their difference on the stack.
- $n \geq 3$: stop execution and return the stack obtained so far.

The first equation defining R returns the stack obtained so far if the code is exhausted. The last equation defining R is a so-called **catch-all equation**: It applies whenever none of the preceding equations applies. Catch-all equations are a notational convenience that can be replaced by several equations providing the full case analysis.

Note that the execution function is defined with tail recursion, which can be realized with a loop at the machine level. This is in contrast to the evaluation function, which is defined with binary recursion. Binary recursion needs a procedure stack when implemented with loops at the machine level.

Exercise 15.2.1 Do the reduction $R[0,3,0,5,2][] \succ^* [2]$ step by step (at the equational level).

15.3 Compilation

We will define a compilation function $\gamma : \mathsf{exp} \to \mathcal{L}(\mathsf{N})$ such that $\forall e.\ R(\gamma e)[] = [Ee]$. That is, expressions are compiled to code that will yield the same value as evaluation when executed on the empty stack.

We define the **compilation function** by structural recursion on expressions:

$$\begin{aligned}
&\gamma : \mathsf{exp} \to \mathcal{L}(\mathsf{N}) \\
&\qquad \gamma x &:=\ & [0, x] \\
&\gamma(e_1 + e_2) &:=\ & \gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [1] \\
&\gamma(e_1 - e_2) &:=\ & \gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [2]
\end{aligned}$$

We now would like to show the correctness of the compiler:

$$R\ (\gamma e)\ []\ =\ [Ee]$$

The first idea is to show the equation by induction on e. This, however, fails since the recursive calls of R leave us with nonempty stacks and partial codes not obtainable by compilation. So we have to generalize both the possible stacks and the possible codes. The generalization of codes can be expressed with concatenation. Altogether we obtain a beautiful correctness theorem telling us much more about code execution than the correctness equation we started with.

Theorem 15.3.1 (Correctness) $R\ (\gamma e +\!\!+ C)\ A = R\ C\ (Ee :: A)$.

Proof By induction on e. The case for addition proceeds as follows:

$$\begin{aligned}
& R\ (\gamma(e_1 + e_2) +\!\!+ C)\ A \\
= {} & R\ (\gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [1] +\!\!+ C)\ A && \text{definition } \gamma \\
= {} & R\ (\gamma e_1 +\!\!+ [1] +\!\!+ C)\ (Ee_2 :: A) && \text{inductive hypothesis} \\
= {} & R\ ([1] +\!\!+ C)\ (Ee_1 :: Ee_2 :: A) && \text{inductive hypothesis} \\
= {} & R\ C\ ((Ee_1 + Ee_2) :: A) && \text{definition } R \\
= {} & R\ C\ (E(e_1 + e_2) :: A) && \text{definition } E
\end{aligned}$$

The equational reasoning implicitly employs conversion and associativity for concatenation $+\!\!+$. The full details can be explored with Coq. ∎

Corollary 15.3.2 $R\ (\gamma e)\ [] = [Ee]$.

Proof Theorem 15.3.1 with $C = A = []$. ∎

Exercise 15.3.3 Do the reduction $\gamma(5-2) \succ^* [0,3,0,5,2]$ step by step (at the equational level).

Exercise 15.3.4 Explore the proof of the correctness theorem starting from the proof script in the accompanying Coq development.

15.4 Decompilation

We now define a decompilation function that for all expressions recovers the expression from its code. This is possible since the compiler uses a reversible compilation scheme, or saying it abstractly, the compilation function is injective. The decompilation function closely follows the scheme used for code execution, where this time a stack of expressions is used.

$$\begin{aligned}
\delta : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{exp}) \to \mathcal{L}(\mathsf{exp}) \\
\delta\ []\ A &:= A \\
\delta\ (0 :: x :: C)\ A &:= \delta\ C\ (x :: A) \\
\delta\ (1 :: C)\ (e_1 :: e_2 :: A) &:= \delta\ C\ (e_1 + e_2 :: A) \\
\delta\ (2 :: C)\ (e_1 :: e_2 :: A) &:= \delta\ C\ (e_1 - e_2 :: A) \\
\delta\ _\ _ &:= []
\end{aligned}$$

The correctness theorem for decompilation follows closely the correctness theorem for compilation.

Theorem 15.4.1 (Correctness) $\delta\ (\gamma e +\!\!+ C)\ B = \delta\ C\ (e :: B)$.

Proof By induction on e. The case for addition proceeds as follows:

$$
\begin{aligned}
& \delta\ (\gamma(e_1 + e_2) +\!\!+ C)\ B \\
= \ & \delta\ (\gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [1] +\!\!+ C)\ B && \text{definition } \gamma \\
= \ & \delta\ (\gamma e_1 +\!\!+ [1] +\!\!+ C)\ (e_2 :: B) && \text{inductive hypothesis} \\
= \ & \delta\ ([1] +\!\!+ C)\ (e_1 :: e_2 :: B) && \text{inductive hypothesis} \\
= \ & \delta\ C\ ((e_1 + e_2) :: B) && \text{definition } \delta
\end{aligned}
$$

The equational reasoning implicitly employs conversion and associativity for concatenation $+\!\!+$. ∎

Corollary 15.4.2 $\delta\ (\gamma e)\ [] = [e]$.

15.5 Discussion

The semantics of the expressions and programs considered here is particularly simple since evaluation of expressions and execution of programs can be accounted for by structural recursion.

We represented expressions as abstract syntactic objects using an inductive type. Inductive types are the canonical representation of abstract syntactic objects. A concrete syntax for expressions would represent expressions as strings. While concrete syntax is important for the practical realisation of programming systems, it has no semantic relevance.

Early papers (late 1960's) on verifying compilation of expressions are McCarthy and Painter [9] and Burstall [3]. Burstall's paper is also remarkable because it seems to be the first exposition of structural recursion and structural induction. Compilation of expressions appears as first example in Chlipala's textbook [4], where it is used to get the reader acquainted with Coq.

The type of expressions is the first inductive type in this text featuring binary recursion. This has the consequence that the respective clauses in the induction principle have two inductive hypotheses. We find it remarkable that the generalization from linear recursion (induction) to binary recursion (induction) comes without intellectual cost.

16 Data Types

We study computational bijections and injections. Both are bidirectional and are obtained with two functions inverting each other. The inverse function of injections yields options so that it can exist if the primary function is not surjective. Injections transport equality deciders and existential witness operators from their codomain to their domain.

We define data types as types that come with an injection into the type of numbers. Data types inherit the order features of numbers and include all first-order inductive types. Data types can be characterized as types having an equality decider and an enumerator. Infinite data types can be characterized as types that are in bijection with the numbers.

You will see many option types and sigma types in this chapter. Option types are needed for the inverses of injections. Sigma types are used to represent the structures for bijections, injections, and data types.

16.1 Inverse Functions

We define predicates formulating basic properties of functions:

$$\begin{aligned}
\mathsf{injective}\, f &:= \forall x x'.\, fx = fx' \to x = x' \\
\mathsf{surjective}\, f &:= \forall y \exists x.\, fx = y \\
\mathsf{bijective}\, f &:= \mathsf{injective}\, f \wedge \mathsf{surjective}\, f \\
\mathsf{inv}\, g f &:= \forall x.\, g(fx) = x \qquad\qquad g \text{ inverts } f
\end{aligned}$$

The predicate $\mathsf{inv}\, g f$ is to be read as **g inverts f** or as **g is an inverse function for f**. There may be different inverse functions for a given function, even with functional extensionality.

Fact 16.1.1

1. $\mathsf{inv}\, g f \to \mathsf{surjective}\, g \wedge \mathsf{injective}\, f$
2. $\mathsf{inv}\, g f \to \mathsf{injective}\, g \vee \mathsf{surjective}\, f \to \mathsf{inv}\, f g$
3. $\mathsf{surjective}\, f \to \mathsf{inv}\, g f \to \mathsf{inv}\, g' f \to \forall y.\, gy = g'y$

Proof All claims follow by straightforward equational reasoning. Details are best understood with Coq. ∎

Note that Fact 16.1.1(3) says that all inverse functions of a surjective function agree.

The following lemma facilitates the construction of inverse functions.

Lemma 16.1.2 $\forall f^{X\to Y}.\ (\forall y \Sigma x.\ fx = y) \to \Sigma g.\ \mathsf{inv}\, fg.$

Proof Let $G : \forall y \Sigma x.\ fx = y$ and define $gy := \pi_1(Gy)$. ∎

Fact 16.1.3 (Transport) $\mathsf{injective}\, f^{X\to Y} \to \mathcal{E}Y \to \mathcal{E}X.$

Proof Exercise. ∎

Exercise 16.1.4 Give a function $\mathsf{N} \to \mathsf{N}$ that has disagreeing inverse functions.

16.2 Bijections

A **bijection** between two types X and Y consists of two functions

$$f : X \to Y$$
$$g : Y \to X$$

inverting each other

$$\forall x.\ g(fx) = x$$
$$\forall y.\ f(gy) = y$$

and thus establishing a bidirectional one-to-one connection between the elements of the two types. Formally, we define **bijection types** as nested sigma types:

$$\mathcal{B}XY \ :=\ \Sigma f^{X\to Y} \Sigma g^{Y\to X}.\ \mathsf{inv}\, gf \wedge \mathsf{inv}\, fg$$

We say that two types are **in bijection** if we have a bijection between them.

Fact 16.2.1 Bijectivity is a computational equivalence relation on types:

1. $\mathcal{B}XX$.
2. $\mathcal{B}XY \to \mathcal{B}YX$.
3. $\mathcal{B}XY \to \mathcal{B}YZ \to \mathcal{B}XZ$.

Proof Straightforward. ∎

Fact 16.2.2 Both functions of a bijection are bijective.

Proof Straightforward. ∎

Theorem 16.2.3 (Pairing) $\mathcal{B}\,\mathsf{N}\,(\mathsf{N}\times\mathsf{N})$.

Proof See Chapter 7. ∎

Exercise 16.2.4 Show that the following types are in bijection.

a) B and $\top+\top$.

b) B and $\mathcal{O}(\mathcal{O}(\bot))$.

c) $\top$ and $\mathcal{O}(\bot)$.

d) $\mathcal{O}(X)$ and $X+\top$.

e) $X+Y$ and $Y+X$.

f) $X\times Y$ and $Y\times X$.

g) N and $\mathcal{L}(\mathsf{N})$.

16.3 Injections

An **injection** of a type X into a type Y consists of two functions

$$f : X \to Y$$
$$g : Y \to \mathcal{O}(X)$$

such that

$$\forall x.\ \ g(fx) = {}^\circ x$$
$$\forall xy.\ \ gy = {}^\circ x \to fx = y$$

We say that f and g **quasi-invert** each other. We may think of an injection as an encoding or an embedding of a type X into a type Y. Following the encoding metaphor, we will refer to f as the **encoding function** and to g as the **decoding function**. We have the property that every x has a unique **code** and that every code can be uniquely decoded.

The decoding function determines whether a member of Y is a code ($gy \neq \emptyset$ iff y is a code). Obtaining this property is a main reason for using option types. Option types also ensure that the empty type embeds into every type.

Formally, we define **injection types** as nested sigma types:

$$\mathcal{I}XY \ :=\ \Sigma f^{X\to Y}\Sigma g^{Y\to\mathcal{O}(X)}.\ (\forall x.\, g(fx) = {}^\circ x) \wedge (\forall xy.\, gy = {}^\circ x \to fx = y)$$

We remark that our definition of injections is carefully chosen to fit practical and theoretical concerns. A previous version did not require the second equation for f and g. From the first equation one can obtain the second equation provided one is willing to modify the decoding function (Lemma 16.3.5). The second equation for injections for instance facilitates the construction of a least witness operator for data types (Fact 16.5.2).

Fact 16.3.1
Let f and g be the encoding and decoding function of an injection. Then:

1. The encoding function is injective: $fx = fx' \to x = x'$.
2. The decoding function is quasi-injective: $gy \neq \emptyset \to gy = gy' \to y = y'$.
3. The decoding function is quasi-surjective: $\forall x\, \exists y.\, gy = {}^\circ x$.
4. The decoding function determines the codes: $gy \neq \emptyset \longleftrightarrow \exists x.\, fx = y$.

Proof Straightforward.

Fact 16.3.2

1. $\mathcal{I}XX$ (reflexivity)
2. $\mathcal{I}XY \to \mathcal{I}YZ \to \mathcal{I}XZ$ (transitivity)
3. $\mathcal{B}XY \to \mathcal{I}XY$.
4. $\mathcal{I}\,\bot X$
5. $\mathcal{I}\,X(\mathcal{O}(X))$

Proof Straightforward. Claim 5 follows with $fx := {}^\circ x$ and $ga := a$. ∎

Fact 16.3.3 (Transport of equality deciders)
$\mathcal{I}XY \to \mathcal{E}Y \to \mathcal{E}X$.

Proof Let $f^{X\to Y}$ from $\mathcal{I}XY$. Then $x = x' \longleftrightarrow fx = fx'$ since f is injective. Thus an equality decider for Y yields an equality decider for X. ∎

We define a **type of witness operators**:

$$\mathcal{W}\,X^{\mathbb{T}} \;:=\; \forall p^{X\to\mathbb{P}}.\, (\forall X.\, \mathcal{D}(px)) \to (\exists x.px) \to (\Sigma x.px)$$

Fact 16.3.4 (Transport of witness operators)
$\mathcal{I}XY \to \mathcal{W}Y \to \mathcal{W}X$.

Proof Let $f^{X\to Y}$ and $g^{Y\to\mathcal{O}(X)}$ from $\mathcal{I}XY$. To show that there is a witness operator for X, we assume a decidable and satisfiable predicate $p^{X\to\mathbb{P}}$. We define a decidable and satisfiable predicate $q^{Y\to\mathbb{P}}$ as follows:

$$qy := \text{MATCH } gy\ [\ {}^\circ x \Rightarrow px \mid \emptyset \Rightarrow \bot\ ]$$

The witness operator for Y gives us a y such that qy. The definition of q gives us an x such that px. ∎

When we construct an injection, it is sometimes convenient to first construct a preliminary decoding function g that satisfies the first equation $\forall x.\, g(fx) = {}^\circ x$ and then use a general construction that from g obtains a proper decoding function satisfying both equations.

Lemma 16.3.5 (Upgrade) Given two functions $f^{X\to Y}$ and $g^{Y\to\mathcal{O}(X)}$ such that Y is discrete and $\forall x.\ g(fx) = {}^\circ x$, one can define a function g' such that f and g' form an injection $\mathcal{I}XY$.

Proof We assume $f^{X\to Y}$ and $g^{Y\to\mathcal{O}(X)}$ such that

$$\forall x.\ g(fx) = {}^\circ x$$

and define $g'^{\,Y\to\mathcal{O}(X)}$ as follows:

$$g'y := \begin{cases} {}^\circ x & \text{if } gy = {}^\circ x \wedge fx = y \\ \emptyset & \text{otherwise} \end{cases}$$

Verifying the two conditions

$$\forall x.\ g'(fx) = {}^\circ x$$
$$\forall xy.\ g'y = {}^\circ x \to fx = y$$

required so that f and g' form an injection is straightforward. ∎

Using a technique known as diagonalisation, Cantor showed that for no set the power set of a set embeds into the set. The result transfers to type theory where the function type $X \to \mathsf{B}$ takes the role of the power set.

Fact 16.3.6 (Cantor) $\mathcal{I}(X \to \mathsf{B})X \to \bot$.

Proof Let f and g be the functions from $\mathcal{I}(X \to \mathsf{B})X$. We define a function

$$h : X \to \mathsf{B}$$
$$hx := \text{MATCH } gx\ [\ {}^\circ\varphi \Rightarrow\ !\varphi x \mid \emptyset \to \mathbf{F}\ ]$$

It suffices to show $h(fh) = !h(fh)$. Follows using the definition of h and the equation $g(fh) = {}^\circ h$ on the right hand side. ∎

A related result is discussed in §8.3.

Exercise 16.3.7 Show $\mathcal{I}(X \to \mathsf{N})X \to \bot$.

Exercise 16.3.8 Show $\forall f^{X\to Y}\forall g.\ \mathsf{inv}\,gf \to \mathcal{E}Y \to \mathcal{I}XY$.

16.4 Data Types

We define **data types** as types that come with an injection into the type N of numbers:

$$\mathsf{dat}\, X := \mathcal{I} X \mathsf{N}$$

With this definition, data types are closed under forming product types, sum types, option types, and list types. Moreover, data types will come with equality deciders, existential witness operators, and least witness operators.

Fact 16.4.1

1. $\bot$, $\top$, and B are data types: $\mathsf{dat}\,\bot$, $\mathsf{dat}\,\top$, $\mathsf{dat}\,\mathsf{B}$.
2. N is a data type: $\mathsf{dat}\,\mathsf{N}$.
3. Types that embed into data types are data types: $\mathcal{I} X Y \to \mathsf{dat}\, Y \to \mathsf{dat}\, X$.
4. If X and Y are data types, then so are $X \times Y$, $X + Y$, $\mathcal{O}(X)$, and $\mathcal{L}(X)$:
   a) $\mathsf{dat}\, X \to \mathsf{dat}\, Y \to \mathsf{dat}\,(X \times Y)$
   b) $\mathsf{dat}\, X \to \mathsf{dat}\, Y \to \mathsf{dat}\,(X + Y)$
   c) $\mathsf{dat}\, X \to \mathsf{dat}\,(\mathcal{O}(X))$
   d) $\mathsf{dat}\, X \to \mathsf{dat}\,(\mathcal{L}(X))$

Proof The injections required for (4a) and (4b) can be constructed with the bijection of Theorem 16.2.3. ∎

Fact 16.4.2 (Equality decider)
Data types have equality deciders.

Proof Follows with Fact 16.3.3. ∎

Fact 16.4.3 (Existential witness operator)
Data types have existential witness operators.

Proof Follows with Fact 16.3.4. ∎

Fact 16.4.4 (Inverse functions)
Bijective functions from data types to discrete types have inverse functions:
$\mathsf{bijective}\, f^{X \to Y} \to \mathsf{dat}\, X \to \mathcal{E} Y \to \Sigma g^{Y \to X}.\ \mathsf{inv}\, g f \wedge \mathsf{inv}\, f g$.

Proof By Fact 16.1.1 (2) it suffices to construct a function g such that $\mathsf{inv}\, f g$. By Lemma 16.1.2 it suffices to show $\forall y \Sigma x.\ f x = y$. Follows from the surjectivity of f with the existential witness operator for X (Fact 16.4.3) and the discreteness of Y. ∎

An **enumerator** for a type X is a function $g^{\mathsf{N}\to\mathcal{O}(X)}$ such that $\forall x\,\exists n.\, gn = {}^\circ x$. It turns out that data types can be characterized as discrete enumerable types. To state the connection precisely, we define **enumerator types**:

$$\mathsf{enum}\, X^{\mathbb{T}} := \Sigma g^{\mathsf{N}\to\mathcal{O}(X)}.\, \forall x\,\exists n.\; gn = {}^\circ x$$

Fact 16.4.5 (Enumerator)
A type is a data type if and only if it has an equality decider and an enumerator:
$\mathsf{dat}\, X \;\Leftrightarrow\; \mathcal{E}X \times \mathsf{enum}\, X$.

Proof Direction $\Rightarrow$ is obvious. For the other direction, we assume an equality decider and an enumerator $g^{\mathsf{N}\to\mathcal{O}(X)}$ for X. The equality decider gives us a decider for the satisfiable predicate $\lambda n.gn = {}^\circ x$. Thus the existential witness operator for N gives us a function $f^{X\to\mathsf{N}}$ such that $\forall x.\, g(fx) = {}^\circ x$. Now the upgrade lemma 16.3.5 yields an injection $\mathcal{I}X\mathsf{N}$. ∎

Exercise 16.4.6 Show that $\mathsf{N} \to \mathsf{B}$ is not a data type.

Exercise 16.4.7 Show $\mathsf{dat}\, X \to \mathcal{W}(\mathcal{L}(X))$.

16.5 Data Types are Ordered

Data types inherit the order of numbers. If x is a member of a data type X, we will write $\#x$ for the unique code the encoding function of the injection $\mathcal{I}X\mathsf{N}$ assigns to x.

