Modeling and Proving in Computational Type Theory Using the Coq Proof Assistant

Version of April 9, 2021

Gert Smolka
Saarland University

Contents

Introduction

This text teaches topics in computational logic every computer scientist should know when discussing correctness of software and hardware. We acquaint the reader with a foundational theory and a programming language for interactively constructing computational theories with machine-checked proofs. As common with programming languages, we teach foundations, canonical case studies, and practical programming in an interleaved fashion.

The foundational theory we are using is a computational type theory extending Martin-Löf type theory with inductive definitions and impredicative propositions. All functions definable in the theory are computable. The basic proof rules of the theory are intuitionistic, and assuming the law of excluded middle is possible. As it will become apparent through our case studies, computational type theory is a congenial foundation for computational models and correctness arguments, improving much on the set-theoretic language coming with mainstream Mathematics.

We will use the Coq proof assistant implementing the computational type theory we are using. The interactive proof assistant assists the user with the construction of theories and checks all definitions and proofs for correctness. Learning computational logic with an interactive proof assistant makes a dramatic difference to learning computational logic offline. The immediate feedback from the proof assistant provides for rapid experimentation and effectively teaches the rules of the underlying type theory. While the proof assistant enforces the rules of type theory, it provides much automation as it comes to routine verifications.

We will use mathematical notation throughout this text and confine all Coq code to Coq files accompanying the chapters. We assume a reader unfamiliar with type theory and the case studies we consider. So there is a lot of material to be explained and understood at mathematical levels above the Coq programming language. In any case, theories and proofs need informal explanations to be appreciated by humans, and informal explanations are needed to understand formalisations in Coq.

The abstraction level coming with mathematical notation gives us freedom in explaining the type theory and helps with separating principal ideas from engineering aspects coming with the Coq language. For instance, we will have equational inductive function definitions at the mathematical level and see Coq's primitives for expressing them only at the coding level. This way we get mathematically satisfying function definitions and a fine explanation of Coq's pattern matching construct.

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**:

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

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:

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

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:

$$\begin{array}{rcl} \& : \mathsf{B} \to \mathsf{B} \to \mathsf{B} & & \\ \mathbf{T} \mathbin{\&} y & := & y \\ \mathbf{F} \mathbin{\&} y & := & \mathbf{F} \end{array} \qquad\qquad \begin{array}{rcl} | : \mathsf{B} \to \mathsf{B} \to \mathsf{B} & & \\ \mathbf{T} \mid y & := & \mathbf{T} \\ \mathbf{F} \mid y & := & y \end{array}$$

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

$$x \mathbin{\&} (y \mid z) = (x \mathbin{\&} y) \mid (x \mathbin{\&} z)$$

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

$$x \mathbin{\&} y = y \mathbin{\&} x$$

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

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

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 Sx and the recursive application is on x.

We define **truncating subtraction** for numbers:

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

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

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

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

Exercise 1.2.1 Define functions as follows:

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

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

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

		$x + 0 = x$	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. eq.
1.2		$\mathsf{S}x - 0 = \mathsf{S}x$	comp. eq.
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. eq.

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]

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

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 admits only 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

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

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 (Exercise 16.3.5).

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

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

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** 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:

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

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

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

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

Exercise 1.7.1 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. eq.
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) $\mathsf{iter}\ !\ n\ \mathsf{T}$ tests whether n is even. Prove $\mathsf{iter}\ !\ (n \cdot 2)\ b = b$ and $\mathsf{iter}\ !\ (\mathsf{S}(n \cdot 2))\ b = !\,b$.

1.9 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:

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

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

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

1.10 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 $f\,x = !(fx)$ 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 Basic Computational Type Theory

This chapter introduces the key ideas of computational type theory in a nutshell. We start with inductive type definitions and inductive function definitions and continue with reduction rules and computational equality. We discuss termination, type preservation, and canonicity, three key properties of computational type theory. We continue with lambda abstractions, beta reduction, and eta equivalence. Finally, we introduce matches and recursive abstractions, the Coq-specific constructs for expressing inductive function definitions.

In this chapter computational type theory appears as a purely computational system with decidable equality. In the next chapter we will learn that the system presented here can express logical propositions and proofs. One crucial extension we will consider later relaxes type checking to operate modulo computational equality of types. The relaxed system will provide for equational and for inductive proofs.