Fact 16.5.1 (Trichotomy) Let X be a data type. Then:
$\forall xy^X.(\#x < \#y) + (x = y) + (\#y < \#x)$.

Proof Trichotomy operator for numbers and injectivity of the encoding function. ∎

Fact 16.5.2 (Least witness operator)
$\mathsf{dat}\, X \to (\forall x.\, \mathcal{D}(px)) \to (\Sigma x.px) \to (\Sigma x.\, px \wedge \forall y.\, py \to \#x \le \#y)$.

Proof Let f and g be the functions from $\mathcal{I}X\mathsf{N}$. We define a predicate on numbers.

$$qn := \text{MATCH } gn\ [\ {}^\circ x \Rightarrow px \mid \emptyset \Rightarrow \bot\ ]$$

It is easy to see that q is decidable and Σ-satisfiable. Thus the least witness operator for numbers (Fact 11.7.3) gives us a least witness n of q. By the definition of q there is x such that px and $fx = n$. It remains to show $\forall y.\, py \to n \le fy$. Let py. Then $q(fy)$. Thus $n \le fy$ since n is the least witness of q. ∎

Exercise 16.5.3 Define an existential least witness operator for data types:
$\forall X^{\mathbb{T}}.\, \mathsf{dat}\, X \to (\forall x.\, \mathcal{D}(px)) \to (\exists x.\, px) \to (\Sigma x.\, px \wedge \forall y.\, py \to \#x \le \#y)$.

16.6 Infinite Types

There are several possibilities for defining infiniteness of types, not all of which are equivalent. We choose a propositional definition that is strong enough to put infinite data types into bijection with numbers. We define **infinite types** as types that for every list have an element that is not in the list:

$$\mathsf{infinite}\, X^{\mathbb{T}} \;:=\; \forall A^{\mathcal{L}(X)}\, \exists x^{X}.\, x \notin A$$

Fact 16.6.1 N is infinite.

Proof $\forall A^{\mathcal{L}(\mathsf{N})} \exists n \forall x.\, x \in A \to x < n$ follows by induction on A. ∎

Fact 16.6.2 (Transport) $\mathcal{I}XY \to \mathsf{infinite}\, X \to \mathsf{infinite}\, Y$.

Proof Let B be a list over Y, and let $f^{X\to Y}$ and $g^{Y\to\mathcal{O}(X)}$ be the functions coming with $\mathcal{I}XY$. We show $\exists y.\, y \notin B$. Let $A^{\mathcal{L}(X)}$ be the list obtained from $g@B$ by deleting the occurrences of $\emptyset$ and erasing the constructor $^\circ$ (Exercise 14.3.6). Since X is infinite, we have $x \notin A$. Then $fx \notin B$ (if $fx \in B$, then $x \in A$). ∎

Given a type X, we call a function $\forall A^{\mathcal{L}(X)}\, \Sigma x.\, x \notin A$ a **generator function** for X.

Fact 16.6.3 (Generator function)

1. Types with generator functions are infinite.
2. Infinite data types have generator functions.
3. Discrete types X with an injective function $\mathsf{N} \to X$ have generator functions.

Proof (1) is obvious.

(2) Follows from the fact that data types have equality deciders and witness operators (Facts 16.4.2 and 16.4.3).

For (3) we assume a discrete type X and an injective function $f^{\mathsf{N}\to X}$. Let $A^{\mathcal{L}(X)}$ and $n := \mathsf{len}\, A$. Since f is injective, the list $f@[0,\dots,n]$ is nonrepeating (Exercise 14.7.7). Since $\mathsf{len}\, A < \mathsf{len}\,(f@[0,\dots,n])$, the discrimination lemma 14.7.5 gives us an $x \notin A$. ∎

Exercise 16.6.4 Show that B is not infinite.

Exercise 16.6.5 Show $\mathsf{dat}\, X \to \mathsf{infinite}\, X \to \Sigma x^{X}.\, \top$.

16.7 Infinite Data Types

We will show that a data type is infinite if and only if it is in bijection with the type of numbers. We base this result on a lemma we call compression lemma.

Suppose we have a function $g : \mathsf{N} \to \mathcal{O}(X)$ where X is an infinite data type. Then we can see g as a sequence over X that has holes and repetitions.

$$g : \quad \emptyset, x_0, x_1, x_0, \emptyset, x_2, x_1, x_3, \ldots$$

If g covers all members of X, we can compress g into a sequence $h : \mathsf{N} \to X$ without holes and repetitions:

$$h : \quad x_0, x_1, x_3, \ldots$$

Seeing h as a function again, we have that h is a bijective function $\mathsf{N} \to X$.

Lemma 16.7.1 (Compression) Let X be an infinite data type. Then we can define a bijective function $\mathsf{N} \to X$.

Proof Let $g : \mathsf{N} \to \mathcal{O}(X)$ be the decoding function from $\mathcal{I}X\mathsf{N}$. We first define a chain $G_0 \subseteq G_1 \subseteq G_2 \subseteq \cdots$ collecting the values of g in X:

$$\begin{aligned} G_0 &:= [] \\ G_{\mathsf{S}n} &:= \text{MATCH } gn \text{ [} {}^\circ x \Rightarrow x :: G_n \mid \emptyset \Rightarrow Gn \text{]} \end{aligned}$$

We have $\forall x \exists n.\, x \in G_n$ since g is the decoding function from $\mathcal{I}X\mathsf{N}$.

For the next step we need a function

$$\Phi : \ \forall A^{\mathcal{L}(X)}\, \Sigma x.\ \ x \notin A \wedge \exists n.\, gn = {}^\circ x \wedge Gn \subseteq A$$

that for a list A yields the first x in g such that $x \notin A$. We postpone the construction of Φ and first show that Φ provides for the construction of a bijective function $\mathsf{N} \to X$.

Let $\varphi A := \pi_1(\Phi A)$. We define a chain $H_0 \subseteq H_1 \subseteq H_2 \subseteq \cdots$ over X and $h : \mathsf{N} \to X$ as follows:

$$\begin{aligned} H_0 &:= [] \\ H_{\mathsf{S}n} &:= \varphi H_n :: H_n \\ hn &:= \varphi(H_n) \end{aligned}$$

It's now straightforward to verify the following facts:

1. $x \in A \to x \neq \varphi A$.
2. $m < n \to hm \in H_n$.

3. $m < n \to hm \neq hn$.

Thus h is injective.

To show that h is surjective, it suffices to show $G_n \subseteq H_n$ and

$$x \in H_n \to \exists k.\ hk = x$$

Both claims follow by induction on n.

To conclude the proof, it remains to construct Φ, which in fact is the most beautiful part of the proof. We fix A and use the generator function provided by Fact 16.6.3 to obtain some $x_0 \notin A$. Using the encoding function from $\mathcal{I}X\mathsf{N}$, we obtain n_0 such that $gn_0 = {}^\circ x_0$. We now do a linear search $k = 0, 1, 2, \ldots$ until we find the first k such that $\exists x.\ gk = {}^\circ x \wedge Gn \subseteq A$. The search can be realized with structural recursion since we have the bound $k \leq n_0$. Formally, we construct a function

$$\forall k.\ k \leq n_0 \to G_k \subseteq A \to \Sigma x.\ x \notin A \wedge \exists n.\ gn = {}^\circ x \wedge Gn \subseteq A$$

by size recursion on $n_0 - k$. For $gk = \emptyset$, we recurse with Sk. For $gk = {}^\circ y$, we check $y \in A$. If $y \in A$, we recurse with Sk. If $y \notin A$, we terminate with $x = y$ and $n = k$. ∎

We can now show that infinite data types are exactly those types that are in bijection with N. In other words, up to bijection, N is the only infinite data type.

Theorem 16.7.2 (Characterizations of infinite data types)
For every type X the following types are equivalent:

1. $\mathcal{I}X\mathsf{N} \times \mathsf{infinite}\, X$
2. $\mathcal{E}X \times \Sigma f^{\mathsf{N}\to X}.\ \mathsf{bijective}\, f$
3. $\mathcal{B}X\mathsf{N}$
4. $\mathcal{I}X\mathsf{N} \times \mathcal{I}\mathsf{N}X$
5. $\mathcal{I}X\mathsf{N} \times \Sigma f^{\mathsf{N}\to X}.\ \mathsf{injective}\, f$

Proof 1 → 2. Compression lemma 16.7.1.
2 → 3. Inverse function lemma 16.4.4.
3 → 4. Fact 16.3.2 (3).
4 → 5. Fact 16.3.1.
5 → 1. Fact 16.6.3 (3). ∎

17 Finite Types

We define finite types as types that come with an equality decider and a list containing all elements of the type. We fix the cardinality of finite types with nonrepeating and covering lists. We show that finite types are data types, that that finite types embed into each other if and only if their cardinality permits. As one would expect, finite types of the same cardinality are in bijection. For every number n, a finite type of cardinality n can be obtained by n-times taking the option type of the empty type.

17.1 Coverings and Listings

A **covering of a type** is a list that contains every member of the type:

$$\mathsf{covering}\ A^{\mathcal{L}(X)} \ :=\ \forall x^X.\, x \in A$$

A **listing of a type** is a nonrepeating covering of the type:

$$\mathsf{listing}\ A^{\mathcal{L}(X)} \ :=\ \mathsf{covering}\ A \wedge \mathsf{nrep}\, A$$

We need a couple of results for coverings and listings of discrete types.

Fact 17.1.1 Given a covering of a discrete type, one can obtain a listing of the type:
$\mathcal{E}X \to \mathsf{covering}\ A^{\mathcal{L}(X)} \to \Sigma B^{\mathcal{L}(X)}.\ \mathsf{listing}\, B$.

Proof Fact 14.7.4. ∎

Fact 17.1.2 All listings of a discrete type have the same length.

Proof Follows with Corollary 14.7.6 (2). ∎

Fact 17.1.3 Let A and B be lists over a discrete type X.

1. $\mathsf{covering}\, A \to \mathsf{nrep}\, B \to \mathsf{len}\, A \le \mathsf{len}\, B \to \mathsf{listing}\, B$.
2. $\mathsf{listing}\, A \to \mathsf{covering}\, B \to \mathsf{len}\, B \le \mathsf{len}\, A \to \mathsf{listing}\, B$.
3. $\mathsf{listing}\, A \to \mathsf{len}\, B = \mathsf{len}\, A \to (\mathsf{nrep}\, B \longleftrightarrow \mathsf{covering}\, B)$.

Proof Follows with Corollary 14.7.6. ∎

17.2 Finite Types

We definite **finite types** as discrete types that come with a covering list:

$$\mathsf{fin}\ X^{\mathbb{T}} \ :=\ \mathcal{T}(X) \times \Sigma A^{\mathcal{L}(X)}.\ \mathsf{covering}\, A$$

This definition ensures that finite types are computational objects we can put our hands on. We already know that for a covering of a discrete type we can compute a listing of the type having a uniquely determined length. It will be convenient to have a second definition for finite types fixing a listing and announcing the **size of the type**:

$$\mathsf{fin}_n\ X^{\mathbb{T}} \ :=\ \mathcal{T}(X) \times \Sigma A^{\mathcal{L}(X)}.\ \mathsf{listing}\, A \wedge \mathsf{len}\, A = n$$

Fact 17.2.1 For every type X:

1. $\mathsf{fin}\, X \ \Leftrightarrow\ \Sigma n.\, \mathsf{fin}_n\, X$
2. $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, X \to m = n$ (uniqueness)

Proof Facts 17.1.1 and 17.1.2. ∎

Fact 17.2.2 If X and Y are finite types, then so are $X \times Y$, $X + Y$ and $\mathcal{O}(X)$.

Proof Discreteness follows with Facts 10.5.1 and 17.3.1. We leave the construction of the covering lists as an exercise. ∎

Fact 17.2.3 Finite types are data types: $\mathsf{fin}\, X \to C\, X$.

Proof By Fact 17.2.1 we assume a covering A for X. We use the upgrade lemma 16.3.5 so that only the first equation for $\mathcal{I}X\mathsf{N}$ needs to be verified. If A is empty, we construct $\mathcal{I}X\mathsf{N}$ with $fx := 0$ and $gn := \emptyset$. Otherwise, A contains an element a. We now use the position-element mappings pos and sub from § 14.9 and define

$$\begin{aligned} fx &:= \mathsf{pos}\, A\, x \\ gn &:= {}^{\circ}\mathsf{sub}\, a\, A\, n \end{aligned}$$

The equation $g(fx) = {}^{\circ}x$ now follows with Fact 14.9.1 ∎

Fact 17.2.4 Finite types are not infinite: $\mathsf{fin}\, X \to \mathsf{infinite}\, X \to \bot$.

Proof Exercise. ∎

Fact 17.2.5 $\mathsf{finite}\, X \ \to\ \mathcal{I}\mathsf{N}\, X \ \to\ \bot$.

Proof Follows with Facts 16.6.2, 16.6.1, and 17.2.4. ∎

Fact 17.2.6 (Injectivity-surjectivity agreement) Functions between finite types of the same cardinality are injective if and only if they are surjective:
$\mathsf{fin}_n X \to \mathsf{fin}_n Y \to \forall f^{X\to Y}.\ \mathsf{injective}\, f \longleftrightarrow \mathsf{surjective}\, f$.

Proof Let A and B be listings for X and Y, respectively, with $\mathsf{len}\, A = \mathsf{len}\, B$. We fix $f^{X\to Y}$ and have $\mathsf{covering}(f@A) \longleftrightarrow \mathsf{nrep}(f@A)$ by Fact 17.1.3 (3).

Let f be injective. Then $f@A$ is nonrepeating by Exercise 14.7.7 (a). Thus $f@A$ is covering. Hence f is surjective.

Let f be surjective. Then $f@A$ is covering and thus nonrepeating. Thus f is injective by Exercise 14.7.7 (b). ∎

Exercise 17.2.7 Prove $\mathsf{fin}_0\, \bot$, $\mathsf{fin}_1\, \top$, and $\mathsf{fin}_2\, \mathbb{B}$.

Exercise 17.2.8 Prove the following:

a) $\mathsf{dat}\, X \to (\exists A^{\mathcal{L}(X)} \forall x.\, x \in A) \to \mathsf{fin}\, X$.

b) $\mathsf{XM} \to \mathsf{dat}\, X \to \mathsf{infinte}\, X \vee (\exists A^{\mathcal{L}(X)} \forall x.\, x \in A)$.

Proving (b) with a sum type rather than a disjunction seems impossible.

Exercise 17.2.9 (Bounded quantification)
Let p be a decidable predicate on a finite type X. Prove the following types:

a) $\mathcal{D}(\forall x.px)$

b) $\mathcal{D}(\exists x.px)$

c) $(\Sigma x.px) + (\forall x.\neg px)$

Exercise 17.2.10
Prove $\mathsf{fin}_m X \to \mathsf{fin}_n Y \to m > 0 \to (\forall f^{X\to Y}.\ \mathsf{injective}\, f \longleftrightarrow \mathsf{surjective}\, f) \to m = n$.

17.3 Finite Ordinals

We define for every number n a finite type F_n with exactly n elements by applying n-times the option type constructor to the empty type $\bot$:

$$\mathsf{F}_n := \mathcal{O}^n(\bot)$$

We refer to the types F_n as **finite ordinals**.

Fact 17.3.1 (Discreteness) The finite ordinals are discrete: $\mathcal{E}(\mathsf{F}_n)$.

Proof Follows by induction on n since $\bot$ is discrete (Fact 10.5.1) and $\mathcal{O}$ preserves discreteness (Fact 10.6.2). ∎

We define a function $\mathsf{L} : \forall n.\ \mathcal{L}(\mathsf{F}_n)$ that yields a listing for every finite ordinal:

$$\begin{aligned} \mathsf{L}_0 &:= [] \\ \mathsf{L}_{\mathsf{S}n} &:= \emptyset :: (^\circ @ \mathsf{L}_n) \end{aligned}$$

For instance, $\mathsf{L}_4 = [\emptyset, {}^\circ\emptyset, {}^{\circ\circ}\emptyset, {}^{\circ\circ\circ}\emptyset]$.

Fact 17.3.2 L_n is a listing of F_n having length n.

Proof By induction on n. ∎

Fact 17.3.3 F_n is a finite type of size n: $\mathsf{fin}_n\, \mathsf{F}_n$.

Proof Facts 17.3.1 and 17.3.2. ∎

17.4 Bijections and Finite Types

Fact 17.4.1 (Transport) $\mathcal{B}XY \to \mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y$.

Proof Bijections map listings to listings and preserve their length. ∎

Theorem 17.4.2 (Finite bijection)
Finite types of the same size are in bijection: $\mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y \to \mathcal{B}\, X\, Y$.

Proof Let A and B be listings of X and Y, respectively, both of length n. If $A = B = []$, we can define functions $X \to \bot$ and $Y \to \bot$ and thus the claim follows with computational elimination for $\bot$. Otherwise, we have $a \in A$ and $b \in B$. The listings A and B give us bijective connections between the elements of X and the positions $0, \ldots, n-1$, and the elements of Y and the positions $0, \ldots, n-1$. We realize the resulting bijection between X and Y using the list operations sub and pos with escape values (§ 14.9):

$$\begin{aligned} f\,x &:= \mathsf{sub}\, b\, B\ (\mathsf{pos}\, A\, x) \\ g\,y &:= \mathsf{sub}\, a\, A\ (\mathsf{pos}\, B\, y) \end{aligned}$$

Recall that pos yields the position of a value in a list, and that sub yields the value at a position of a list. Since A and B are covering, the escape values a and b will not be used by sub. ∎

Corollary 17.4.3 $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to (\mathcal{B}XY \Leftrightarrow m = n)$.

Proof Direction ⇒ follows with Facts 17.4.1 and 17.2.1 (2). Direction ⇐ follows with Theorem 17.4.2. ∎

Exercise 17.4.4 Prove the following:

a) $\mathsf{fin}_n\, X \to \mathcal{B} X \mathsf{F}_n$.

b) $\mathcal{B}\,\mathsf{F}_m\,\mathsf{F}_n \to m = n$.

c) $\mathcal{B}\,\mathsf{F}_n\,\mathsf{F}_{\mathsf{S}n} \to \bot$.

17.5 Injections and Finite Types

We may consider a type X is smaller than a type Y if X can be embedded into Y with an injection. For finite types, this abstract notion of size agrees with the numeric size we have assigned to finite types through nonrepeating lists.

Lemma 17.5.1 (Transport of covering lists)
$\mathcal{I} X Y \to \mathsf{covering}_Y\, B \to \Sigma A.\ \mathsf{covering}_X\, A$.

Proof Let B be a covering of Y, and let $f : X \to Y$ and $g : Y \to \mathcal{O}(X)$ be the functions coming with $\mathcal{I} X Y$. Let $A : \mathcal{L}(X)$ be the list obtained from $g@B$ by deleting the occurrences of $\emptyset$ and erasing the constructor $^\circ$ (Exercise 14.3.6). We show that A is covering. Let $x : X$. Then $fx \in B$. Hence ${}^\circ x = g(fx) \in g@B$. Thus $x \in A$. ∎

Fact 17.5.2 (Transport of finiteness)
$\mathcal{I} X Y \to \mathsf{fin}\, Y \to \mathsf{fin}\, X$.

Proof Fact 16.3.3 and Lemma 17.5.1. ∎

Fact 17.5.3 (Characterizations of finite types)
For every type X the following types are equivalent:

1. $\mathsf{fin}\, X$
2. $\Sigma n.\, \mathsf{fin}_n\, X$
3. $\Sigma n.\, \mathcal{B} X \mathsf{F}_n$
4. $\Sigma Y.\, \mathcal{I} X Y \times \mathsf{fin}\, Y$

Note that each of the types gives us a characterization of finite types.

Proof The equivalences follow with Facts 17.2.1 and 17.3.3, Theorem 17.4.2, and Facts 16.3.2 (3) and 17.5.2. ∎

Fact 17.5.4
$\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to m \le n \to \mathcal{I}\, X\, Y$.

Proof The proof is similar to the proof of Theorem 17.4.2. Again we use the upgrade lemma 16.3.5 so that only the first equation for $\mathcal{I} X Y$ needs to be verified.