2.1 Inductive Type Definitions

Our explanation of computational type theory starts with **inductive type definitions**. Here are the already discussed definitions for a type of numbers and for a type constructor for pairs:

$$\begin{aligned} \mathsf{N} &::= 0 \mid \mathsf{S}(\mathsf{N}) \\ \mathsf{Pair}(X : \mathbb{T},\ Y : \mathbb{T}) &::= \mathsf{pair}(X, Y) \end{aligned}$$

Each of the definitions introduces a system of typed constants called **constructors** consisting of a single **type constructor** and a list of **value constructors**:

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

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

Note that the constructors S, Pair, and pair have types that classify them as functions. From the types of 0 and S and the information that there are no other value constructors for N it is clear that the values of N are obtained as the terms 0, $\mathsf{S}0$, $\mathsf{S}(\mathsf{S}0)$, $\mathsf{S}(\mathsf{S}(\mathsf{S}0))$ and so forth. Analogously, given two types s and t, the values of the type $\mathsf{Pair}\,st$ are described by the terms $\mathsf{pair}\,stuv$ where u is a value of s and v is a value of t.

One distinguishing feature of computational type theory are **dependent function types** $\forall x^s.t$. In the type of the value constructor pair the primitive for dependent function types is used twice so that Pair can take two types s and t as arguments and then behave as a simply typed function $s \to t \to \mathsf{Pair}\,st$.

Computational type theory sees a **simple function type** $s \to t$ as a dependent function type $\forall x^s.t$ where the return type t does not depend on the argument x of type s. In other words, $s \to t$ is notation for $\forall x^s.t$ where the variable x does not occur in t. For instance, $\mathsf{N} \to \mathsf{N}$ is notation for $\forall x^{\mathsf{N}}.\,\mathsf{N}$.

As usual in Mathematics, the names for **bound variables** do not matter. For instance, $\forall X.\,X \to X$ and $\forall Y.\,Y \to Y$ are understood as equal types.

There is also the primitive type $\mathbb{T}$, which may be understood as the type of all types. For now, it is fine to assume $\mathbb{T} : \mathbb{T}$, that is, that the type of $\mathbb{T}$ is $\mathbb{T}$.

A key feature of computational type theory is the fact that types and functions are values like all other values. So it is perfectly fine that functions like Pair and pair take types as arguments.

Type theory only admits **well-typed terms**, that is, terms respecting the types of their constituents. Examples for ill-typed terms are $\mathsf{S}\,\mathbb{T}$ and $\mathsf{pair}\,0\,\mathsf{N}$. In the basic type theory we are considering now, every well-typed term has a unique type.

Exercise 2.1.1 Convince yourself that the following terms are all well-typed. In each case give the type of the term.

$$\mathsf{S}, \quad \mathsf{Pair}\,\mathsf{N}, \quad \mathsf{Pair}\,(\mathsf{Pair}\,\mathsf{N}\,(\mathsf{N} \to \mathsf{N})), \quad \mathsf{Pair}\,\mathsf{N}\,\mathbb{T}, \quad \mathsf{pair}\,(\mathsf{N} \to \mathsf{N})\,\mathbb{T}\,\mathsf{S}\,\mathsf{N}$$

2.2 Inductive Function Definitions

Inductive function definitions define functions by case analysis on one or several inductive arguments called **discriminating arguments**. We shall look at the examples appearing in Figure 2.2. Each of the three definitions introduces the name of the defined function (a constant) with a functional type. Then **defining equations** are given that realize the case analysis on the discriminating arguments. Note that add and swap have exactly one discriminating argument, while sub has two discriminating arguments. Since there is only one value constructor for pairs, there is only one defining equation for swap. The function add realizes a structural recursion on its discriminating argument. The structural recursion of sub can be attributed to either of its two discriminating arguments.

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

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

$$\begin{aligned}
&\mathsf{swap} : \forall X^{\mathbb{T}}.\forall Y^{\mathbb{T}}.\ \mathsf{Pair}XY \to \mathsf{Pair}YX \\
&\qquad \mathsf{swap}\,XY(\mathsf{pair}\_\_xy) \;:=\; \mathsf{pair}YXyx
\end{aligned}$$

Figure 2.1: Inductive function definitions

The left hand sides of the defining equations are called **patterns**. An important requirement for patterns is **linearity**, that is, none of the variables bound by a pattern must occur more that once in the pattern. For this reason the first two arguments of the constructor pair in the pattern for swap are written as underlines. The defining equations for a function must be **exhaustive**. That is, there must be a defining equation for every value constructor of the type of the first discriminating argument. If there are further discriminating arguments, as in the case of sub, the respective defining equations again must be exhaustive.