Let A be listing of X of length m and B be a listings of Y of length $n \geq m$. If $A = []$, we can define a function $X \to \bot$ and the claim follows with computational elimination for $\bot$. Otherwise, we have an escape values $a \in A$ and $b \in B$. We realize the injection of X into Y as follows:

$$\begin{aligned} f\,x &:= \mathsf{sub}\,b\,B\ (\mathsf{pos}\,A\,x) \\ g\,y &:= {}^\circ\mathsf{sub}\,a\,A\ (\mathsf{pos}\,B\,y) \end{aligned}$$

■

Fact 17.5.5
$\mathsf{fin}_m\,X \to \mathsf{fin}_n\,Y \to \mathcal{I}\,X\,Y \to m \leq n$.

Proof Let A be a nonrepeating list of length m over X, B be a covering list over Y of length n, and $f : X \to Y$ be injective. Then $f@A \subseteq B$ is nonrepeating and thus $m \leq n$ by Corollary 14.7.6. ■

Theorem 17.5.6 (Finite injection) $\mathsf{fin}_m\,X \to \mathsf{fin}_n\,Y \to (m \leq n \Leftrightarrow \mathcal{I}XY)$.

Proof Follows with Facts 17.5.4 and 17.5.5. ■

Corollary 17.5.7 $\mathsf{fin}_m\,X \to \mathsf{fin}_n\,Y \to m > n \to \mathcal{I}XY \to \bot$.

Fact 17.5.8 (Finite sandwich)

1. $\mathcal{I}XY \to \mathcal{I}YX \to (\mathsf{fin}_n\,X \Leftrightarrow \mathsf{fin}_n\,Y)$.
2. $\mathcal{I}XY \to \mathcal{I}YX \to \mathsf{fin}\,X \to \mathcal{B}XY$.

Proof Claim 1 follows with Facts 17.2.1, 17.5.2, and 17.5.5. Claim 2 follows with Fact 17.2.1, Claim 1, and Theorem 17.4.2. ■

Exercise 17.5.9 Prove the following:

a) $\mathcal{I}\,\mathsf{F}_m\,\mathsf{F}_n \Leftrightarrow m \leq n$.

b) $\mathcal{I}\,\mathsf{F}_{\mathsf{S}n}\,\mathsf{F}_n \to \bot$.

Exercise 17.5.10 Prove the following:

1. $\mathcal{I}\,\mathsf{N}\,\mathsf{F}_n \to \bot$.
2. $\mathsf{F}_n \neq \mathsf{N}$.

18 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, among them 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.

18.1 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.

18.2 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.[1]

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

- 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.

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

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 18.2.1 (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). ∎

Corollary 18.2.2 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.

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

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 18.2.3 (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 18.2.1. ∎

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 18.2.4 (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 → 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 → 3. Trivial.

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

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

4 → 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 18.2.4 we learn that a provability predicate is consistent if there are consistent propositions.

Exercise 18.2.5 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 18.2.6 Let $X \to Y$ be provable. Show that X and Y are both independent if X is consistent and Y is unprovable.

Exercise 18.2.7 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 $\neg\mathsf{provable}\ \bot \longleftrightarrow \neg\forall X^{\mathbb{P}}.\ \mathsf{provable}\ X$.

Exercise 18.2.8 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.

18.3 Prominent Independent Propositions

Figure 18.1 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.

Note that LPO and Markov talk about tests on numbers and quantify only over types in $\mathbb{T}_1$. The other propositions in Figure 18.1 do not involve data types but quantify over universes.

Using the sandwich theorem, we will be able to show that all propositions in Figure 18.1 are independent provided we are given two nontrivial metatheorems.

Theorem 18.3.1 (Meta) TVS $\wedge$ FE is consistent.

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

Figure 18.1: Independent propositions

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

Figure 18.2: Provable implications

Theorem 18.3.2 (Meta) Neither Markov nor PI nor FE is provable.

From Theorem 18.3.1 we obtain consistency as a corollary.

Corollary 18.3.3 (Meta) $\bot$ is unprovable.

Proof Follows with Fact 18.2.4 from Theorem 18.3.1. ∎

We will now prove the implications shown in Figure 18.2. The implications suffice so that with the sandwich theorem 18.2.3 and the metatheorems 18.3.1 and 18.3.2 we know that all propositions of Figure 18.1 are independent.

Fact 18.3.4 $\mathsf{XM} \longleftrightarrow \forall X^{\mathbb{P}}.\ (X \longleftrightarrow \top) \vee (X \longleftrightarrow \bot)$.

Proof Straightforward. ∎

Fact 18.3.5 $\mathsf{TVS} \longleftrightarrow \mathsf{XM} \wedge \mathsf{PE}$.

Proof Straightforward using Fact 18.3.4. ∎

Fact 18.3.6 XM → LPO and LPO → Markov.

Proof The first claim is obvious. For the second claim assume LPO and f such that $H : \neg\forall n.\, fn = \mathbf{F}$. By LPO we assume $H_1 : \neg\exists n.\, fn = \mathbf{T}$ and prove falsity. By H we prove $fn = \mathbf{F}$ for some n. By boolean case analysis we assume $fn = \mathbf{T}$ 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 18.3.7 $\bot$ and $\top$ are pure.

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

Fact 18.3.8 PE → PI.

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

Theorem 18.3.9 (Meta) All propositions in Figure 18.1 are independent.

Proof By the sandwich theorem 18.2.3 and Facts 18.3.5, 18.3.8, and 18.3.6 it suffices to show that TVS and FE are consistent and that PI, Markov, and FE are unprovable. This is exactly what Theorems 18.3.1 and 18.3.2 say. ∎

Exercise 18.3.10 Prove $\mathsf{TVS} \longleftrightarrow \forall XYZ{:}\mathbb{P}.\ X = Y \vee X = Z \vee Y = Z$. Note that the equivalence characterizes TVS without using $\top$ and $\bot$.

Exercise 18.3.11 Prove $\mathsf{TVS} \longleftrightarrow \forall p^{\mathbb{P}\to\mathbb{P}}.\, p\top \to p\bot \to \forall X.pX$. Note that the equivalence characterizes TVS without using propositional equality.

Exercise 18.3.12 Prove that $\forall X^{\mathbb{T}}.\, X = \top \vee X = \bot$ is contradictory.

Exercise 18.3.13 We define **implicational excluded middle** as

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

a) Prove XM → IXM.

b) Prove that IXM is consistent.

We remark that neither IXM nor IXM → XM is provable in Coq's type theory.

Exercise 18.3.14 We define **weak excluded middle** as

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

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

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

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

Exercise 18.3.16 Prove $\mathsf{FE} \to \mathsf{B} \neq (\top \to \top)$.

Exercise 18.3.17 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 18.3.18 Suppose there is a function $f : (\top \vee \top) \to \mathsf{B}$ such that $f(\mathsf{L}\,\mathsf{I}) = \mathbf{T}$ and $f(\mathsf{R}\,\mathsf{I}) = \mathbf{F}$. Prove $\neg\,\mathsf{PI}$. Convince yourself that without the elim restriction you could define a function f as assumed.

Exercise 18.3.19 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.

18.4 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:

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

Exercise 18.4.1 Prove $x \in (p - q) \longleftrightarrow x \in p \wedge x \notin q$. Check that the equation $(x \in (p - q)) = (x \in p \wedge x \notin q)$ holds by computational equality.

Exercise 18.4.2 Assume FE and PE and prove the following:

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

18.5 No Computational Omniscience

Coq's type theory is carefully designed such that every definable function is computable. 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**:

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

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 18.5.1 CO → LPO.

Proof Straightforward. ∎

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

Conjecture 18.5.2 (Meta) TVS ∧ FE ∧ ¬CO is consistent.

It turns out that assuming CO is also consistent.

Theorem 18.5.3 (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.

18.6 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

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

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).

19 Natural Deduction

We formalize an intuitionistic and a classical ND system using indexed inductive types. We prove properties of the systems using induction on derivations obtained with recursive eliminators. We show that intuitionistic provability implies classical provability, that the double negation law is independent in the intuitionistic system, and that classical provability reduces to intuitionistic provability using double negation. We define boolean evaluation of formulas and show that a certifying boolean solver yields agreement of classical ND with boolen entailment as well as decidability. The chapter is designed such that it can serve as in introduction to propositional ND systems, and also as first example for the use of indexed inductive definitions.

19.1 ND Systems

We start with an informal explanation of natural deduction systems. *Natural deduction systems* (ND systems) come with a class of *formulas* and a system of *deduction rules* for building *derivations* of pairs (A, s) consisting of a list of formulas A (the *context*) and a single formula s (the *conclusion*). The formulas in A play the role of *assumptions*. That a pair (A, s) is derivable with the rules of the system is understood as saying that s is provable with the assumptions in A and the rules of the system. Given a concrete class of formulas, we can have different sets of rules and compare their deductive power. Given a concrete deduction system, we may ask the following questions:

- *Consistency:* Are there formulas we cannot derive?
- *Weakening property:* Given a derivation of (A, s) and a list B such that $A \subseteq B$, can we always obtain a derivation of (B, s)?
- *Cut property:* Given derivations of (A, s) and $(s :: A, t)$, can we always obtain a derivation of (A, t)?
- *Decidability:* Is it decidable whether a pair (A, s) is derivable?

All but the last property formulate basic integrity conditions for natural deduction systems.

We will consider the following type of **formulas**:

$$s, t, u, v : \mathsf{For} := x \mid \bot \mid s \to t \mid s \wedge t \mid s \vee t \qquad (x : \mathsf{N})$$

$$\mathsf{A}_{\vdash}\ \frac{s \in A}{A \vdash s} \qquad \mathsf{E}_{\bot}\ \frac{A \vdash \bot}{A \vdash s} \qquad \mathsf{I}_{\to}\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \mathsf{E}_{\to}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

$$\mathsf{I}_{\wedge}\ \frac{A \vdash s \qquad A \vdash t}{A \vdash s \wedge t} \qquad \mathsf{E}_{\wedge}\ \frac{A \vdash s \wedge t \qquad A, s, t \vdash u}{A \vdash u}$$

$$\mathsf{I}^1_{\vee}\ \frac{A \vdash s}{A \vdash s \vee t} \qquad \mathsf{I}^2_{\vee}\ \frac{A \vdash t}{A \vdash s \vee t} \qquad \mathsf{E}_{\vee}\ \frac{A \vdash s \vee t \qquad A, s \vdash u \qquad A, t \vdash u}{A \vdash u}$$

Figure 19.1: Deduction rules of the intuitinistic ND system

Formulas of the kind x are called **atomic formulas**. Atomic formulas represent atomic propositions whose meaning is left open. For the other kinds of formulas the symbols used give away the intended meaning. Formally, the type For of formulas is accommodated as an inductive type that has a value constructor for each kind of formula (5 altogether).[1] We will use the notation $\neg s := s \to \bot$.

Exercise 19.1.1 (Formulas)

a) Show some of the constructor laws for the type of formulas.

b) Define an eliminator providing for structural induction on formulas.

c) Define an equality decider for the type of formulas.

19.2 Intuitionistic ND System

The deduction rules of the intuitionistic ND system we will consider are given in Figure 19.1 using several notational gadgets:

- *Turnstile notation* $A \vdash s$ for pairs (A, s).
- *Comma notation* A, s for lists $s :: A$.
- *Ruler notation* for deduction rules. For instance,

$$\frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

  describes a rule (known as modus ponens) that obtains a derivation of (A, t) from derivations of $(A, s \to t)$ and (A, s). We say that the rule has two *premises* and one *conclusion*.

[1] The use of abstract syntax is discussed more carefully in Chapter 7.

All rules in Figure 19.1 express proof rules you are familiar with from mathematical reasoning and the logical reasoning you have seen in this text. In fact, the system of rules in Figure 19.1 can derive exactly those pairs (A, s) that are known to be **intuitionistically deducible** (given the formulas we consider). Since reasoning in type theory is intuitionistic, Coq can prove a goal (A, s) if and only if the rules in Figure 19.1 can derive the pair (A, s) (where atomic formulas are accommodated as propositional variables in type theory). We will exploit this coincidence when we construct derivations using the rules in Figure 19.1.

The rules in Figure 19.1 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 Figure 19.1 is known as **assumption rule**. Note that every rule but the assumption rule is an introduction or an elimination rule for some logical constant. Also 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 known as **explosion rule**.

Note that no deduction rule contains more than one logical constant. This results in an important modularity property. If we want to omit a logical constant, for instance $\wedge$, we just omit all rules containing this constant. Note that every system with $\bot$ and $\to$ can express negation. When trying to understand the structural properties of the system, it is usually a good idea to just consider $\bot$ and $\to$. Note that the assumption rule cannot be omitted since it is the only rule not taking a derivation as premise.

Exercise 19.2.1 A derivation for $s \vdash \neg\neg s$ may be depicted as a *derivation tree* as follows:

$$\dfrac{\dfrac{}{s, \neg s \vdash \neg s}\,\mathsf{A} \qquad \dfrac{}{s, \neg s \vdash s}\,\mathsf{A}}{\dfrac{s, \neg s \vdash \bot}{s \vdash \neg\neg s}\,\mathsf{I}_{\to}}\,\mathsf{E}_{\to}$$

The labels A, $\mathsf{E}_{\to}$, and $\mathsf{I}_{\to}$ act as names for the rules used (assumption, elimination, and introduction). We ease our notation by omitting the list brackets at the left of $\vdash$.

Give derivation trees for $A \vdash (s \to s)$ and $\neg\neg\bot \vdash \bot$.

19.3 Formalisation with Indexed Inductive Type Definition

It turns out that propositional deduction systems like the one in Figure 19.2 can be formalized elegantly and directly with inductive type definitions accommodating deduction rules as value constructors of derivation types $A \vdash s$.

$$
\begin{array}{rll}
s \in A \to A \vdash s & & \mathsf{A}_{\vdash} \\
A \vdash \bot \to A \vdash s & & \mathsf{E}_{\bot} \\
A, s \vdash t \to A \vdash (s \to t) & & \mathsf{I}_{\to} \\
A \vdash (s \to t) \to A \vdash s \to A \vdash t & & \mathsf{E}_{\to} \\
A \vdash s \to A \vdash t \to A \vdash (s \wedge t) & & \mathsf{I}_{\wedge} \\
A \vdash (s \wedge t) \to A, s, t \vdash u \to A \vdash u & & \mathsf{E}_{\wedge} \\
A \vdash s \to A \vdash (s \vee t) & & \mathsf{I}^1_{\vee} \\
A \vdash t \to A \vdash (s \vee t) & & \mathsf{I}^2_{\vee} \\
A \vdash (s \vee t) \to A, s \vdash u \to A, t \vdash u \to A \vdash u & & \mathsf{E}_{\vee}
\end{array}
$$

Prefixes for A, s, t, u omitted, constructor names given at the right

Figure 19.2: Value constructors for derivation types $A \vdash s$

Let us explain this fundamental idea. We may see the deduction rules in Figure 19.1 as functions that given derivations for the pairs in the premises yield a derivation for the pair in the conclusion. The introduction rule for conjunctions, for instance, may be seen as a function that given derivations for (A, s) and (A, t) yields a derivation for $(A, s \wedge t)$. We now go one step further and formalize the deduction rules as the value constructors of an inductive type constructor

$$\vdash \; : \; \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{T}$$

This way the values of an inductive type $A \vdash s$ represent the derivations of the pair (A, s) we can obtain with the deduction rules. To emphasize this point, we call the types $A \vdash s$ **derivation types**.

The value constructors for the derivation types $A \vdash s$ of the intuitionistic ND system appear in Figure 19.2. Note that the types of the constructors follow exactly the patterns of the deduction rules in Figure 19.2.

When we look at the target types of the constructors in Figure 19.2, it becomes clear that the argument s of the type constructor $A \vdash s$ is *not* a parameter since it is instantiated by the constructors for the introduction rules ($\mathsf{I}_{\to}$, $\mathsf{I}_{\wedge}$, $\mathsf{I}^1_{\vee}$, $\mathsf{I}^2_{\vee}$). Such nonparametric arguments of type constructors are called **indices**. In contrast, the argument A of the type constructor $A \vdash s$ is a parameter since it is not instantiated in the target types of the constructors. More precisely, the argument A is a *nonuniform* parameter of the type constructor $A \vdash s$ since it is instantiated in some argument types of some of the constructors ($\mathsf{I}_{\to}$, $\mathsf{E}_{\wedge}$, and $\mathsf{E}_{\vee}$).

We call inductive type definitions where the type constructor has indices **indexed inductive definitions**. Indexed inductive definitions can also introduce **indexed**

inductive predicates. In fact, we alternatively could introduce $\vdash$ as an indexed inductive predicate and this way demote derivations from computational objects to proofs.

The suggestive BNF-style notation we have used so far to write inductive type definitions does not generalize to indexed inductive type definitions. So we will use an explicit format giving the type constructor together with the list of value constructors. Often, the format used in Figure 19.2 will be convenient.

We can now do simple proofs using the value constructors. We offer some examples. We ease our notation by writing $\neg\neg\bot \vdash \bot$ for $[\neg\neg\bot] \vdash \bot$, for instance.

Fact 19.3.1 (1) $s, A \vdash s$ (2) $\neg\neg\bot \vdash \bot$ (3) $s \vdash \neg\neg s$

Proof (1) follows with $\mathsf{A}_{\vdash}$.

(2) follows with $\mathsf{E}_{\to}$ from $\neg\neg\bot \vdash \neg\neg\bot$ and $\neg\neg\bot \vdash \neg\bot$. The first subgoal follows with $\mathsf{A}_{\vdash}$. The second subgoal follows with $\mathsf{I}_{\to}$ and $\mathsf{A}_{\vdash}$.

(3) follows with $\mathsf{I}_{\to}$ from $s, \neg s \vdash \bot$, which follows with $\mathsf{E}_{\to}$ from $s, \neg s \vdash \neg s$ and $s, \neg s \vdash s$, which both follow with $\mathsf{A}_{\vdash}$. ∎

Fact 19.3.2 (Cut) $A \vdash s \to A, s \vdash t \to A \vdash t$.

Proof We assume $A \vdash s$ and $A, s \vdash t$ and derive $A \vdash t$. By $\mathsf{I}_{\to}$ we have $A \vdash (s \to t)$. Thus $A \vdash t$ by $\mathsf{E}_{\to}$. ∎

The cut lemma gives us a function that given a derivation $A \vdash s$ and a derivation $A, s \vdash t$ yields a derivation $A \vdash t$. Informally, the cut lemma says that once we have derived s from A, we can use s like an assumption.

Fact 19.3.3 (Bottom) $A \vdash \neg\neg\bot \to A \vdash \bot$.

Proof Exercise. ∎

19.4 The Eliminator

For more interesting proofs it will be necessary to do inductions on derivations. As it was the case for non-indexed inductive types, we can define an eliminator providing for the necessary inductions. The definition of the eliminator is shown in Figure 19.3. While the definition of the eliminator is frighteningly long, it is regular and modular: Every deduction rule (i.e., value constructor) is accounted for with a separate type clause and a separate defining equation. To understand the definition of the eliminator, it suffices that you pick one of the deduction rules and look at the type clause and the defining equation for the respective value constructor.

$$
\begin{aligned}
&\mathsf{E}_{\vdash} : \forall p^{\mathcal{L}(\mathsf{For})\to\mathsf{For}\to\mathbb{T}}.\\
&\quad (\forall As.\ s \in A \to pAs) \to\\
&\quad (\forall As.\ pA\bot \to pAs) \to\\
&\quad (\forall Ast.\ p(s :: A)t \to pA(s \to t)) \to\\
&\quad (\forall Ast.\ pA(s \to t) \to pAs \to pAt) \to\\
&\quad (\forall Ast.\ pAs \to pAt \to pA(s \wedge t)) \to\\
&\quad (\forall Astu.\ pA(s \wedge t) \to p(s :: t :: A)u \to pAu) \to\\
&\quad (\forall Ast.\ pAs \to pA(s \vee t)) \to\\
&\quad (\forall Ast.\ pAt \to pA(s \vee t)) \to\\
&\quad (\forall Astu.\ pA(s \vee t) \to p(s :: A)u \to p(t :: A)u \to pAu) \to\\
&\quad \forall As.\ A \vdash s \to pAs
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{E}_{\vdash}\, pf_1 \ldots f_9\, A\, \_\, (\mathsf{A}_{\vdash}\, sh) &:= f_1 Ash\\
(\mathsf{E}_{\bot}\, sd) &:= f_2 As(\mathsf{E}_{\vdash} \ldots A\bot d)\\
(\mathsf{I}_{\to}\, std) &:= f_3 Ast(\mathsf{E}_{\vdash} \ldots (s :: A)td)\\
(\mathsf{E}_{\to}\, std_1d_2) &:= f_4 Ast(\mathsf{E}_{\vdash} \ldots A(s \to t)d_1)(\mathsf{E}_{\vdash} \ldots Asd_2)\\
(\mathsf{I}_{\wedge}\, std_1d_2) &:= f_5 Ast(\mathsf{E}_{\vdash} \ldots Asd_1)(\mathsf{E}_{\vdash} \ldots Atd_2)\\
(\mathsf{E}_{\wedge}\, stud_1d_2) &:= f_6 Astu(\mathsf{E}_{\vdash} \ldots A(s \wedge t)d_1)(\mathsf{E}_{\vdash} \ldots (s :: t :: A)ud_2)\\
(\mathsf{I}^1_{\vee}\, std) &:= f_7 Ast(\mathsf{E}_{\vdash} \ldots Asd)\\
(\mathsf{I}^2_{\vee}\, std) &:= f_8 Ast(\mathsf{E}_{\vdash} \ldots Atd)\\
(\mathsf{E}_{\vee}\, stud_1d_2d_3) &:= f_9 Astu(\mathsf{E}_{\vdash} \ldots A(s \vee t)d_1)\\
&\qquad\qquad (\mathsf{E}_{\vdash} \ldots (s :: A)ud_2)\\
&\qquad\qquad (\mathsf{E}_{\vdash} \ldots (t :: A)ud_3)
\end{aligned}
$$

Figure 19.3: Eliminator for $A \vdash s$

The eliminator formalizes the idea of induction on derivations, which informally is easy to master. With a proof assistant, the eliminator can be derived automatically from the inductive type definition, and its application can be supported such that the user is presented the proof obligations for the constructors once the induction is initiated.

As it comes to the patterns (i.e., the left-hand sides) of the defining equations, there is a new feature coming with indexed inductive types. Recall that patterns must be linear, that is, no variable must occur twice, and no constituent must be referred to by more than one variable. With parameters, this requirement was easily satisfied by not furnishing constructors in patterns with their parameter arguments. If the type constructor we do the case analysis on has indices, there is the additional complication that the value constructors for this type constructor may instantiate the index arguments. Thus there is a conflict with the preceding arguments of the defined function providing abstract arguments for the indices. Again, there is a simple general solution: The conflicting preceding arguments of the defined function are written with the underline symbol '_' and thus don't introduce variables, and the necessary instantiation of the function type is postponed until the instantiating constructor is reached. In the definition shown in Figure 19.3, the critical argument of $\mathsf{E}_{\vdash}$ that needs to be written as '_' in the defining equations is s in the head type $\forall As.\ A \vdash s \to pAs$ of $\mathsf{E}_{\vdash}$.

19.5 Induction on Derivations

We are now ready to prove interesting properties of the intuitionistic ND system using induction on derivations. We will carry out the inductions informally and leave it to reader to check (with Coq) that the informal proofs translate into formal proofs applying the eliminator $\mathsf{E}_{\vdash}$.

We start by defining a function translating derivations $A \vdash s$ into derivations $B \vdash s$ provided B contains every formula in A.

Fact 19.5.1 (Weakening) $A \vdash s \to A \subseteq B \to B \vdash s$.

Proof By induction on $A \vdash s$ with B quantified. All proof obligations are straightforward. We consider the constructor $\mathsf{I}_{\to}$. We have $A \subseteq B$ and a derivation $A, s \vdash t$, and we need a derivation $B \vdash (s \to t)$. Since $A, s \subseteq B, s$, the inductive hypothesis gives us a derivation $B, s \vdash t$. Thus $\mathsf{I}_{\to}$ gives us a derivation $B \vdash (s \to t)$. ∎

Next we show that premises of top level implications are interchangeable with assumptions.

Fact 19.5.2 (Implication) $A \vdash (s \to t) \iff A, s \vdash t$.

Proof Direction $\Leftarrow$ holds by $\mathsf{I}_{\to}$. For direction $\Rightarrow$ we assume $A \vdash (s \to t)$ and obtain $A, s \vdash (s \to t)$ with weakening. Now $\mathsf{A}_{\vdash}$ and $\mathsf{E}_{\to}$ yield $A, s \vdash t$. ∎

As a consequence, we can represent all assumptions of a derivation $A \vdash s$ as premises of implications at the right-hand side. To this purpose, we define a *reversion function* $A \cdot s$ with $[] \cdot t := t$ and $(s :: A) \cdot t := A \cdot (s \to t)$. For instance, we have $[s_1, s_2, s_3] \cdot t = s_3 \to s_2 \to s_1 \to t$.

Fact 19.5.3 (Reversion) $A \vdash s \Leftrightarrow \vdash A \cdot s$.

Proof By induction on A with s quantified using the implication lemma. ∎

A formula is **ground** if it contains no variable. We assume a recursively defined predicate $\mathsf{ground}\, s$ for groundness.

Fact 19.5.4 (Ground Prover) $\forall s.\ \mathsf{ground}\, s \to ([] \vdash s) + ([] \vdash \neg s)$.

Proof By induction on s using weakening. ∎

Exercise 19.5.5 Establish the following functions:

a) $A \vdash (s_1 \to s_2 \to t) \to A \vdash s_1 \to A \vdash s_2 \to A \vdash t$.

b) $\neg\neg s \in A \to A, s \vdash \bot \to A \vdash \bot$.

c) $A, s, \neg t \vdash \bot \to A \vdash \neg\neg(s \to t)$.

Exercise 19.5.6 Prove the following types.

a) $A \vdash ((\neg s \to \neg\neg\bot) \to \neg\neg s)$.

b) $A \vdash ((s \to \neg\neg t) \to \neg\neg(s \to t))$.

c) $A \vdash (\neg\neg(s \to t) \to \neg\neg s \to \neg\neg t)$.

d) $A \vdash (\neg\neg\neg s \to \neg s)$.

e) $A \vdash (\neg s \to \neg\neg\neg s)$.

Exercise 19.5.7 Prove $\forall s.\ \mathsf{ground}\, s \to \vdash (s \vee \neg s)$.

Exercise 19.5.8 Prove $\forall A s.\ \mathsf{ground}\, s \to A, s \vdash t \to A, \neg s \vdash t \to A \vdash t$.

Exercise 19.5.9 Prove the deduction laws for conjunctions and disjunctions:

a) $A \vdash (s \wedge t) \Leftrightarrow A \vdash s \times A \vdash t$

b) $A \vdash (s \vee t) \Leftrightarrow \forall u.\ A, s \vdash u \to A, t \vdash u \to A \vdash u$

19.6 Heyting Entailment

The proof techniques we have used so far do not suffice to show negative results about the intuitionistic ND system. By a negative result we mean a proof saying that a certain derivation type is empty, for instance,

$$\not\vdash \bot \qquad \not\vdash x \qquad \not\vdash (\neg\neg x \to x)$$

(we write $\not\vdash s$ for the proposition $([] \vdash s) \to \bot$). Speaking informally, the above propositions say that there a no derivations for falsity, atomic formulas, and the double negation law for atomic formulas.

A powerful technique for showing negative results is evaluation of formulas into a finite and ordered domain of so-called *truth values*. Evaluation into the boolean domain $0 < 1$ is well-known and suffices to disprove $\vdash \bot$ and $\vdash x$. To disprove $\vdash (\neg\neg x \to x)$, we need to switch to a three-valued domain $0 < 1 < 2$. Using the order of the truth values, we interpret conjunction as minimum and disjunction as maximum. Falsity is interpreted as the least truth value. Implication of truth values is interpreted as a comparison that in the positive case yields the greatest truth value 2 and in the negative case yields the second argument:

$$\mathsf{imp}\ a\,b \ :=\ \text{IF } a \le b \text{ THEN } 2 \text{ ELSE } b$$

Note that the given order-theoretic interpretations of the logical constants agree with the familiar boolean interpretations for the two-valued domain $0 < 1$. The order-theoretic evaluation of formulas originated around 1930 with the work of Arend Heyting on so-called Heyting algebras generalizing Boolean algebras.

We will work with the truth value domain $0 < 1 < 2$. Using evaluation into this domain we will define a predicate $A \models s$ called **Heyting entailment** for which we can show the implication

$$A \vdash s \ \to\ A \models s$$

known as **soundness**. Using soundness, we can disprove $\vdash (\neg\neg x \to x)$ by disproving $\models (\neg\neg x \to x)$. It is common to refer to $A \vdash s$ as a **syntactic entailment relation** and to $A \models s$ as a **semantic entailment relation**.

We represent our domain of **truth values** $0 < 1 < 2$ with an inductive type V and the order of truth values with a boolean function $a \le b$. As a matter of convenience, we will write the numbers 0, 1, 2 for the value constructors of V. An **assignment** is

a function $\alpha : \mathsf{N} \to \mathsf{V}$. We define **evaluation of formulas** $\mathcal{E}\alpha s$ as follows:

$$\begin{aligned}
\mathcal{E} &: (\mathsf{N} \to \mathsf{V}) \to \mathsf{For} \to \mathsf{V} \\
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha \bot &:= 0 \\
\mathcal{E}\alpha(s \to t) &:= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha t \text{ THEN } 2 \text{ ELSE } \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \wedge t) &:= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha t \text{ THEN } \mathcal{E}\alpha s \text{ ELSE } \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \vee t) &:= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha t \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathcal{E}\alpha s
\end{aligned}$$

Note that conjunction is interpreted as minimum, disjunction is interpreted as maximum, and implications is interpreted as the test imp described above.

We define **evaluation of contexts** $\mathcal{E}\alpha A$ such that the empty context yields the greatest truth value and nonempty contexts yield the minimal truth value their formulas evaluate to:

$$\begin{aligned}
\mathcal{E} &: (\mathsf{N} \to \mathsf{V}) \to \mathcal{L}(\mathsf{For}) \to \mathsf{V} \\
\mathcal{E}\alpha[] &= 2 \\
\mathcal{E}\alpha(s :: A) &= \text{IF } \mathcal{E}\alpha s \le \mathcal{E}\alpha A \text{ THEN } \mathcal{E}\alpha s \text{ ELSE } \mathcal{E}\alpha A
\end{aligned}$$

You may think of a context as a conjunction of formulas where the empty context yields the greatest truth value.

We now define **Heyting entailment** as

$$A \vDash s \;:=\; \forall \alpha.\ (\mathcal{E}\alpha A \le \mathcal{E}\alpha s) = \mathbf{T}$$

There are some clever design decisions in our definition of Heyting entailment, contributed by Chad. E. Brown, which much simplify the soundness proof on paper and with Coq. Of particular importance is the separate evaluation of contexts.

With our definitions we have the computational equalities

$$\begin{aligned}
\mathcal{E}(\lambda_.1)\bot &= 0 \\
\mathcal{E}(\lambda_.1)x &= 1 \\
\mathcal{E}(\lambda_.1)(\neg x) &= 0 \\
\mathcal{E}(\lambda_.1)(\neg\neg x) &= 2 \\
\mathcal{E}(\lambda_.1)(\neg\neg x \to x) &= 1
\end{aligned}$$

So once we have soundness, we can disprove $\vdash \bot$, $\vdash x$, and $\vdash (\neg\neg x \to x)$.

Lemma 19.6.1 $s \in A \to A \vDash s$.

Proof By induction on A. The base case is trivial. For the cons case we distinguish two cases.

For the first case we need to show $s :: A \vDash s$. We fix an assignment, and do case analysis on $\mathcal{E}\alpha s$ and $\mathcal{E}\alpha A$ (9 cases). Each case is straightforward.

For the second case we need to show $(\mathcal{E}\alpha(u :: A) \le \mathcal{E}\alpha s) = \mathbf{T}$ where $s \in A$. The inductive hypothesis gives us $(\mathcal{E}\alpha A \le \mathcal{E}\alpha s) = \mathbf{T}$. We do case analysis on $\mathcal{E}\alpha s$, $\mathcal{E}\alpha u$, and $\mathcal{E}\alpha A$ (27 cases). Each case is straightforward. ∎

Fact 19.6.2 (Soundness) $A \vdash s \to A \vDash s$.

Proof By induction on $A \vdash s$. The case for the assumption rule follows with Lemma 19.6.1. The case for the explosion rule fixes an assignment α, obtains $(\mathcal{E}\alpha A \le 0) = \mathbf{T}$ by the inductive hypothesis, and shows $(\mathcal{E}\alpha A \le \mathcal{E}\alpha s) = \mathbf{T}$ by case analysis on $\mathcal{E}\alpha s$ and $\mathcal{E}\alpha A$ (9 cases). All cases are straightforward. The remaining rules are similar. ∎

Corollary 19.6.3 $\vdash s \to \mathcal{E}\alpha s = 2$.

A formula s is **independent in** $\vdash$ if one can prove both $(\vdash s) \to \bot$ and $(\vdash \neg s) \to \bot$.

Corollary 19.6.4 (Independence) $\neg\neg x \to x$ and $x \vee \neg x$ are independent in $\vdash$.

Proof Follows with Corollary 19.6.3 and $\alpha n := 1$. ∎

Corollary 19.6.5 (Consistency) $\nvdash \bot$.

Exercise 19.6.6 Show that x, $\neg x$, and $((x \to y) \to x) \to x$ are independent in $\vdash$.

Exercise 19.6.7 Show $\neg\forall s.\, ((\vdash (\neg\neg s \to s)) \to \bot)$.

Exercise 19.6.8 Show $A \vDash \bot \longleftrightarrow \forall\alpha.\, \mathcal{E}\alpha A = 0$ and $\vDash s \longleftrightarrow \forall\alpha.\, \mathcal{E}\alpha s = 2$.

19.7 Classical ND System

The classical ND system is obtained from the intuitionistic ND system by replacing the **explosion rule**

$$\frac{A \vdash \bot}{A \vdash s}$$

with the proof by **contradiction rule**:

$$\frac{A, \neg s \vdash \bot}{A \vdash s}$$

Formally, we accommodate the classical ND system with a separate derivation type constructor

$$\dot{\vdash} : \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{T}$$

with separate value constructors. Classical ND can prove the double negation law.

Fact 19.7.1 (Double Negation) $A \dot{\vdash} (\neg\neg s \to s)$.

Proof Straightforward using the contradiction rule. ∎

Fact 19.7.2 (Cut) $A \dot{\vdash} s \to A, s \dot{\vdash} t \to A \dot{\vdash} t$.

Proof Same as for the intuitionistic system. ∎

Fact 19.7.3 (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 $(\forall B.\ A, \neg s \subseteq B \to B \dot{\vdash} \bot) \to A \subseteq B \to B \dot{\vdash} s$ for the contradiction rule must be checked. Straightforward with the contradiction rule. ∎

The classical system can prove the explosion rule. Thus every intuitionistic derivation $A \vdash s$ can be translated into a classical derivation $A \dot{\vdash} s$.

Fact 19.7.4 (Explosion) $A \dot{\vdash} \bot \to A \dot{\vdash} s$.

Proof By contradiction and weakening. ∎

Fact 19.7.5 (Translation) $A \vdash s \to A \dot{\vdash} s$.

Proof By induction on $A \vdash s$ using the explosion lemma for the explosion rule. ∎

Fact 19.7.6 (Implication) $A, s \dot{\vdash} t \Leftrightarrow A \dot{\vdash} (s \to t)$.

Proof Same proof as for the intuitionistic system. ∎

Because of the contradiction rule the classical system has the distinguished property that every proof problem can be turned into a refutation problem.

Fact 19.7.7 (Refutation) $A \dot{\vdash} s \Leftrightarrow A, \neg s \dot{\vdash} \bot$.

Proof Direction ⇒ follows with weakening. Direction ⇐ follows with the contradiction rule. ∎

While the refutation lemma tells us that classical ND can represent all information in the context, the implication lemmas tell us that both intuitionistic and classical ND can represent all information in the claim.

Exercise 19.7.8 Show $\dot{\vdash} s \vee \neg s$ and $\dot{\vdash} ((s \to t) \to s) \to s$.

Exercise 19.7.9 Show that classical ND is not sound for Heyting entailment: $\neg(\forall As.\ A \dot{\vdash} s \ \to\ A \vDash s)$.

Exercise 19.7.10 Prove the deduction laws for conjunctions and disjunctions:

a) $A \dot{\vdash} (s \wedge t) \ \Leftrightarrow\ A \dot{\vdash} s \times A \dot{\vdash} t$

b) $A \dot{\vdash} (s \vee t) \ \Leftrightarrow\ \forall u.\ A, s \dot{\vdash} u \ \to\ A, t \dot{\vdash} u \ \to\ A \dot{\vdash} u$

Exercise 19.7.11 Show that classical ND can express conjunction and disjunction with implication and falsity. To do so, define a translation function fst not using conjunction and prove $\dot{\vdash} (s \wedge t \to fst)$ and $\dot{\vdash} (fst \to s \wedge t)$. Do the same for disjunction.

19.8 Glivenko's Theorem

It turns out that a formula is classically provable if and only if its double negation is intuitionistically provable. Thus a classical provability problem can be reduced to an intuitionistic provability problem.

Lemma 19.8.1 $A \dot{\vdash} s \ \to\ A \vdash \neg\neg s$.

Proof By induction on $A \dot{\vdash} s$. This yields the following proof obligations.

1. $s \in A \ \to\ A \vdash \neg\neg s$.
2. $A, \neg s \vdash \neg\neg\bot \ \to\ A \vdash \neg\neg s$.
3. $A, s \vdash \neg\neg t \ \to\ A \vdash \neg\neg(s \to t)$.
4. $A \vdash \neg\neg(s \to t) \ \to\ A \vdash \neg\neg s \ \to\ A \vdash \neg\neg t$.

The obligations for conjunctions and disjunctions are omitted. The proofs are routine with Exercise 19.5.6 and the implication lemma 19.5.2. ∎

Theorem 19.8.2 (Glivenko) $A \dot{\vdash} s \ \Leftrightarrow\ A \vdash \neg\neg s$.

Proof Direction $\Rightarrow$ follows with Lemma 19.8.1. Direction $\Leftarrow$ follows with translation (19.7.5) and double negation (19.7.1). ∎

Corollary 19.8.3
Classical ND reduces to intuitionistic ND refutation: $A \dot{\vdash} s \Leftrightarrow A, \neg s \vdash \bot$.

Corollary 19.8.4 Intuitionistic and classical refutation agree: $A \vdash \bot \Leftrightarrow A \dot{\vdash} \bot$.

Proof Direction $\Rightarrow$ follows with translation 19.7.5. Direction $\leftarrow$ follows with Glivenko's theorem and the bottom law 19.3.3. ∎

Corollary 19.8.5 Classical ND is consistent: $(\dot{\vdash} \bot) \to \bot$.

Proof Let $\dot{\vdash} \bot$. By consistency of intuitionistic ND (19.6.5) it suffices to prove $\vdash \bot$. Follows by Glivenko and the bottom law 19.3.3. ∎

Exercise 19.8.6 Disprove $\dot{\vdash} x$ and $\dot{\vdash} \neg x$.

Exercise 19.8.7 Disprove $A \dot{\vdash} (s \vee t) \Leftrightarrow A \dot{\vdash} s \vee A \dot{\vdash} t$.

19.9 Boolean Entailment

Boolean evaluation evaluates formulas into a two-valued domain. We choose the type B of boolean values and fix the order $\mathbf{F} < \mathbf{T}$. Specializing the ideas we have seen for the three-valued Heyting evaluation of Section 19.6, we define **boolean evaluation** and **boolean entailment** as shown in Figure 19.4. Note that the definitions in the Figure respect the familiar unordered view of boolean evaluation and the ordered view coming with Heyting evaluation. We will call functions $\alpha : \mathsf{N} \to \mathsf{B}$ **boolean assignments**.