Every defining equation must be well-typed. Using the type declared for the function, every variable bound by the pattern of a defining equation receives a unique type. Give the types for the bound variables, type checking of the right-hand side of a defining equation is as usual.

As long as there is exactly one discriminating argument, the patterns of the defining equations are uniquely determined by the value constructors of the type of the discriminating argument.

The defining equations of an inductive function definition serve as **reduction rules** that rewrite applications of the defined function. For instance, the application $\mathsf{sub}\,(\mathsf{S}s)\,(\mathsf{S}t)$ can be **reduced** to $\mathsf{sub}\,s\,t$ using the third defining equation of sub. Things are arranged such that at most one defining equation applies to an application, and such that every application where all discriminating arguments start with a constructor can be reduced. Thus a **closed term** (no free variables) can be reduced as long as it contains an application of a defined function.

Things are arranged such that reduction **always terminates**. Without a restriction on recursion defining functions whose reduction rules do not terminate would be easy. The structural recursion requirement is a restrictive but easy to check condition ensuring termination.

Since reduction always terminates, we can compute a **normal form** for every term. There is no restriction on the application of reduction rules: reduction rules can be applied to any subterm of a term and in any order. Since the reduction rules obtained from the defining equations do not overlap, terms nevertheless have unique normal forms. We say that a term **evaluates** to its normal form.

A closed term to which no reduction rule applies is called a **canonical term**. Given that the defining equations for a function must be exhaustive (i.e., cover all constructors of discriminating arguments), every canonical term of an inductive type starts with a constructor of the type. This ensures that the canonical terms of an inductive type are exactly the terms that one can build with the constructors of the type.

We have now discussed **three key properties** of computational type theory:

- **Termination** Reduction always terminates.
- **Type preservation** Reduction preserves types: If a term of type t is reduced, one obtains a term that again has type t.
- **Canonicity** An irreducible closed term of an inductive type starts with a constructor of the type.

Canonicity gives an integrity guarantee for inductive types saying that the elements of an inductive type do not change when functions returning values of the type are added with inductive definitions. Taken together, the key properties tell us that every closed term of a given inductive type evaluates to a term starting with a constructor of the type.

Exercise 2.2.1 Give all reduction chains that reduce the term

$$\mathsf{sub}\,(\mathsf{S}0)\,(\mathsf{add}\,(\mathsf{S}(\mathsf{S}0))\,0)$$

to its normal form. Note that there are chains of different length. Here is an example for a unique reduction chain to normal form: $\mathsf{sub}\,(\mathsf{S}0)\,(\mathsf{S}y) \succ_\delta \mathsf{sub}\,0\,y \succ_\delta 0$. We use the notation $s \succ_\delta t$ for a single reduction step rewriting with a defining equation.

2.3 Plain Definitions

There are also **plain function definitions** with a single defining equation

$$f x_1 \dots x_n \;:=\; t$$

where the pattern does not contain a constructor. Thus there is no discriminating argument. In addition, there are **plain constant definitions** of the form

$$c : t \;:=\; s$$

which introduce a constant c of type t reducing to the term s without taking arguments.

Plain definitions must not be recursive. This ensures that the key properties of computational type theories are preserved.

2.4 Lambda Abstractions

A key ingredient of computational type theory are **lambda abstractions**

$$\lambda x^t.s$$

describing functions with a single argument. Lambda abstractions come with an **argument variable** x and an **argument type** t. The argument variable x may be used in the **body** s. A lambda abstraction does not given a name to the function it describes. A nice example is the nested lambda abstraction

$$\lambda X^{\mathbb{T}}.\lambda x^X.x$$

having the type $\forall X^{\mathbb{T}}.X \to X$. The abstraction describes a polymorphic identity function. The reduction rule for lambda abstractions

$$(\lambda x^t.s)\,u \succ_\beta s^x_u$$

is called **β-reduction** and replaces an application $(\lambda x^t.s)\,u$ with the term s^x_u obtained from the body s by replacing every free occurence of the argument variable x with the term u. Applications of the form $(\lambda x^t.s)\,u$ are called **β-redexes**. Here is an example for two β-reductions:

$$(\lambda X^{\mathbb{T}}.\lambda x^X.x)\,\mathsf{N}\,7 \succ_\beta (\lambda x^{\mathsf{N}}.x)\,7 \succ_\beta 7$$

As with dependent function types, the particular name of an argument variable does not matter. For instance, $\lambda X^{\mathbb{T}}.\lambda x^X.x$ and $\lambda Y^{\mathbb{T}}.\lambda y^Y.y$ are understood as equal terms.