It turns out that classical ND and boolean entailment agree: $A \dot{\vdash} s \Leftrightarrow A \dot{\models} s$. The direction from ND entailment to boolean entailment is known as *soundness*, and the direction from boolean entailment to ND entailment is known as *completeness*. Recall that classical ND is not sound for Heyting entailment (Exercise 19.7.9).

The soundness proof for boolean entailment is analogous to the soundness proof for Heyting entailment. In Coq, the exactly same proof scripts can be used.

Lemma 19.9.1 $s \in A \to A \dot{\models} s$.

Proof By induction on A. ∎

Fact 19.9.2 (Soundness) $A \dot{\vdash} s \to A \dot{\models} s$.

Proof By induction on $A \dot{\vdash} s$ using Lemma 19.9.1 for the assumption rule. ∎

We can now give a consistency proof for classical ND that does not make use of intuitionistic ND.

Corollary 19.9.3 Classical ND is consistent: $(\dot{\vdash} \bot) \to \bot$.

Proof Follows with soundness and $\mathcal{E}\alpha\bot = 0$. ∎

$$\begin{aligned}
&\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathsf{B}\\
&\mathcal{E}\alpha x \;:=\; \alpha x\\
&\mathcal{E}\alpha \bot \;:=\; \mathbf{F}\\
&\mathcal{E}\alpha(s \to t) \;:=\; \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathbf{T}\\
&\mathcal{E}\alpha(s \wedge t) \;:=\; \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathbf{F}\\
&\mathcal{E}\alpha(s \vee t) \;:=\; \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathbf{T} \text{ ELSE } \mathcal{E}\alpha t\\
&\\
&\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathcal{L}(\mathsf{For}) \to \mathsf{B}\\
&\mathcal{E}\alpha([]) \;:=\; \mathbf{T}\\
&\mathcal{E}\alpha(s :: A) \;:=\; \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha A \text{ ELSE } \mathbf{F}\\
&\\
&A \dot{\models} s \;:=\; \forall \alpha.\ \text{IF } \mathcal{E}\alpha A \text{ THEN } \mathcal{E}\alpha s = \mathbf{T} \text{ ELSE } \top
\end{aligned}$$

Figure 19.4: Boolean evaluation and entailment

Fact 19.9.4

1. $A \dot{\models} s \longleftrightarrow \forall \alpha.\ \mathcal{E}\alpha A = \mathbf{T} \to \mathcal{E}\alpha s = \mathbf{T}$
2. $A \dot{\models} \bot \longleftrightarrow \forall \alpha.\ \mathcal{E}\alpha A = \mathbf{F}$
3. $\mathcal{E}\alpha A = \mathbf{T} \longleftrightarrow \forall s \in A.\ \mathcal{E}\alpha s = \mathbf{T}$

Proof Claims (1) and (2) have straightforward proofs with boolean case analysis. Claim (3) follows by induction on A and boolean case analysis. ∎

Exercise 19.9.5 Show that x and $\neg x$ are independent in $\dot{\vdash}$.

Exercise 19.9.6 Give boolean assignments showing that $\neg\neg\neg x$ is independent in $\dot{\vdash}$.

Exercise 19.9.7 Show the following properties of boolean entailment:

a) $A \dot{\models} s \longleftrightarrow A, \neg s \dot{\models} \bot$

b) $A \dot{\models} s \longleftrightarrow \dot{\models} A \cdot s$

c) $A \dot{\models} (s \to t) \longleftrightarrow A, s \dot{\models} t$

Exercise 19.9.8 Show $(\forall st.\ \dot{\vdash}(s \vee t) \to (\dot{\vdash} s) + (\dot{\vdash} t)) \to \bot$.

Exercise 19.9.9 Show that boolean entailment can express conjunction and disjunction with implication and falsity. To do so, define a translation function fst not using conjunction and prove $\dot{\models}(s \wedge t \to fst)$ and $\dot{\models}(fst \to s \wedge t)$. Do the same for disjunction.

19.10 Certifying Boolean Solvers

A **certifying boolean solver** is a function

$$\forall A.\ (\Sigma\,\alpha.\ \mathcal{E}\alpha A = \mathbf{T}) + (A \dot{\vdash} \bot)$$

that given a list of formulas yields either an assignment satisfying all formulas in the list or a classical ND refutation of the list. Note that a refutation $A \dot{\vdash} \bot$ certifies that there is no boolean assignment satisfying all formulas in A (Fact 19.9.4 (2)).

Fact 19.10.1
Given a certifying boolean solver, classical ND is complete and decidable:

1. $A \dot{\models} s \;\to\; A \dot{\vdash} s$ (completeness)
2. $\mathcal{D}(A \dot{\vdash} s)$ (decidability)

Proof By the refutation laws (Fact 19.7.7 and Exercise 19.9.7) it suffices to show the claims for $s = \bot$.

For (1) we assume $A \dot{\models} \bot$ and show $A \dot{\vdash} \bot$. The certifying boolean solver gives us either the claim or $\mathcal{E}\alpha A = \mathbf{T}$. The second case yields a contradiction with Fact 19.9.4 (2).

For (2) the certifying boolean solver gives us either the claim or $\mathcal{E}\alpha A = \mathbf{T}$. In the second case we assume $A \dot{\vdash} \bot$ and obtain a contradiction with soundness and Fact 19.9.4 (2). ∎

In Chapter 22, we will construct a certifying boolean solver using the tableau method.

Exercise 19.10.2 (Refutation predicates)
A refutation predicate is a predicate $\rho : \mathcal{L}(\mathsf{For}) \to \mathbb{P}$ satisfying the following rules:

$$\begin{aligned}
\rho(s :: A \dashv B) &\to \rho(A \dashv s :: B) && \text{rotation}\\
\rho(x :: \neg x :: A) && \text{clash}\\
\rho(\bot :: A) && \text{falsity}\\
\rho(\neg s :: A) \to \rho(t :: A) &\to \rho((s \to t) :: A) && \text{implication}\\
\rho(s :: \neg t :: A) &\to \rho(\neg(s \to t) :: A)) && \text{implication}^-\\
\rho(s :: t :: A) &\to \rho((s \wedge t) :: A) && \text{conjunction}\\
\rho(\neg s :: A) \to \rho(\neg t :: A) &\to \rho(\neg(s \wedge t) :: A) && \text{conjunction}^-\\
\rho(s :: A) \to \rho(t :: A) &\to \rho((s \vee t) :: A) && \text{disjunction}\\
\rho(\neg s :: \neg t :: A) &\to \rho(\neg(s \vee t) :: A) && \text{disjunction}^-
\end{aligned}$$

We will show in Chapter 22 that there is a certifying solver

$$\forall A.\ (\Sigma\,\alpha.\ \mathcal{E}\alpha A = \mathbf{T}) + \rho A$$

that given a refutation predicate ρ yields either a satisfying assignment or a refutation using ρ.

a) Show that $\lambda A.\ A \vdash \bot$ is a refutation predicate.

b) Show that $\lambda A. \forall \alpha.\ \mathcal{E}\alpha A = \mathbf{F}$ is a refutation predicate.

Exercise 19.10.3 Assume a certifying boolean solver and show $\mathcal{D}(A \models s)$.

19.11 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:

$$\begin{aligned}
\theta\cdot x &:= \theta x\\
\theta\cdot \bot &:= \bot\\
\theta\cdot(s \to t) &:= \theta\cdot s \to \theta\cdot t\\
\theta\cdot(s \wedge t) &:= \theta\cdot s \wedge \theta\cdot t\\
\theta\cdot(s \vee t) &:= \theta\cdot s \vee \theta\cdot t\\
\\
\theta\cdot[] &:= []\\
\theta\cdot(s :: A) &:= \theta\cdot s :: \theta\cdot A
\end{aligned}$$

We will write θs and θA for $\theta\cdot s$ and $\theta\cdot A$.

We show that intuitionistic and classical ND provability are preserved under application of substitutions. This says that atomic formulas may serve as variables for formulas.

Fact 19.11.1 $s \in A \to \theta s \in \theta A$.

Proof By induction on A. ∎

Fact 19.11.2 $A \vdash s \to \theta A \vdash \theta s$ and $A \vdash\!\!\!\cdot\ s \to \theta A \vdash\!\!\!\cdot\ \theta s$.

Proof By induction on $A \vdash s$ and $A \vdash\!\!\!\cdot\ s$ using Fact 19.11.1. ∎

Next we show that Heyting and boolean entailment are preserved under application of substitutions.

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:* $\exists s.\ \not\Vdash s$.
5. *Substitutivity:* $A \Vdash s \to \theta A \Vdash \theta s$.
6. *Explosion:* $A \Vdash \bot \to A \Vdash s$.
7. *Implication:* $A \Vdash (s \to t) \longleftrightarrow A, s \Vdash t$.
8. *Conjunction:* $A \Vdash (s \wedge t) \longleftrightarrow A \Vdash s \wedge A \Vdash t$.
9. *Disjunction:* $A \Vdash (s \vee t) \longleftrightarrow \forall u.\ A, s \Vdash u \to A, t \Vdash u \to A \Vdash u$.

Figure 19.5: Requirements for entailment predicates

Lemma 19.11.3
$\mathcal{E}\alpha(\theta s) = \mathcal{E}(\lambda n.\mathcal{E}\alpha(\theta n))s$ holds both for Heyting and boolean evaluation.

Fact 19.11.4 $A \vDash s \to \theta A \vDash \theta s$ and $A \dot{\vDash} s \to \theta A \dot{\vDash} \theta s$.

Proof By induction on A using Lemma 19.11.3. ∎

19.12 Entailment Predicates

An **entailment predicate** is a predicate

$$\Vdash\colon \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{P}$$

satisfying the properties listed in Figure 19.5. 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 19.12.1 Intuitionistic ND ($A \vdash s$), classical ND ($A \dot{\vdash} s$), Heyting entailment ($A \vDash s$), and boolean entailment ($A \dot{\vDash} s$) are all entailment predicates.

Proof Follows with the results shown before. ∎

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 19.12.2 (Modus Ponens) $A \Vdash (s \to t) \to A \Vdash s \to A \Vdash t$.

Proof By implication and cut. ∎

Fact 19.12.3 $A \vdash s \to A \Vdash s$. That is, intuitionistic ND is a least entailment predicate.

Proof By induction on $A \vdash s$ using modus ponens. ∎

Fact 19.12.4 $\Vdash s \to \Vdash \neg s \to \bot$.

Proof Let $\Vdash s$ and $\Vdash \neg s$. By Fact 19.12.2 we have $\Vdash \bot$. By consistency and explosion we obtain a contradiction. ∎

Fact 19.12.5 (Reversion) $A \Vdash s \longleftrightarrow \Vdash A \cdot s$.

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: $\hat{\alpha} n := \text{IF } \alpha n \text{ THEN } \neg\bot \text{ ELSE } \bot$.

Lemma 19.12.6 (Tebbi) $\text{IF } \mathcal{E}\alpha s \text{ THEN } \Vdash \hat{\alpha} s \text{ ELSE } \Vdash \neg\hat{\alpha} s$.

Proof Induction on s using Fact 19.12.2 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 19.12.7 $\Vdash s \to \dot{\models} s$.

Proof Let $\Vdash s$ and α. We assume $\mathcal{E}\alpha s = \mathbf{F}$ 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 19.12.4. ∎

Theorem 19.12.8 (Sandwich)
Let $\Vdash$ be an entailment predicate. Then $A \vdash s \to A \Vdash s$ and $A \Vdash s \to A \dot{\models} s$.

Proof Claim 1 is Fact 19.12.3. Claim 2 follows with Lemma 19.12.7 and Facts 19.12.5 and 19.12.1. ∎

Exercise 19.12.9 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 19.12.10 Tebbi's lemma provides for a particularly elegant proof of Lemma 19.12.7. Verify that Lemma 19.12.7 can also be obtained from the facts (1) $\vdash \hat{\alpha} s \vee \vdash \neg\hat{\alpha} s$ and (2) $\dot{\models} \hat{\alpha} s \to \mathcal{E}\alpha s = \mathbf{T}$ using Facts 19.12.3 and 19.12.4.

19.13 Notes

The study of natural deduction originated in the 1930's with the work of Gerhard Gentzen [5, 6] and Stanisław Jaśkowski [7]. The standard text on natural deduction and proof theory is Troelstra and Schwichtenberg [12].

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.

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.

20 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 13 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 13 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 20.9 and 20.10 address advanced topics and may be skipped on first reading.

20.1 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]

$$\begin{aligned}&\mathsf{E}_{\mathsf{zero}} : \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to \forall x.\, \mathsf{zero}\ x \to px\\ &\mathsf{E}_{\mathsf{zero}}\ pa\ \_\ \mathsf{Z} := a \qquad : p0\end{aligned}$$

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

$$\mathsf{E}_{\mathsf{zero}}\ pa\ \_\ \mathsf{Z} : p\,0 \quad \text{and} \quad \mathsf{E}_{\mathsf{zero}}\ pa\,0\,\mathsf{Z} : p\,0$$

validating the right hand side of the defining equation.

We call the variable x in the type of $\mathsf{E}_{\mathsf{zero}}$ an **index variable** since it is determined as an index of an inductive predicate.

Following our convention for eliminators, we refer to $\mathsf{E}_{\mathsf{zero}}$ as **elimination lemma** when we use it as a declared constant. Nowhere in this chapter will we exploit that $\mathsf{E}_{\mathsf{zero}}$ is a defined function.

Exercise 20.1.1 Prove the following facts.

a) $\mathsf{zero}\ x \longleftrightarrow x = 0$

b) $\neg\mathsf{zero}(\mathsf{S}x)$

c) $\neg\mathsf{zero}\,1$

d) $\mathcal{D}(\mathsf{zero}\,x)$

Exercise 20.1.2 Prove the following impredicative characterization for zero:

$$\mathsf{zero}\ x \longleftrightarrow \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to px$$

Exercise 20.1.3 Convince yourself that $\mathsf{E}_{\mathsf{zero}}(\lambda x.\ \textsc{if}\ x\ \textsc{then}\ \top\ \textsc{else}\ \bot)\,\mathsf{I}\,5$ is a proof of $\neg\mathsf{zero}\,5$.

[1] Note that zero is not affected by the elim restriction.

20.2 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]

$$\begin{aligned}&\mathsf{E}_{\mathsf{eq}}: \forall X^{\mathbb{T}}\, x^{X}\, p^{X\to\mathbb{T}}.\ px \to \forall y.\ \mathsf{eq} X x y \to py\\ &\mathsf{E}_{\mathsf{eq}}\ X x p a\ \_\ (\mathsf{Q}\ \_\ \_) := a \qquad : px\end{aligned}$$

The **flow of information during type checking** the left-hand side of the defining equation of 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 $\mathsf{eq}\,Xxx$ of $\mathsf{Q}\,Xx$, which is x.

Using the eliminator E_{eq}, we can now define the rewriting law:

$$\begin{aligned}&\mathsf{R}: \forall X^{\mathbb{T}}\, x^{X}\, y^{X}\, p^{X\to\mathbb{P}}.\ \mathsf{eq}\,Xxy \to px \to py\\ &:= \lambda X x y p h a.\ \mathsf{E}_{\mathsf{eq}}\,Xxpayh\end{aligned}$$

Exercise 20.2.1 Prove $\forall X^{\mathbb{T}}\, x^{X}\, y^{X}.\ \mathsf{eq} X x y \longleftrightarrow \forall p^{X\to\mathbb{P}}.\ px \to py$.

20.3 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

$$\begin{aligned}\mathsf{even}&: \mathsf{N} \to \mathbb{P}\\ \mathsf{even}_{\mathsf{B}}&: \mathsf{even}\ 0\\ \mathsf{even}_{\mathsf{S}}&: \forall n.\ \mathsf{even}\ n \to \mathsf{even}(\mathsf{S}(\mathsf{S}n))\end{aligned}$$

[2] Note that eq is not affected by the elim restriction.

with two proof constructors giving us proof terms for exactly the even numbers:

$\mathsf{even}\ 0$	even_B
$\mathsf{even}\ 2$	$\mathsf{even}_\mathsf{S}\ 2\ \mathsf{even}_\mathsf{B}$
$\mathsf{even}\ 4$	$\mathsf{even}_\mathsf{S}\ 4\ (\mathsf{even}_\mathsf{S}\ 2\ \mathsf{even}_\mathsf{B})$
$\ldots$	$\ldots$

More generally, we can prove that every multiple of 2 is an even number:

$$\mathsf{even}(k \cdot 2) \tag{20.1}$$

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 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:

$$\begin{aligned}
&\mathsf{E}_\mathsf{even} : \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\ \mathsf{even}\ n \to pn \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.\ \mathsf{even}\ n \to pn \\
&\quad \mathsf{E}_\mathsf{even}\ p\ a\ f\ _\ \mathsf{even}_\mathsf{B} := a && : p0 \\
&\mathsf{E}_\mathsf{even}\ p\ a\ f\ _\ (\mathsf{even}_\mathsf{S}\ n'\ h) := f n' h\ (\mathsf{E}_\mathsf{even}\ p\ a\ f\ n'\ h) && : p(\mathsf{S}(\mathsf{S}n'))
\end{aligned}$$

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 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 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 even_S is recursive so that it can provide the **inductive hypothesis** pn in the clause for even_S.

When we translate the equational definition of E_even into a computational definition

$$\begin{aligned}
&\mathsf{E}_\mathsf{even} : \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\ \mathsf{even}\ n \to pn \to p(\mathsf{S}(\mathsf{S}n))) \to \forall n.\ \mathsf{even}\ n \to pn \\
&\quad := \lambda p a f.\ \textsc{fix}\ F n h.\ \textsc{match}\ h\ [\ \mathsf{even}_\mathsf{B} \Rightarrow a \mid \mathsf{even}_\mathsf{S}\ n' h' \Rightarrow f n' h' (F n' h')\ ]
\end{aligned}$$

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 20.3.1 (Impredicative Characterization) Prove the equivalence

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

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 20.3.2 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 $\mathsf{even}_{\mathsf{S}}$. 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 20.3.3 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 20.3.4 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 20.1: Proof diagram for an inductive proof with $\mathsf{E}_{\mathsf{even}}$

20.4 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. even 36).

We explain the idea with a proof of the proposition

$$\forall n.\ \mathsf{even}\ n \to\ \exists\ k.\ n = k \cdot 2 \tag{20.2}$$

The formal proof appears as a proof diagram in Figure 20.1. Informally, we say that we prove (20.2) by induction on the derivation of even n. If 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 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 20.1 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 20.4.1 $\mathsf{even}\ n \longleftrightarrow\ \exists\ k.\ n = k \cdot 2$.

Proof Follows with (20.2) and (20.1). ∎

We may see the equivalence as a **specification** for the inductive predicate even. We call the two directions of the equivalence **soundness** (→) and **completeness** (←). 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 20.4.2 Define an inductive predicate odd and show that is satisfies the specification $\mathsf{odd}\ x \longleftrightarrow \exists k.\ x = \mathsf{S}(k \cdot 2)$.

Exercise 20.4.3 Give specifications for the inductive predicates zero and eq and prove their corrrectness.

Exercise 20.4.4 Define an inductive proposition $\mathsf{F} : \mathbb{P}$ with a single recursive proof constructor $\mathsf{L} : \mathsf{F} \to \mathsf{F}$ and show $\mathsf{F} \longleftrightarrow \bot$.

Exercise 20.4.5 Prove $\mathcal{D}(\mathsf{even}\, x)$ using the division theorem.

20.5 Inversion Lemmas and Unfolding

Proving negative facts about even such as

$$\neg\mathsf{even}\ 1 \tag{20.3}$$

$$\neg\mathsf{even}\ 3 \tag{20.4}$$

$$\neg\mathsf{even}\ n \to \neg\mathsf{even}\ (\mathsf{S}(\mathsf{S}n)) \tag{20.5}$$

$$\neg\mathsf{even}\ (\mathsf{S}(n \cdot 2)) \tag{20.6}$$

takes insight and a technique called unfolding. Clearly, (20.3) and (20.4) both follow from (20.6). Moreover, (20.6) follows by induction on n using (20.3) for the base case and (20.5) for the successor case. Finally, (20.5) follows from the positive fact

$$\mathsf{even}\ (\mathsf{S}(\mathsf{S}n)) \to \mathsf{even}\ n \tag{20.7}$$

We are thus left with (20.3) and (20.7), which we will refer to as **inversion lemmas**. Note that (20.7) is in fact the converse of the proof constructor even_S.

For (20.3) it is best to prove the generalized fact

$$\mathsf{even}\ k \to k = 1 \to \bot \tag{20.8}$$

which follows with the elimination lemma using $0 \neq 1$ and $\mathsf{S}(\mathsf{S}k) \neq 1$.

For (20.7) it is again best to prove the generalized fact

$$\mathsf{even}\ k \to k = \mathsf{S}(\mathsf{S}n) \to \mathsf{even}\ n \tag{20.9}$$

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 (20.8) and (20.9) used for proving the inversion lemmas (20.3) and (20.7) 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]

20.6 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 20.6.1 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 (20.3) and (20.7). ∎

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 20.6.2 $(\neg\,\mathsf{even}\,n \to \mathsf{even}(\mathsf{S}n)) \wedge (\neg\,\mathsf{even}(\mathsf{S}n) \to \mathsf{even}\,n)$.

Proof By induction on n using (20.3) and (20.7). ∎

[3] The tactic depelim comes with the Equations package supporting definition of functions with equations and well-founded recursion.

Fact 20.6.3 $\mathcal{D}\,(\mathsf{even}\,n)$.

Proof By induction on n using Facts 20.6.1, 20.6.2, and (20.7). ∎

Exercise 20.6.4 Prove the following facts about even.

a) $\mathsf{even}\,x \to \mathsf{even}\,y \to \mathsf{even}(x+y)$

b) $\mathsf{even}\,x \to \mathsf{even}(x+y) \to \mathsf{even}\,y$

20.7 Linear Order on Numbers

Coq defines the linear order on numbers inductively:

$$\begin{aligned}&\mathsf{le}: \mathsf{N} \to \mathsf{N} \to \mathbb{P}\\ &\mathsf{le}_\mathsf{B}: \forall x.\, \mathsf{le}\,xx\\ &\mathsf{le}_\mathsf{S}: \forall xy.\, \mathsf{le}\,xy \to \mathsf{le}\,x(\mathsf{S}y)\end{aligned}$$

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

$$\frac{}{\mathsf{le}\,xx} \qquad\qquad \frac{\mathsf{le}\,xy}{\mathsf{le}\,x(\mathsf{S}y)}$$

Note that the proof rules express basics facts about the linear order on numbers. Thus every proposition $\mathsf{le}\,xy$ 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 20.7.1 $\mathsf{le}\,xy \longleftrightarrow \exists k.\, k + x = y$.

Proof The direction → is by induction on the derivation of $\mathsf{le}\,xy$. 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 ← we show $\mathsf{le}\,x(k+x)$ by induction on k. Straightforward. ∎

We have shown $\mathsf{le}\,xy \longleftrightarrow \exists k.\, k + x = y$ rather than $\mathsf{le}\,xy \longleftrightarrow \exists k.\, x + k = y$ so that we don't need the commutativity of +.

Exercise 20.7.2 Define an eliminator for le that suffices for the induction used for the direction → of Fact 20.7.1.

Exercise 20.7.3 Prove the following inversion lemma:
$\forall xy.\, \mathsf{le}\,xy \to x = y \vee \exists y'.\, y = \mathsf{S}y' \wedge \mathsf{le}\,xy'$.

Exercise 20.7.4 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 $\mathsf{le}'\, xy \longleftrightarrow \exists k.\, x + k = y$.

c) Prove $\mathsf{le}'\, xy \longleftrightarrow \mathsf{le}\, xy$.

Exercise 20.7.5 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 20.7.1. 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. $x < y \vee x = y \vee y < x$
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).

20.8 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 20.8.1 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) $Dxy \longleftrightarrow x = 2 \cdot y$

b) $Mxyz \longleftrightarrow x = 3 \cdot y + z \wedge z \leq 2$

c) $Upn \longleftrightarrow \exists k.\, k \geq n \wedge pk \qquad (p : \mathsf{N} \to \mathbb{P})$

d) $Lpn \longleftrightarrow \exists k.\, k \leq n \wedge pk \qquad (p : \mathsf{N} \to \mathbb{P})$

20.9 Pureness of zero

We will now prove the proposition

$$\forall h^{\mathsf{zero}\,0}.\ h = \mathsf{Z} \tag{20.10}$$

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

$$\hat{\mathsf{E}}_{\mathsf{zero}} :\ \forall p^{\forall x.\ \mathsf{zero}\,x \to \mathbb{T}}.\ p0\mathsf{Z} \to \forall xh.\, pxh$$
$$\hat{\mathsf{E}}_{\mathsf{zero}}\ pa\ \_\ \mathsf{Z} :=\ a \qquad : p0\mathsf{Z}$$

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 (20.10) 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:

$$\begin{aligned} p &: \ \forall x^{\mathsf{N}}.\ \mathsf{zero}\,x \to \mathbb{P} \\ p\,0\,h &:= \ (h = \mathsf{Z}) \\ p\,(\mathsf{S}\_)\,h &:= \ \top \end{aligned}$$

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 20.9.1 Explain why $\forall x^{\mathsf{N}}\,h^{\mathsf{zero}\,x}.\ h = \mathsf{Z}$ does not type check.

Exercise 20.9.2 Define $\mathsf{E}_{\mathsf{zero}}$ with $\hat{\mathsf{E}}_{\mathsf{zero}}$.

Exercise 20.9.3 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 20.9.4 Prove $\mathsf{pure}(\mathsf{zero}\,x)$.

20.10 Axiom K

Axiom K is the proposition

$$\mathsf{K} \ := \ \forall X^{\mathbb{T}}\,x^{X}\,p^{\mathsf{eq}\,X\,x\,x \to \mathbb{P}}.\ p\,(\mathsf{Q}\,X\,x) \to \forall h.\,p\,h.$$

stating that $\mathsf{Q}\,X\,x$ is the only the proof of $\mathsf{eq}\,X\,x\,x$. 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 20.9. Following the proof for zero, we may try to prove

$$\forall X^{\mathbb{T}}\,x^{X}\,h^{\mathsf{eq}\,X\,x\,x}.\ h = \mathsf{Q}\,X\,x$$

which is equivalent to K. Defining a full eliminator for eq is not difficult:

$$\begin{aligned} \hat{\mathsf{E}}_{\mathsf{eq}} &: \ \forall X^{\mathbb{T}}\,x^{X}\,p^{\forall y.\ \mathsf{eq}\,X\,x\,y \to \mathbb{T}}.\ p\,x\,(\mathsf{Q}\,X\,x) \to \forall y h.\ p\,y\,h \\ \hat{\mathsf{E}}_{\mathsf{eq}}\,X\,x\,p\,a\,\_\,(\mathsf{Q}\,\_\,\_) &:= \ a \qquad : p\,x\,(\mathsf{Q}\,X\,x) \end{aligned}$$

The crux now is that we cannot find a predicate p and a proof a such that

$$\hat{\mathsf{E}}_{\mathsf{eq}}\ X x p a x\ :\ \forall h^{\mathsf{eq}Xxx}.\ h = \mathsf{Q}Xx$$

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 20.10.1 Prove that K is equivalent to $\forall X^{\mathbb{T}}\, x^X\, h^{\mathsf{eq}Xxx}.\ h = \mathsf{Q}Xx$.

Exercise 20.10.2 Prove that PI implies K. See Section 18.3 for the definition of PI.

Exercise 20.10.3 Define E_{eq} with $\hat{\mathsf{E}}_{\mathsf{eq}}$.

Exercise 20.10.4 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.

20.11 Summary

We have defined inductive predicates satisfying the following specifications:

$$\begin{aligned}
\mathsf{zero}\, x &\longleftrightarrow x = 0\\
\mathsf{eq}\, X x y &\longleftrightarrow \forall p^{X\to\mathbb{P}}.\ p x \to p y\\
\mathsf{even}\, n &\longleftrightarrow \exists k.\ x = 2 \cdot k\\
\mathsf{le}\, x y &\longleftrightarrow \exists k.\ x + k = y
\end{aligned}$$

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.

21 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.

21.1 Characterizations of Excluded Middle

Recall that excluded middle

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

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

1. $\forall X^{\mathbb{P}}.\ X \vee \neg X$ *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. $\forall X^{\mathbb{P}}.\ (X \longleftrightarrow \top) \vee (X \longleftrightarrow \bot)$

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 $X \mapsto (X \vee \neg X)$ and $Y \mapsto \bot$ we can assume $\neg(X \vee \neg X)$ and prove $X \vee \neg X$. We assume X and prove $\bot$. Straightforward since we have $\neg(X \vee \neg X)$. ∎

$$\begin{array}{rcll} \neg(X \wedge Y) & \longleftrightarrow & \neg X \vee \neg Y & \text{de Morgan} \\ \neg(\forall a.pa) & \longleftrightarrow & \exists a.\neg pa & \text{de Morgan} \\ (\neg X \to \neg Y) & \longleftrightarrow & (Y \to X) & \text{contraposition} \\ (X \to Y) & \longleftrightarrow & \neg X \vee Y & \text{classical implication} \end{array}$$

Figure 21.1: 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 21.1.2 (Counterexample) $\mathsf{XM} \longleftrightarrow \forall A^{\mathbb{T}} \forall p^{A\to\mathbb{P}}.\ (\forall a.pa) \vee \exists a.\neg pa.$

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 21.1 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

$$\begin{array}{rcll} \neg(X \vee Y) & \longleftrightarrow & \neg X \wedge \neg Y & \text{de Morgan} \\ \neg(\exists a.pa) & \longleftrightarrow & \forall a.\neg pa & \text{de Morgan} \end{array}$$

have constructive proofs.

Exercise 21.1.3

a) Prove the right-to-left directions of the equivalences in Figure 21.1.

b) Prove the left-to-right directions of the equivalences in Figure 21.1 using XM.

Exercise 21.1.4 Prove the following equivalences possibly using XM. In each case find out which direction needs XM.

$$\begin{aligned}\neg(\exists a.\neg pa) &\longleftrightarrow \forall a.pa\\ \neg(\exists a.\neg pa) &\longleftrightarrow \neg\neg\forall a.pa\\ \neg\neg(\exists a.pa) &\longleftrightarrow \neg\forall a.\neg pa\end{aligned}$$

Exercise 21.1.5 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 21.1.6 Make sure you can prove the de Morgan laws for disjunction and existential quantification (not using XM).

Exercise 21.1.7 Explain why Peirce's law and the double negation law are independent in Coq's type theory.

Exercise 21.1.8 (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 21.1.2.

21.2 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 21.2.1 The following **distribution laws** for double negation are provable:

$$\begin{aligned}
\neg\neg(X \to Y) &\longleftrightarrow (\neg\neg X \to \neg\neg Y)\\
\neg\neg(X \wedge Y) &\longleftrightarrow \neg\neg X \wedge \neg\neg Y\\
\neg\neg\top &\longleftrightarrow \top\\
\neg\neg\bot &\longleftrightarrow \bot
\end{aligned}$$

Exercise 21.2.2 Prove the equivalences of Fact 21.2.1.

Exercise 21.2.3 Prove the following propositions:

$$\begin{aligned}
\neg(X \wedge Y) &\longleftrightarrow \neg\neg(\neg X \vee \neg Y)\\
(\neg X \to \neg Y) &\longleftrightarrow \neg\neg(Y \to X)\\
(\neg X \to \neg Y) &\longleftrightarrow (Y \to \neg\neg X)\\
(X \to Y) &\to \neg\neg(\neg X \vee Y)
\end{aligned}$$

Exercise 21.2.4 Prove $\neg(\forall a.\neg pa) \longleftrightarrow \neg\neg\exists a.pa$.

Exercise 21.2.5 Prove the following implications:

$$\begin{aligned}
\neg\neg X \vee \neg\neg Y &\to \neg\neg(X \vee Y)\\
(\exists a.\neg\neg pa) &\to \neg\neg\exists a.pa\\
\neg\neg(\forall a.pa) &\to \forall a.\neg\neg pa
\end{aligned}$$

Convince yourself that the converse directions are not provable without excluded middle.

Exercise 21.2.6 Make sure you can prove the double negations of the following propositions:

$$\begin{gathered}
X \vee \neg X\\
\neg\neg X \to X\\
\neg(X \wedge Y) \to \neg X \vee \neg Y\\
(\neg X \to \neg Y) \to Y \to X\\
((X \to Y) \to X) \to X\\
(X \to Y) \to \neg X \vee Y\\
(X \to Y) \vee (Y \to X)
\end{gathered}$$

21.3 Definiteness and Stability

We define definiteness and stability of propositions as follows:

$$\begin{aligned} \mathsf{definite}\ X^{\mathbb{P}} &:= X \vee \neg X \\ \mathsf{stable}\ X^{\mathbb{P}} &:= \neg\neg X \to X \end{aligned}$$

Fact 21.3.1

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 21.3.2 Implication, conjunction, disjunction, and negation preserve definiteness:

1. $\mathsf{definite}\ X \to \mathsf{definite}\ Y \to \mathsf{definite}\ (X \to Y)$.
2. $\mathsf{definite}\ X \to \mathsf{definite}\ Y \to \mathsf{definite}\ (X \wedge Y)$.
3. $\mathsf{definite}\ X \to \mathsf{definite}\ Y \to \mathsf{definite}\ (X \vee Y)$.
4. $\mathsf{definite}\ X \to \mathsf{definite}\ (\neg X)$.

Fact 21.3.3 (Definite de Morgan) $\mathsf{definite}\ X \vee \mathsf{definite}\ Y \to \neg(X \wedge Y) \longleftrightarrow \neg X \vee \neg Y$.

Fact 21.3.4 Implication, conjunction, and universal quantification preserve stability:

1. $\mathsf{stable}\ Y \to \mathsf{stable}\ (X \to Y)$.
2. $\mathsf{stable}\ X \to \mathsf{stable}\ Y \to \mathsf{stable}\ (X \wedge Y)$.
3. $(\forall a.\ \mathsf{stable}\ (pa)) \to \mathsf{stable}\ (\forall a.pa)$.

Exercise 21.3.5 Prove the above facts.

Exercise 21.3.6 Prove $\mathsf{Markov} \longleftrightarrow \forall f^{\mathsf{N}\to\mathsf{B}}.\ \mathsf{stable}(\exists n.\ fn = \mathbf{T})$. The equivalence says that Markov is equivalent to stability of satisfiability of tests on numbers.

Exercise 21.3.7 Prove $(\forall a.\ \mathsf{stable}\ (pa)) \to \neg(\forall a.pa) \longleftrightarrow \neg\neg\exists a.\neg pa$.

Exercise 21.3.8 We define **classical variants** of conjunction, disjunction, and existential quantification:

$$\begin{aligned} X \wedge_c Y &:= (X \to Y \to \bot) \to \bot & \neg(X \to \neg Y) \\ X \vee_c Y &:= (X \to \bot) \to (Y \to \bot) \to \bot & \neg X \to \neg\neg Y \\ \exists_c a.pa &:= (\forall a.\, pa \to \bot) \to \bot & \neg(\forall a.\, \neg pa) \end{aligned}$$

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) $X \wedge Y \to X \wedge_c Y$ and $X \wedge_c Y \longleftrightarrow \neg\neg(X \wedge Y)$.

b) $X \vee Y \to X \vee_c Y$ and $X \vee_c Y \longleftrightarrow \neg\neg(X \vee Y)$.

c) $(\exists a.pa) \to \exists_c a.pa$ and $(\exists_c a.pa) \longleftrightarrow \neg\neg(\exists a.pa)$.

d) $X \vee_c \neg X$.

e) $\neg(X \wedge_c Y) \longleftrightarrow \neg X \vee_c \neg Y$.

f) $(\forall a.\ \mathsf{stable}\,(pa)) \ \to\ \neg(\forall a.pa) \longleftrightarrow \exists_c a.\neg pa$.

g) $X \wedge_c Y$, $X \vee_c Y$, and $\exists_c a.pa$ are stable.

22 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.

22.1 Boolean Operations

We will work with the boolean operations conjunction, disjunction, and negation, which we define as follows:

$$\mathbf{T} \,\&\, b = b \qquad \mathbf{T} \mid b = \mathbf{T} \qquad !\mathbf{T} = \mathbf{F}$$
$$\mathbf{F} \,\&\, b = \mathbf{F} \qquad \mathbf{F} \mid b = b \qquad !\mathbf{F} = \mathbf{T}$$

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

$$!(a \,\&\, b) = !a \mid !b \qquad !(a \mid b) = !a \,\&\, !b \qquad !!a = a$$

and conjunctions can be pushed below disjunctions with the distribution law

$$a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, b)$$

We will also make use of the negation law

$$b \wedge !\, b = \mathbf{F}$$

to eliminate conjunctions.

There are also the **reflection laws**

$$\begin{aligned} a \,\&\, b = \mathbf{T} &\longleftrightarrow a = \mathbf{T} \wedge b = \mathbf{T} \\ a \mid b = \mathbf{T} &\longleftrightarrow a = \mathbf{T} \vee b = \mathbf{T} \\ !\, a = \mathbf{T} &\longleftrightarrow \neg(a = \mathbf{T}) \end{aligned}$$

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 $\mathbf{T}$.

22.2 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 22.1. 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

$$\alpha \vDash s \;:=\; \mathcal{E}\alpha s = \mathbf{T}$$

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.

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= \mathbf{F} \\
\mathcal{E}\alpha(s \to t) &:= !\mathcal{E}\alpha s \mid \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \wedge t) &:= \mathcal{E}\alpha s \,\&\, \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \vee t) &:= \mathcal{E}\alpha s \mid \mathcal{E}\alpha t
\end{aligned}$$

Figure 22.1: 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 $\mathcal{E}\alpha s = \mathbf{T}$. Reasoning with boolean equations will be the main workhorse in our proofs.

Exercise 22.2.1 Convince yourself that the predicate $\alpha \vDash s$ is decidable.

Exercise 22.2.2 Verify a function translating formulas into equivalent formulas not containing conjunctions and disjunctions.

Exercise 22.2.3 Verify the reflection laws

$$\begin{aligned}
\alpha \vDash (s \wedge t) &\longleftrightarrow \alpha \vDash s \wedge \alpha \vDash t \\
\alpha \vDash (s \vee t) &\longleftrightarrow \alpha \vDash s \vee \alpha \vDash t \\
\alpha \vDash \neg s &\longleftrightarrow \neg(\alpha \vDash s)
\end{aligned}$$

22.3 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:

$$\begin{aligned}
S, T &: \mathsf{SFor} ::= s^+ \mid s^- && \textbf{signed formula} \\
C, D &: \mathsf{Cla} := \mathcal{L}(\mathsf{SFor}) && \textbf{clause}
\end{aligned}$$

Clauses represent conjunctions. We define evaluation of signed formulas and clauses as follows:

$$\begin{array}{lll} \mathcal{E}\alpha(s^+) & := & \mathcal{E}\alpha s \\ \mathcal{E}\alpha(s^-) & := & !\mathcal{E}\alpha s \end{array} \qquad \begin{array}{lll} \mathcal{E}\alpha\,[] & := & \mathbf{T} \\ \mathcal{E}\alpha(S :: C) & := & \mathcal{E}\alpha S \;\&\; \mathcal{E}\alpha C \end{array}$$

Note that the empty clause represents truth. We also consider lists of clauses

$$\Delta \;:\; \mathcal{L}(\mathsf{Cla})$$

and interpret them disjunctively:

$$\begin{array}{rcl} \mathcal{E}\alpha\,[\,] & := & \mathbf{F} \\ \mathcal{E}\alpha\,(C :: \Delta) & := & \mathcal{E}\alpha C \mid \mathcal{E}\alpha\Delta \end{array}$$

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 $\mathsf{sat}\ C$. 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 $\mathbf{T}$ and the negative variables in C to $\mathbf{F}$.

Fact 22.3.1 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 22.3.2 Every nonempty clausal DNF is satisfiable.

Exercise 22.3.3 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 22.3.4 Write a function that maps lists of clauses to equivalent formulas.