For notational convenience, we usually omit the type of the argument variable of a lambda abstraction if it is not relevant or if it is determined by the context. We also omit parentheses and lambdas relying on two basic rules:

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

To specify the type of an argument variable, we use either the notation x^t or the notation $x : t$, depending on what we think is more readable.

Adding lambda abstractions and β-reduction to a computational type theory preserves its key properties: termination, type preservation, and canonicity.

Exercise 2.4.1 Type checking is crucial for termination of β-reduction. Convince yourself that β-reduction of an ill-typed term $(\lambda x.xx)(\lambda x.xx)$ does not terminate, and that no typing of the argument variables makes the term well-typed.

2.5 Typing Rules

Type checking is an algorithm that determines whether a term or a defining equation is well-typed. In case a term is well-typed, the type of the term is determined. In case a defining equation is well-typed, the types of the variables bound by the pattern are determined. We will not say much about type checking but rather rely on the reader's intuition and the implementation of type checking in the proof assistant. In case of doubt one can always ask the proof assistant.

Type checking is based on typing rules. The typing rules for applications and lambda abstractions may be written as

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

and may be read as follows:

- An application st has type v^x_t if s has type $\forall x^u.v$ and t has type u.
- An abstraction $\lambda x^u.s$ has type $\forall x^u.v$ if u has type $\mathbb{T}$ and s has type v under the assumption that the argument variable has type u.

The rule for applications makes precise how dependent function types are instantiated with the argument terms of applications.

Recall that simple function types $u \to v$ are dependent function types $\forall \_^u.v$ where the result type v does not depend on the argument variable. If the specialize the typing rules to simple function types,

$$\frac{\vdash s : u \to v \qquad \vdash t : u}{\vdash st \ : \ v} \qquad\qquad \frac{\vdash u : \mathbb{T} \qquad x : u \vdash s : v}{\vdash \lambda x^u.s \ : \ u \to v}$$

we obtain something that will look familiar to programmers.

2.6 Let Expressions

We will also use **let expressions**

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

providing for localized plain definitions. The reduction rule for let expressions

$$\text{LET } x^t = s \text{ IN } u \ \succ \ u^x_s$$

is called **ζ-rule** (zeta rule).

2.7 Matches

Matches are expressions realizing the structural case analysis coming with inductive types. Matches for numbers take the form

$$\textsc{match}\ s\ [\,0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v\,]$$

and come with two reduction rules:

$$\begin{aligned} \textsc{match}\ 0\ [\,0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v\,] &\succ u \\ \textsc{match}\ \mathsf{S}s\ [\,0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v\,] &\succ v^x_s \end{aligned}$$

In general, a match for an inductive type has one **clause** and one reduction rule for every constructor of the type.

Matches can be expressed as applications of defined functions, and this translation will be our preferred view on matches. For matches on numbers, we may define the function

$$\begin{aligned} \mathsf{M_N} :\ &\forall Z^{\mathbb{T}}.\ \mathsf{N} \to Z \to (\mathsf{N} \to Z) \to Z \\ \mathsf{M_N}\, Z\, 0\, a f &:= a \\ \mathsf{M_N}\, Z\, (\mathsf{S}x)\, a f &:= f x \end{aligned}$$

and replace matches with applications of this function:

$$\textsc{match}\ s\ [\,0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v\,] \quad \rightsquigarrow \quad \mathsf{M_N}\,\_\,s\,u\,(\lambda x.v)$$

We say that $\mathsf{M_N}$ is the simply typed **match function** for N. In this text, we will see matches always as applications of match functions.

Since matches are notation for application of match functions, they are type checked according to the typing rule for applications and the type of the match function used. In practice, it is convenient to compile this information into a typing rule for matches:

> A term $\textsc{match}\ s\ [\cdots]$ has type u if s is has an inductive type v, the match has a clause for every constructor of v, and every clause of the match yields a result of type u.

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

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

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

Another notational device we take from Coq writes matches with exactly one clause as pseudo-let expressions. For instance:

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

Exercise 2.7.1 (Boolean negation) Consider the inductive type definition

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

for booleans and the plain constant definition

$$! \ := \ \lambda x^{\mathsf{B}}.\ \text{MATCH } x\ [\,\mathbf{T} \Rightarrow \mathbf{F} \mid \mathbf{F} \Rightarrow \mathbf{T}\,]$$

for a boolean negation function.

a) Define a boolean match function $\mathsf{M_B}$.

b) Give a complete reduction chain for $!(!\,\mathbf{T})$. Distinguish between δ- and β-steps.