Exercise 22.3.5 Our formal proof of Fact 22.3.1 is unexpectedly tedious in that it requires two inductive lemmas:

1. $\alpha \vDash C \;\longleftrightarrow\; \forall S \in C.\ \alpha \vDash S$.
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 22.3.1. Can you do better?

$$
\begin{array}{rcl}
\mathsf{dnf}\ C\ [] & := & [C] \\
\mathsf{dnf}\ C\ (x^+ :: D) & := & \text{IF } \ulcorner x^- \in C \urcorner \text{ THEN } [] \text{ ELSE } \mathsf{dnf}\ (x^+ :: C)\ D \\
\mathsf{dnf}\ C\ (x^- :: D) & := & \text{IF } \ulcorner x^+ \in C \urcorner \text{ THEN } [] \text{ ELSE } \mathsf{dnf}\ (x^- :: C)\ D \\
\mathsf{dnf}\ C\ (\bot^+ :: D) & := & [] \\
\mathsf{dnf}\ C\ (\bot^- :: D) & := & \mathsf{dnf}\ C\ D \\
\mathsf{dnf}\ C\ ((s \to t)^+ :: D) & := & \mathsf{dnf}\ C\ (s^- :: D) +\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \to t)^- :: D) & := & \mathsf{dnf}\ C\ (s^+ :: t^- :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^+ :: D) & := & \mathsf{dnf}\ C\ (s^+ :: t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^- :: D) & := & \mathsf{dnf}\ C\ (s^- :: D) +\!\!+ \mathsf{dnf}\ C\ (t^- :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^+ :: D) & := & \mathsf{dnf}\ C\ (s^+ :: D) +\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^- :: D) & := & \mathsf{dnf}\ C\ (s^- :: t^- :: D)
\end{array}
$$

Figure 22.2: Definition of the DNF function $\mathsf{dnf} : \mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$

22.4 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**:

$$\mathcal{E}\alpha(\mathsf{dnf}\ C\ D) = \mathcal{E}\alpha C\ \&\ \mathcal{E}\alpha D \tag{22.1}$$

$$\mathsf{solved}\ C\ \to\ E \in \mathsf{dnf}\ C\ D\ \to\ \mathsf{solved}\ E \tag{22.2}$$

Thus 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 22.2. Note that the defining equations are clear from the two correctness properties, the boolean identities given in Section 22.1, and the idea that the first formula on the agenda controls the recursion.

Theorem 22.4.1 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 22.4.2 C is satisfiable if and only if dnf [] C is nonempty.

Corollary 22.4.3 Satisfiability of clauses and formulas is decidable.

Corollary 22.4.4 There is a solver $\forall C.\ (\Sigma\alpha.\ \alpha \vDash C) + \neg\mathsf{sat}\ C$.

Corollary 22.4.5 There is a solver $\forall s.\ (\Sigma\alpha.\ \alpha \vDash s) + \neg\mathsf{sat}\ s$.

Exercise 22.4.6 Convince yourself that the predicate $S \in C$ is decidable.

Exercise 22.4.7 Write a size function for clauses such that every recursion step of the DNF function decreases the size of the agenda.

Exercise 22.4.8 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 $\mathcal{D}$.

22.5 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 22.5.1

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 22.4.3). ∎

Exercise 22.5.2 Declare a function $\forall s.\ \mathsf{valid}\ s + (\Sigma\alpha.\ \mathcal{E}\alpha s = \mathbf{F})$ 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 22.3: Definition of $\mathsf{tab} : \mathsf{Cla} \to \mathbb{P}$

22.6 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 22.3 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 22.6.1 (Soundness) Clauses derivable with tab are unsatisfiable.

Proof $\mathsf{tab}\,C \to \alpha \vDash C \to \bot$ follows by induction on tab. ∎

Fact 22.6.2 (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 22.6.3 (Move Rules) The following rules hold for tab:

$$\frac{\mathsf{tab}\,(\mathsf{rev}\,D \mathbin{+\!\!+} C \mathbin{+\!\!+} E)}{\mathsf{tab}\,(C \mathbin{+\!\!+} D \mathbin{+\!\!+} E)} \qquad \frac{\mathsf{tab}\,(D \mathbin{+\!\!+} C \mathbin{+\!\!+} E)}{\mathsf{tab}\,(C \mathbin{+\!\!+} D \mathbin{+\!\!+} E)} \qquad \frac{\mathsf{tab}\,(C \mathbin{+\!\!+} S :: D)}{\mathsf{tab}\,(S :: C \mathbin{+\!\!+} D)}$$

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 22.6.4 $\mathsf{dnf}\ C\ D = [\,] \to \mathsf{tab}\,(D \mathbin{+\!\!+} C)$.

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 22.6.4 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 22.6.5 The clauses derivable with tab are exactly the unsatisfiable clauses.

Proof Follows with Fact 22.6.1, Corollary 22.4.2, and Lemma 22.6.4. ∎

Corollary 22.6.6 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 22.6.7 Prove $\mathsf{tab}\,(C \mathbin{+\!\!+} S :: D \mathbin{+\!\!+} T :: E) \longleftrightarrow \mathsf{tab}\,(C \mathbin{+\!\!+} T :: D \mathbin{+\!\!+} S :: E)$.

Exercise 22.6.8 Give an inductive predicate that derives exactly the satisfiable clauses. Start with an inductive predicate deriving exactly the solved clauses.

22.7 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 22.7.1 $\mathcal{E}\alpha\hat{C} = \mathcal{E}\alpha C$, $\mathcal{E}\alpha A^+ = \mathcal{E}\alpha A$, and $\widehat{A^+} = A$.

Fact 22.7.2 (Decidability) Satisfiability of unsigned clauses is decidable.

Proof Follows with Corollary 22.4.3 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 22.4. 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 22.7.3 Let ρ be a refutation predicate. Then $\mathsf{tab}\, C \to \rho\hat{C}$.

Proof Straightforward by induction on $\mathsf{tab}\, C$. ∎

Fact 22.7.4 (Completeness)
Every refutation predicate holds for all unsatisfiable unsigned clauses.

Proof Follows with Theorem 22.6.5 and Lemma 22.7.3. ∎

$$\frac{\rho\,(s :: A + B)}{\rho\,(A + s :: B)} \qquad \frac{}{\rho\,(x :: \neg x :: A)} \qquad \frac{}{\rho\,(\bot :: A)}$$

$$\frac{\rho\,(\neg s :: A) \qquad \rho\,(t :: A)}{\rho\,((s \to t) :: A)} \qquad \frac{\rho\,(s :: \neg t :: A)}{\rho\,(\neg(s \to t) :: A)}$$

$$\frac{\rho\,(s :: t :: A)}{\rho\,((s \wedge t) :: A)} \qquad \frac{\rho\,(\neg s :: A) \qquad \rho\,(\neg t :: A)}{\rho\,(\neg(s \wedge t) :: A)}$$

$$\frac{\rho\,(s :: A) \qquad \rho\,(t :: A)}{\rho\,((s \vee t) :: A)} \qquad \frac{\rho\,(\neg s :: \neg t :: A)}{\rho\,(\neg(s \vee t) :: A)}$$

Figure 22.4: 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 22.7.5 Every sound refutation predicate is decidable and holds exactly for unsatisfiable unsigned clauses.

Proof Facts 22.7.4 and 22.7.2. ∎

Theorem 22.7.6 The minimal refutation predicate inductively defined with the rules for refutation predicates derives exactly the unsatisfiable unsigned clauses.

Proof Follows with Fact 22.7.4 and a soundness lemma similar to Fact 22.6.1. ∎

Exercise 22.7.7 (Certifying Solver) Declare a function $\forall A.\ (\Sigma\alpha.\ \alpha \vDash A) + \neg\mathsf{sat}\,A$.

Exercise 22.7.8 Show that boolean entailment

$$A \dot{\vDash} s \ :=\ \forall \alpha.\ \alpha \vDash A \to \alpha \vDash s$$

is decidable.

Exercise 22.7.9 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.

23 Well-Founded Recursion

We will define a recursion operator generalizing size recursion. The operator obtains a function

$$\forall x.\, px$$

from a step function

$$\forall x.\, (\forall y.\, Ryx \to py) \to px$$

The step function provides for recursion that is guarded by a well-founded relation R. The prototype of a well-founded relation is the order relation $x < y$ on numbers. Well-founded relations are defined constructively with a transfer predicate providing higher-order recursion as it is needed for the definition of the well-founded recursion operator.

We will prove an unfolding equation for the well-founded recursion operator. Using the unfolding equation, we can show that the function constructed by the well-founded recursion operator satisfies the equational specification underlying the step function. No complementary specification is needed for this purpose. Thus we obtain a method that constructs satisfying functions for equational specifications whose recursion can be guarded by well-founded relations.

The topics presented in this chapter belong to the core of constructive type theory. Our presentation consists of a theoretical thread and a practical development for a gcd function.

23.1 Well-Founded Recursion Operator

We assume a binary relation $R : X \to X \to \mathbb{P}$ and say that a point y is **below** a point x if Ryx. We define the **accessible points of R** inductively: A point x is accessible if every point below x is accessible. Formally, we capture the accessible points of R with a transfer predicate called **accessibility**:

$$A_R(x : X) : \mathbb{P} \ ::= \ C_R(\forall y.\, Ryx \to A_R y)$$

We say that a relation is **well-founded** if all points are accessible:

$$\mathsf{wf}\, R \ := \ \forall x.\, A_R(x)$$

The accessibility predicate A_R provides for higher-order recursion: A derivation $C_R(\alpha) : A_R(x)$ carries a function $\alpha : \forall y.\, Ryx \to A_R(y)$ that for all y below x yields a structurally smaller derivation. This is exactly the recursion we need for the well-founded recursion operator.

We assume a **return type function** $p : X \to \mathbb{T}$ and a **step function**

$$F : \forall x.\, (\forall y.\, Ryx \to py) \to px$$

and define the worker function for the well-founded recursion operator as follows:

$$\begin{aligned} &W' : \forall x.\, A_R(x) \to px \\ &W'x(C\alpha) \;:=\; Fx(\lambda yr.\, W'y(\alpha yr)) \end{aligned}$$

The definition of the **well-founded recursion operator**

$$W : \forall X^{\mathbb{T}}\, \forall R^{X\to X\to\mathbb{P}}\, \forall p^{X\to\mathbb{T}}.\, \mathsf{wf}\, R \to (\forall x.\, (\forall y.\, Ryx \to py) \to px) \to \forall x.\, px$$

is now routine:

$$WXRpHFx \;:=\; W'XRpFx(Hx)$$

We will speak of **well-founded induction** when we use W for proofs.

We see W' as the canonical eliminator for the accessibility predicate A. The full type of W' explains this view:

$$W' : \forall X^{\mathbb{T}}\, \forall R^{X\to X\to\mathbb{P}}\, \forall p^{X\to\mathbb{T}}.\, (\forall x.\, (\forall y.\, Ryx \to py) \to px) \to \forall x.\, A_R(x) \to px$$

When we use W' for proofs, we speak of an **accessibility induction** or of an **induction on** $A_R(x)$.

23.2 Unfolding Equation

We show an unfolding equation for the well-founded recursion operator W saying that the function constructed satisfies the equational specification underlying the guard function. The unfolding equation is shown for step functions that are **extensional** in their proof argument:

$$\forall x\varphi\varphi'.\;\; (\forall zr.\, \varphi zr = \varphi' zr) \to Fx\varphi = Fx\varphi'$$

Lemma 23.2.1 Let F be an extensional step function for a relation R.
Then $\forall aa'.\, W'Fxa = W'Fxa'$.

Proof It suffices to prove $Ax \to \forall aa'.\, W'Fxa = W'Fxa'$. We proof this claim by induction on $A_R(x)$. We destructure a and a'. Using the defining equation for W' and the extensionality of F, we need to show $W'Fy(\varphi yr) = W'Fy(\varphi' yr)$ for $r : Ryx$, which follows with the inductive hypothesis. ∎

Fact 23.2.2 (Unfolding Equation)
Let F be an extensional step function for a well-founded relation R.
Then $WFx = Fx(\lambda y\_.\, WFy)$.

Proof Unfold W on the left, destruct the accessibility derivation, and apply the defining equation of W'. Then use the extensionality of F to reduce to $W'Fy(\varphi yr) = WFy$. Now unfold W on the right and conclude with Lemma 23.2.1. ∎

We will see that we can use the unfolding equation to directly construct solutions for recursive specifications where the recursion respects a well-founded relation. The first step is to formulate the recursive specification and the *guarded proofs* for the recursive calls (i.e., the proofs for Ryx) as a step function F. The well-founded recursion operator and the unfolding equation give us a function $f := WF$ such that $fx = Fx(\lambda y\_.\, fy)$. Since the *continuation function* $\lambda y\_.\, fy$ throws away the guardedness proofs, the equation $fx = Fx(\lambda y\_.\, fy)$ gives us that f satisfies the recursive specification.

We say that a function $f : \forall x.px$ **satisfies** a step function F if the equation $fx = Fx(\lambda y\_.\, fy)$ holds for all x. Fact 23.2.2 tells us that WF satisfies F (assuming a well-founded relation R). It turns out that WF is the only function satisfying F (up to functional extensionality).

Theorem 23.2.3 Let F be an extensional step function for a well-founded relation R. Then WF satisfies F and agrees with every function satisfying F.

Proof Fact 23.2.2 says that WF satisfies F. For the agreement we assume that f satisfies F and show $\forall x.\, fx = WFx$. We do this by well-founded induction on x and R. By using the assumption on the left and the unfolding equation (Fact 23.2.2) on the right, the claim reduces to $Fx(\lambda y\_.fy) = Fx(\lambda y\_.WFy)$. With the extensionality of F this reduces to $fy = WFy$ for Ryx, which follows by the inductive hypothesis. ∎

For practical examples, showing extensionality of step functions is routine. We remark that all step functions are trivially extensional once we assume that accessibility propositions $A_R(x)$ are pure. Interestingly, it turns out that all accessibility propositions are pure once we assume functional extensionality.

Fact 23.2.4 Assuming general functional extensionality, every accessibility proposition $A_R(x)$ is pure.

Proof We assume general functional extensionality and prove

$$\forall x.\, A_R(x) \to \forall a^{A_R(x)}\, \forall a'^{A_R(x)}.\, a = a'$$

by induction on $A_R(x)$. After destructuring the derivations a and a' it suffices to prove $\alpha = \alpha'$ for two functions $\forall y.\, Ryx \to A_R(y)$. Using functional extensionality, we assume $r : Ryx$ and show $\alpha yr = \alpha' yr$ at type $A_R(y)$. The equation follows with the inductive hypothesis. ∎

Exercise 23.2.5 The proofs of Lemma 23.2.1 and Fact 23.2.4 use a particular setup for accessibility induction which may be justified with an interface lemma

$$(\forall x.\, A_R(x) \to \forall a.\, pxa) \to \forall x\, \forall a^{A_R(x)}.\, pxa$$

Understand the need for a justification and prove the lemma.

23.3 Well-Founded Relations

Fact 23.3.1 (Accessibility function for numbers)
There is a function $\forall n^{\mathsf{N}}.\, A_< n$.

Proof It suffices to construct a function

$$R : \forall nx.\, x < n \to A_< x$$

We define R using structural recursion on n:

$$\begin{aligned} R0xh &:= \text{MATCH } \ulcorner\bot\urcorner\ [] && h : x < 0 \\ R(\mathsf{S}n)xh &:= C_{<x}(\lambda yh'.\, Rny\ulcorner y < n\urcorner) && h : x < \mathsf{S}n,\ \ h' : y < x \end{aligned}$$ ∎

Note that the proof of Fact 23.3.1 is very similar to the proof for Lemma 12.1.1.

Corollary 23.3.2 The order relation $<$ on numbers is well-founded.

In practice, a size function mapping a type to the domain of a well-founded relation is often useful. It turns out that preimages of well-founded relations under functions are again well-founded relations. This way size recursion can be understood as pure well-founded recursion. We will show that $\lambda xy.R(\sigma x)(\sigma y)$ is a well-founded relation on X if σ is a function $X \to Z$ and R is a well-founded relation on Z.

Lemma 23.3.3 (Transport)
Let $R : Z \to Z \to \mathbb{P}$ and $\sigma : X \to Z$. Let $S := \lambda xy.\, R(\sigma x)(\sigma y)$.
Then $A_R(\sigma x) \to A_S(x)$.

Proof It suffices to show

$$\forall z.\, A_R(z) \to \forall x.\, \sigma x = z \to A_S(x)$$

We do this by induction on $A_R(z)$. We assume $\sigma x = z$ and prove $A_S(x)$. It suffices to construct a function $\forall y.\, Syx \to A_S(y)$. We assume Syx and show $A_S(y)$. Since $R(\sigma y)z$, we have the inductive hypothesis $\forall x'.\, \sigma x' = \sigma y \to A_S(x')$. The claim $A_S(y)$ follows. ∎

Fact 23.3.4 (Preimage)
Let $\sigma : X \to Z$, and $R : Z \to Z \to \mathbb{P}$ be a well-founded relation.
Then $\lambda xy.\, R(\sigma x)(\sigma y)$ is a well-founded relation.

Proof Immediate with Lemma 23.3.3. ∎

Corollary 23.3.5 (Well-founded size recursion)
Let $\sigma : X \to Z$, and $R : Z \to Z \to \mathbb{P}$ be a well-founded relation.
Then we have a function

$$\forall p^{X \to \mathbb{T}}.\, (\forall x.\, (\forall y.\, R(\sigma y)(\sigma x) \to py) \to px) \to \forall x.\, px$$

Proof Follows with the well-founded recursion operator W and Fact 23.3.4. ∎

Exercise 23.3.6 Use the results of this chapter to define a size recursion operator for numbers as used in Chapter 12. Write out the unfolding equation we now obtain for the size recursion operator.

23.4 Gcd Example

Figure 23.1 shows a recursive specification of a function. Using the well-founded recursion operator W, we will construct a function satisfying the specification. We shall use the unfolding equation for the recursion operator to prove that the specifying equations are satisfied. We remark that the specified function computes greatest common divisors. This information will not be used for our construction and proofs.

Using the recursion operator W, we will construct a cartesian version

$$\mathsf{N} \times \mathsf{N} \to \mathsf{N}$$

of the specified function using the return type function

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

$$
\begin{aligned}
&f : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\quad f\,0\,y &&= y \\
&\quad f\,(\mathsf{S}x)\,0 &&= \mathsf{S}x \\
&\quad f\,(\mathsf{S}x)\,(\mathsf{S}y) &&= \begin{cases} f\,(x-y)\,(\mathsf{S}y) & \text{if } y \le x \\ f\,(\mathsf{S}x)\,(y-x) & \text{if } y > x \end{cases}
\end{aligned}
$$

Termination conditions:

$$
\begin{aligned}
y \le x &\;\to\; (x-y) + \mathsf{S}y < \mathsf{S}x + \mathsf{S}y \\
y > x &\;\to\; \mathsf{S}x + (y-x) < \mathsf{S}x + \mathsf{S}y
\end{aligned}
$$

Figure 23.1: Recursive specification of a gcd function

$$
\begin{aligned}
&F : \forall c^{\mathsf{N}\times\mathsf{N}}.\ (\forall c'.\ Rc'c \to \mathsf{N}) \to \mathsf{N} \\
&\quad F\,(0,y)\,\_ &&:= y \\
&\quad F\,(\mathsf{S}x,0)\,\_ &&:= \mathsf{S}x \\
&\quad F\,(\mathsf{S}x,\mathsf{S}y)\,\varphi &&:= \begin{cases} \varphi\,(x-y,\mathsf{S}y)\,\ulcorner (x-y)+\mathsf{S}y < \mathsf{S}x+\mathsf{S}y \urcorner & \text{if } y \le x \\ \varphi\,(\mathsf{S}x,y-x)\,\ulcorner \mathsf{S}x+(y-x) < \mathsf{S}x+\mathsf{S}y \urcorner & \text{if } y > x \end{cases}
\end{aligned}
$$

Figure 23.2: Step function for the specification in Figure 23.1

and the well-founded relation R on $\mathsf{N}\times\mathsf{N}$ obtained with the size function

$$\sigma c := \pi_1 c + \pi_2 c$$

from the canonical order on N. We define the necessary step function

$$F : \forall c^{\mathsf{N}\times\mathsf{N}}.\ (\forall c'.\ Rc'c \to \mathsf{N}) \to \mathsf{N}$$

as shown in Figure 23.2 following the equations in Figure 23.1. The two recursive calls are modeled with the continuation function $\varphi : \forall c'.\ Rc'c \to \mathsf{N}$, which is supplied with the guardedness proofs needed. The guardedness proofs are for the conclusions of the termination conditions appearing in Figure 23.1. The conditions obtained with the case analysis on $(y \le x) + (y > x)$ are not needed for the guardedness proofs, which are omitted in Figure 23.2 for readability.

We now have a function WF (we omit the arguments for the well-founded relation R). We define a cascaded version

$$fxy := WF(x,y)$$

With a routine proof we show that F is extensional. Fact 23.2.2 thus gives us the unfolding equation

$$f\,x\,y \;=\; F(x,y)(\lambda c\,\_.\,f(\pi_1 c)(\pi_2 c))$$

A case analysis on x and y following the specification in Figure 23.1 now confirms that f satisfies the equations of the specification.

23.5 Step Functions as Specifications

How can we represent a recursive specification of a function

$$f : \forall x.\, p\,x$$

formally in type theory? The conventional fixed-point approach represents a recursive specification with an **unguarded step function**

$$U : (\forall x.\, p\,x) \to \forall x.\, p\,x$$

and looks for a function f such that

$$\forall x.\; f\,x = U\,f\,x$$

The first argument of U acts as continuation function providing for recursive calls. We will call a function f satisfying the above equation a **solution of U**. Under functional extensionality, the solutions of U are the fixed points of U.

Since Coq admits only total functions, not every unguarded step function U has a solution in Coq. For a solution to exist it is sufficient that the recursion established by U is terminating. We can ensure termination by providing a well-founded relation R guarding the recursive calls in U. Formally, we switch to a **guarded step function**

$$F : \forall x.\; (\forall y.\, R\,y\,x \to p\,y) \to p\,x$$

taking a guarded continuation function as argument and define the solutions of F as all functions satisfying the unfolding equation

$$\forall x.\; f\,x = F\,x\,(\lambda y\,\_.\,f\,y)$$

The solutions of a guarded step function F are thus the solutions of the *associated unguarded step function*

$$U_F \;:=\; \lambda f\,x.\,F\,x\,(\lambda y\,\_.\,f\,y)$$

which throws away the guardedness proofs. Under functional extensionality, the associated unguarded step function will be equal to the unguarded step function used as starting point for the definition of the guarded step function. Thus, under

$$
\begin{aligned}
U &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
U\,0\,y &= y \\
U\,(\mathsf{S}x)\,0 &= \mathsf{S}x \\
U\,(\mathsf{S}x)\,(\mathsf{S}y) &= \begin{cases} U\,(x-y)\,(\mathsf{S}y) & \text{if } y \le x \\ U\,(\mathsf{S}x)\,(y-x) & \text{if } y > x \end{cases}
\end{aligned}
$$

Figure 23.3: Unguarded step function for the specification in Figure 23.1

functional extensionality, the solutions of the guarded step function are exactly the solutions of the unguarded step function.

We have shown that the well-founded recursion operator yields a solution for every extensional guarded step function, and that the solutions of extensional guarded step functions are unique.

We recall that the well-founded recursion operator is obtained by a straightforward application of the eliminator for the accessibility predicate. The elegance of the entire approach thus rests on the accessibility predicate, which on the one hand provides exactly the higher-order recursion needed for the well-founded recursion operator, and on the other hand captures the notion of well-foundedness in way that works very well constructively.

Figure 23.3 shows the cascaded version of the unguarded step function for the recursive specification of a gcd function in Figure 23.1.

23.6 Functional Induction

When we prove properties of a function satisfying an equational specification, we use well-founded induction to account for the recursion in the specification. It turns out that one can formulate special induction predicates that combine the equations of the specification with the well-founded induction corresponding to the recursion. Figures 23.4 shows the induction predicate for the specification of a gcd function in Figure 23.1. Note that the predicate has a clause for every equation of the specification in Figure 23.1.

Fact 23.6.1 A function satisfies the induction predicate for a gcd function if and only if it satisfies the unguarded step function for a gcd function. Formally, we have

$$\forall f.\ \mathsf{Ind}(f) \longleftrightarrow \forall xy.\ fxy = Ufxy$$

The functions Ind and U are defined in Figures 23.4 and 23.3.

$$
\begin{aligned}
&\mathsf{Ind} : (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathbb{T} \\
&:= \lambda f. \\
&\quad \forall p^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathbb{T}}. \\
&\quad (\forall y.\ p0yy) \to \\
&\quad (\forall x.\ p(\mathsf{S}x)0(\mathsf{S}x)) \to \\
&\quad (\forall xyz.\ y \le x \to z = f(x-y)(\mathsf{S}y) \to p(x-y)(\mathsf{S}y)z \to p(\mathsf{S}x)(\mathsf{S}y)z) \to \\
&\quad (\forall xyz.\ y > x \to z = f(\mathsf{S}x)(y-x) \to p(\mathsf{S}x)(y-x)z \to p(\mathsf{S}x)(\mathsf{S}y)z) \to \\
&\quad \forall xy.\ pxy(fxy)
\end{aligned}
$$

Figure 23.4: Induction predicate for a gcd function

Proof Suppose $H : \mathsf{Ind}(f)$. We show the claim with H and $pxyz := (z = Ufxy)$. This yields 4 proof obligations, one fo each equation of U. All proof obligations have straightforward proofs.

Suppose $\forall xy.\ fxy = Ufxy$. Moreover, assume $p^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathbb{T}}$ and the four conditions on p. We show $\forall xy.\ pxy(fxy)$ by size induction on $x + y$ (formally, we switch to pairs (x, y)). We use the assumption and show $pxy(Ufxy)$. We now consider the 4 cases underlying U and Ind. The base cases are immediate with the respective conditions for p. The recursive cases follow with the inductive hypothesis and the respective conditions for p. ∎

We remark that the induction predicate in Figure 23.4 can be formulated without the auxiliary variable z. We have introduced the auxiliary variable for readability and conciseness.

Induction predicates for recursively specified functions are special-purpose induction schemes facilitating proofs about the specified functions. For instance, assuming a function satisfying the induction predicate in Figure 23.4, no further induction scheme is needed to show that the function computes greatest common divisors.

Exercise 23.6.2 Prove that every function f satisfying the induction predicate in Figure 23.4 satisfies $\gamma xy(fxy)$. Take the definition of γ given in § 12.6 and use Fact 12.6.3.

Exercise 23.6.3 Give an unguarded step function and an induction predicate for a division function and show that they are equivalent. Use the induction predicate to show that the specified function is in fact a division function as specified in § 12.4. Use the well-founded recursion operator and the unfolding equation to obtain a division function satisfying the specifications.

23.7 Notes

Given an equational specification, a function satisfying the specifications can be constructed provided the recursion of the specification can be guarded with a well-founded relation. The construction comes with the proviso that the step function modeling the specification and the guard conditions is extensional. The construction is based on a transfer predicate known as accessibility predicate providing a constructive definition of well-foundedness using higher-order recursion.

The inductive definition of well-foundedness is due to Aczel [1]. Nordström [10] adapts Aczel's definition to a constructive type theory without propositions and advocates functions recursing on accessibility derivations. Balaa and Bertot [2] define a well-founded recursion operator in Coq based on the transfer predicate for accessibility. They prove the unfolding equation for the operator and suggest that Coq should support the construction of functions with an automation tool taking care of the tedious routine proofs coming with well-founded recursion. Currently, Coq supports well-founded recursion with a package called Equations.

24 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.

24.1 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

$$\forall x.\ px \longleftrightarrow \mathsf{tsat}(Fx)$$

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 $px \longleftrightarrow \exists n.\ Fxn = \mathbf{T}$ 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 24.1.1 Decidable predicates are semi-decidable.

Fact 24.1.2 tsat is semi-decidable.

We define **semi-decision types** $S(X)$ as follows:

$$\begin{aligned} &S :\ \mathbb{P} \to \mathbb{T} \\ &S(X)\ :=\ \Sigma f^{\mathsf{N}\to\mathsf{B}}.\ X \longleftrightarrow \mathsf{tsat}\, f \end{aligned}$$

Fact 24.1.3 $\forall X^{\mathbb{P}}.\ \mathcal{D}(X) \to S(X)$.

Fact 24.1.4 (Transport) $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ (X \longleftrightarrow Y) \to S(X) \to S(Y)$.

Fact 24.1.5 $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ S(X) \to S(Y) \to S(X \vee Y)$.

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 24.1.6 $\forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ S(X) \to S(Y) \to S(X \wedge Y)$.

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 $\forall x^X.\ S(px)$. 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 24.1.7 $\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ (\forall x.\ S(px)) \Leftrightarrow (\Sigma F^{X \to \mathsf{N} \to \mathsf{B}}\ \forall x.\ px \longleftrightarrow \mathsf{tsat}\,(Fx))$.

Proof Direction $\Rightarrow$. We assume $H : \forall x.\ S(px)$ and $x : X$ and show $px \longleftrightarrow \mathsf{tsat}(Fx)$ for $Fx := \pi_1(Hx)$. Straightforward.

Direction $\Leftarrow$. We assume $H : \forall x.\ px \longleftrightarrow \mathsf{tsat}\,(Fx)$ and $x : X$ and show $S(px)$. Trivial with Fx as test. ∎

Corollary 24.1.8 A predicate is semi-decidable iff it has a certifying semi-decider.

Recall the discussion of computational omniscience in Section 18.5. It turns out that from a decider for tsat we can get a function translating semi-decisions into decisions, and vice versa.

Fact 24.1.9 $(\forall X^{\mathbb{P}}.\ S(X) \to \mathcal{D}(X)) \Leftrightarrow (\forall f^{\mathsf{N} \to \mathsf{B}}.\ \mathcal{D}(\mathsf{tsat}\, f))$.

Proof Direction $\Rightarrow$ follows since f is a test for $S(\mathsf{tsat}\, f)$. For direction $\Leftarrow$ we assume $X \longleftrightarrow \mathsf{tsat}\, f$ and show $\mathcal{D}(X)$. By the primary assumption we have either $\mathsf{tsat}\, f$ or $\neg\mathsf{tsat}\, f$. Thus $\mathcal{D}(X)$. ∎

24.2 Markov-Post Equivalence

Recall the definition of Markov's principle:

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

We also need the function type

$$\mathsf{Post} := \forall X^{\mathbb{P}}.\ S(X) \to S(\neg X) \to \mathcal{D}(X)$$

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 24.2.1 Post $\to \forall X^{\mathbb{P}}.\ \mathcal{D}(X) \Leftrightarrow S(X) \times S(\neg X)$.

Proof Straightforward. ∎

Fact 24.2.2 Markov → 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 $\Sigma n.\ hn = \mathbf{T}$. Since we have a witness operator and Markov, it suffices to show $\neg\forall n.\ hn = \mathbf{F}$. We assume $H : \forall n.\ hn = \mathbf{F}$ and show $\neg X$ and $\neg\neg X$. Follows since H implies that f and g are constantly false. ∎

Fact 24.2.3 Post → Markov.

Proof We assume Post and $H : \neg\neg\mathsf{tsat}\, f$ and prove $\mathsf{tsat}\, f$ (using Exercise 24.4.1). It suffices to show $\mathcal{D}(\mathsf{tsat}\, f)$. Using Post it suffices to show $S(\mathsf{tsat}\, f)$ and $S(\neg\mathsf{tsat}\, f)$. $S(\mathsf{tsat}\, f)$ holds with f as test. $S(\neg\mathsf{tsat}\, f)$ holds with $\lambda\_.\mathbf{F}$ as test. ∎

Theorem 24.2.4 Markov ⇔ Post.

Proof Facts 24.2.2 and 24.2.3. ∎

We say that a predicate p is **co-semi-decidable** if its **complement** $\overline{p} := \lambda x.\neg px$ is semi-decidable.

Corollary 24.2.5 Given Markov, a predicate is decidable iff it is semi-decidable and co-semi-decidable.

Corollary 24.2.6 Given Markov and ¬CO, tsat is not co-semi-decidable.

Proof Suppose that tsat is co-semi-decidable. Since tsat is semi-decidable (Fact 24.1.2), tsat is decidable by Fact 24.2.2. Contradiction with ¬CO. ∎

Exercise 24.2.7 Prove $\forall X^{\mathbb{P}}.\ (X \vee \neg X) \to S(X) \to S(\neg X) \to \mathcal{D}(X)$.

Exercise 24.2.8 Prove Markov $\to \forall X.\ \mathcal{D}(X) \Leftrightarrow S(X) \times S(\neg X)$.

Exercise 24.2.9 Prove Markov $\to (\forall X.\ S(X) \to S(\neg X)) \to$ CO. Note that this implies that semi-decisions don't propagate through implications and negations if Markov and ¬CO are assumed.

24.3 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 $\forall x.\ px \longleftrightarrow q(fx)$. Formally, we define a predicate as follows:

$$\begin{aligned}&\mathsf{red}: \ \forall X^{\mathbb{T}} Y^{\mathbb{T}}.\ (X\to\mathbb{P})\to(Y\to\mathbb{P})\to(X\to Y)\to\mathbb{P}\\ &\mathsf{red}\ XYpqf \ :=\ \forall x.\ p(x)\longleftrightarrow q(fx)\end{aligned}$$

We treat the polymorphic arguments of red as implicit arguments.

Fact 24.3.1 Decidability and undecidability propagate along reductions as follows.

$$\begin{aligned}\mathsf{red}\,pqf &\to (\forall y.\,\mathcal{D}(qy))\to(\forall x.\,\mathcal{D}(px))\\ \mathsf{red}\,pqf &\to (\forall y.\,\mathcal{S}(qy))\to(\forall x.\,\mathcal{S}(px))\\ \mathsf{ex}\,(\mathsf{red}\,pq)&\to \mathsf{ex}\,(\mathsf{bdec}\,q)\to\mathsf{ex}\,(\mathsf{bdec}\,p)\\ \mathsf{ex}\,(\mathsf{red}\,pq)&\to \neg\mathsf{ex}\,(\mathsf{bdec}\,p)\to\neg\mathsf{ex}\,(\mathsf{bdec}\,q)\end{aligned}$$

Proof Straightforward. ∎

Fact 24.3.2 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 24.3.3 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 24.3.4 Prove $\mathsf{red}\,p\,q\,f\to\mathsf{red}\,\overline{q}\,\overline{p}\,f$.

Technical Summary

$$
\begin{aligned}
\mathcal{D}(X) &:= X + \neg X \\
\mathcal{S}(X) &:= \Sigma f.\ X \longleftrightarrow \mathsf{tsat}\, f \\
\mathsf{tsat}\, f &:= \exists n^{\mathsf{N}}.\ f n = \mathbf{T} \\
\mathsf{bdec}\, p g &:= \forall x^{X}.\ p x \longleftrightarrow g x = \mathbf{T} \\
\mathsf{CO} &:= \mathsf{ex}\,(\mathsf{bdec}\ \mathsf{tsat}) \\
\mathsf{Markov} &:= \forall f.\ \neg(\forall n.\ f n = \mathbf{F}) \to \mathsf{tsat}\, f \\
\mathsf{Post} &:= \forall X.\ \mathcal{S}(X) \to \mathcal{S}(\neg X) \to \mathcal{D}(X) \\
\mathsf{red}\, p q f &:= \forall x.\ p x \longleftrightarrow q(f x)
\end{aligned}
$$

$$
\begin{gathered}
\mathsf{tsat}\, f \Leftrightarrow \Sigma n.\ f n = \mathbf{T} \\
\mathsf{tsat}(\lambda n.\ f n \mid g n) \Leftrightarrow \mathsf{tsat}\, f + \mathsf{tsat}\, g \\
(\forall n^{\mathsf{N}}.\ \mathcal{D}(p n)) \to \mathsf{ex}\, p \to \mathsf{sig}\, p \\
\mathsf{sig}\,(\mathsf{bdec}\, p) \Leftrightarrow \forall x.\ \mathcal{D}(p x) \\
(\forall f.\ \mathcal{D}(\mathsf{tsat}\, f)) \to \mathsf{CO} \\
\mathcal{D}(X) \to \mathcal{S}(X) \times \mathcal{S}(\neg X) \\
\mathcal{S}(\mathsf{tsat}\, f) \\
(\forall f.\ \mathcal{D}(\mathsf{tsat}\, f)) \Leftrightarrow \forall X.\ \mathcal{S}(X) \to \mathcal{D}(X) \\
\mathsf{Markov} \Leftrightarrow \mathsf{Post} \\
\mathsf{Markov} \to (\mathcal{D}(X) \Leftrightarrow \mathcal{S}(X) \times \mathcal{S}(\neg X)) \\
\mathsf{Markov} \to (\forall f.\ \mathcal{S}(\neg \mathsf{tsat}\, f)) \to \mathsf{CO} \\
(\forall x.\ \mathcal{S}(p x)) \Leftrightarrow \mathsf{sig}\,(\mathsf{red}\, p\, \mathsf{tsat})
\end{gathered}
$$

- Decisions propagate through $\to$, $\neg$, $\wedge$, $\vee$.
- Semi-decisions propagate through $\wedge$, $\vee$. (conjunction needs pairing function)
- Decisions and semi-decisions travel through $\longleftrightarrow$.
- Deciders and semi-deciders travel from target to source of reductions.

24.4 Heap

$$
\begin{aligned}
\mathsf{tsat}\, f &:= \exists n^{\mathsf{N}}.\ f n = \mathbf{T} \\
\mathsf{bdec}\, p g &:= \forall x^{X}.\ p x \longleftrightarrow g x = \mathbf{T} \\
\mathsf{CO} &:= \mathsf{ex}\,(\mathsf{bdec}\ \mathsf{tsat}) \\
(\forall f.\ \mathcal{D}(\mathsf{tsat}\, f)) &\to \mathsf{CO}
\end{aligned}
$$

Exercise 24.4.1 Prove the following equivalences:

$$\begin{aligned} \mathsf{CO} &\longleftrightarrow \exists f.\ \mathsf{bdec}\,\mathsf{tsat}\,f \\ \mathsf{LPO} &\longleftrightarrow \forall f.\ \mathsf{tsat}\,f \vee \neg\mathsf{tsat}\,f \\ \mathsf{Markov} &\longleftrightarrow \forall f.\ \neg\neg\mathsf{tsat}\,f \to \mathsf{tsat}\,f \end{aligned}$$

Note that CO says that tsat f is decidable, that LPO says that tsat f is definite, and that Markov says that tsat f is stable (in each case for all tests f). Thus we may write the above equivalences as follows:

$$\begin{aligned} \mathsf{CO} &\longleftrightarrow \mathsf{ex}\,(\mathsf{bdec}\ \mathsf{tsat}) \\ \mathsf{LPO} &\longleftrightarrow \forall f.\ \mathsf{definite}\,(\mathsf{tsat}\,f) \\ \mathsf{Markov} &\longleftrightarrow \forall f.\ \mathsf{stable}\,(\mathsf{tsat}\,f) \end{aligned}$$

Bibliography

[1] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[2] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 1–16. Springer Berlin Heidelberg, 2000.

[3] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[4] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[5] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(1):176–210, 1935. Translation in: Collected papers of Gerhard Gentzen, ed. M. E. Szabo, North-Holland,1969.

[6] Gerhard Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39(1):405–431, 1935. Translation in: Collected papers of Gerhard Gentzen, ed. M. E. Szabo, North-Holland, 1969.

[7] Stanisław Jaśkowski. On the rules of supposition in formal logic, Studia Logica 1: 5—32, 1934. Reprinted in Polish Logic 1920-1939, edited by Storrs McCall, 1967.

[8] Edmund Landau. *Grundlagen der Analysis: With Complete German-English Vocabulary*, volume 141. American Mathematical Soc., 1965.

[9] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[10] Bengt Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619, Sep 1988.

[11] Raymond M. Smullyan and Melvin Fitting. *Set Theory and the Continuum Hypothesis*. Dover, 2010.

[12] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory*. Cambridge University Press, 2nd edition, 2000.