Exercise 2.7.2 (Swap function for pairs)

a) Define a function swap swapping pairs using a plain constant definition, lambda abstractions, and a match.

b) Define a matching function for the type constructor Pair.

c) Give a complete reduction chain for $\mathsf{swap}\,\mathsf{N}\,\mathsf{B}\,(\mathsf{S}0)\,\mathbf{T}$.

2.8 Recursive Abstractions

Recursive abstractions are like lambda abstractions but provide a local variable for the function described so that recursion can be expressed:

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

Using a recursive abstraction and a match, we can define a constant D reducing to a recursive function doubling the number given as argument:

$$D^{\mathsf{N}\to\mathsf{N}} \ := \ \text{FIX } f^{\mathsf{N}\to\mathsf{N}}\, x^{\mathsf{N}}.\ \text{MATCH } x\ [\,0 \Rightarrow 0 \mid \mathsf{S}x' \Rightarrow \mathsf{S}(\mathsf{S}(f x'))\,]$$

The reduction rule for recursive abstractions looks as follows:

$$(\text{FIX } f x.\, s)\, t \ \succ \ (\lambda f.\lambda x.\, s)\, (\text{FIX } f x.\, s)\, t$$

Without limitations on recursive abstractions, one can easily write recursive abstractions whose reduction does not terminate. Coq imposes two limitations:

- An application of a recursive abstraction can only be reduced if the argument term t starts with a constructor.

- A recursive abstraction is only admissible if its recursion goes through a match and is structural.

In this text we will not use recursive abstractions at all since we prefer inductive function definitions as means for describing recursive functions. Using this device, a function D doubling its argument can be defined as follows:

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

Exercise 2.8.1 Figure 2.2 gives a complete reduction chain for $D(\mathsf{S}0)$ where D is defined with a recursive abstraction as shown above. Verify every single reduction step and convince yourself that there is no other reduction chain.

2.9 Computational Equality

Computational equality is an algorithmically decidable equivalence relation on well-typed terms. Two terms are **computationally equal** if and only if their normal forms are identical up to α-equivalence and η-equivalence. The notions of α-equivalence and η-equivalence will be defined in the following.

Two terms are **α-equivalent** if they are equal up to renaming of bound variables. We have introduced several constructs involving bound variables, including dependent function types $\forall x^t.s$, patterns of defining equations and matches, lambda abstractions $\lambda x^t.s$, let expressions, and recursive abstractions. Alpha equivalence abstracts away from the particular names of bound variables but preserves the reference structure described by bound variables. For instance, $\lambda X^{\mathbb{T}}.\lambda x^X.x$ and $\lambda Y^{\mathbb{T}}.\lambda y^Y.y$ are α-equivalent abstractions having the α-equivalent types $\forall X^{\mathbb{T}}.X \to X$ and $\forall Y^{\mathbb{T}}.Y \to Y$. For all technical purposes α-equivalent terms are considered equal, so we can write the type of $\lambda X^{\mathbb{T}}.\lambda x^X.x$ as either $\forall X^{\mathbb{T}}.X \to X$ or $\forall Y^{\mathbb{T}}.Y \to Y$. We mention that alpha equivalence is ubiquitous in mathematical language. For instance, the terms $\{\, x \in \mathsf{N} \mid x^2 > 100 \cdot x \,\}$ and $\{\, n \in \mathsf{N} \mid n^2 > 100 \cdot n \,\}$ are α-equivalent and thus describe the same set.

The notion of **η-equivalence** is obtained with the **η-equivalence law**

$$\lambda x.sx \ \approx\ s \qquad \text{if } x \text{ does not occur free in } s$$

which equates a well-typed lambda abstraction $\lambda x.sx$ with the term s provided x does not occur free in t. Eta equivalence realizes the commitment to not distinguish between the function described by a term s and the lambda abstraction $\lambda x.sx$. A concrete example is the η-equivalence between the construct S for numbers and the lambda abstraction $\lambda n^{\mathsf{N}}.\mathsf{S}n$.

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

Computational equality is also known as **definitional equality**. Moreover, we say that two terms are **convertible** if they are computationally equal, and call **conversion** the process of replacing a term with a convertible term. This makes the connection with the conversion steps appearing in Chapter 1.

A complex operation the reduction rules build on is **substitution** s_t^x. Substitution 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**.

2.10 Values and Canonical Terms

We see terms as **syntactic descriptions** of **informal semantic objects** called **values**. Example for values are numbers, functions, and types. Reduction of a term preserves the value of the term, and also the type of the term. We often talk about values ignoring their syntactic representation as terms. In a proof assistant, however, values will always be represented through syntactic descriptions. The same is true for formalizations on paper, what we can formalize are syntactic descriptions, not values. We may see values as objects of our mathematical imagination.

The **values of a type** are also referred to as **elements**, **members**, or **inhabitants** of the type. We call a type **inhabited** if it has at least one inhabitant, and **uninhabited** or **empty** or **void** if it has no inhabitant. Values of functional types are referred to as **functions**.

As syntactic objects, terms may not be well-typed. Ill-typed terms are semantically meaningless and must not be used for computation and reasoning. Ill-typed terms are always rejected by a proof assistant. Working with a proof assistant 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 a well-typed term.

Recall that a term is **closed** if it has no free variables (bound variables are fine), and **canonical** if it is closed and irreducible. Computational 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 (so far we have $\mathbb{T}$).

Moreover, computational type theory is designed such that every closed term reduces to a canonical term of the same type. More generally, every term reduces to an irreducible term of the same type.

Different canonical terms may describe the same value, in particular when it comes to functions. Canonical terms that are equal up to α- and η-equivalence always describe the same value.

For simple inductive types such as N, the canonical terms of the type are in one-to-one correspondence with the values of the type. In this case we may see the values of the type as the canonical terms of the type. For function types the situation is more complicated since semantically we may want to consider two functions as equal if they agree on all arguments.

2.11 Choices Made by Coq

Coq provides neither inductive function definitions nor plain function definitions. Thus recursive functions must always be described with recursive abstractions. There is syntactic sugar facilitating the translation of function definitions (inductive or plain) into Coq's kernel language. There are plain constant definitions without arguments making it possible to preserve the constants coming with function definitions. We have already seen an example of the translation with the function D in Section 2.8.

Not having function definitions makes reduction more fine-grained and introduces additional normal forms, which turns out be a nuisance in practice. For that reasons Coq refines the basic reduction rules with *simplification rules* simulating the reductions one would have with function definitions. Sometimes the simulation is not perfect and the user is confronted with unpleasant intermediate terms.

Figure 2.2 shows a complete reduction chain for an application $D(\mathsf{S}0)$ where D is defined in Coq style with recursive abstractions. The example shows the tediousness coming with Coq's fine-grained reduction style.

In this text we will be happy to work with function definitions and to not use recursive abstractions at all. The accompanying demo files show how this high-level style can be simulated with Coq's primitives.

Having recursive abstractions and native matches is a design decision from Coq's early days (around 1990). Agda is a modern implementation of computational type theory that comes with inductive function definitions and does not offer matches and recursive abstractions.

2.12 Discussion

We have outlined a typed and terminating functional language where functions and types are first-class citizen that may be used as arguments and results. Termination is ensured by restricting recursion to structural recursion on inductive types.

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

$\hat{D}$ is the term the constant D reduces to

Figure 2.2: Reduction chain for $D(\mathsf{S}0)$ defined with a recursive abstraction

Termination buys two important properties: decidability of computational equality and integrity of inductive types (i.e., canonicity).

The generalisation of simple function types to dependent function types we have seen is a key feature of modern type theories. One speaks of *dependent type theories* to acknowledge the presence of dependent function types.

Our presentation was informal and we gave no proofs. We took some motivation from the previous chapter but it may take time until you fully understand what is said in this dense chapter. Previous familiarity with functional programming will be helpful. The next few chapters will explore the expressivity of the system and provide you with examples and case studies. For details concerning type checking and reduction, the Coq proof assistant and the accompanying demo files will prove useful.

Formalizing the system presented in this chapter and proving the claimed properties is a major project we will not attack in this text. Instead we will explore the expressivity of the system and study numerous formalizations based on the system.

In the system presented so far type checking and reduction are separated: For type checking terms and definitions we don't need reduction, and for reducing terms we don't need type checking. Soon we will boost the expressivity of the system by extending it such that type checking operates modulo computational equality of types.

Last but not least we mention that every function definable with a closed term in computational type theory is algorithmically computable. This claim rests on the fact that there is an algorithm that evaluates every closed term to a canonical term of the same type. The evaluation algorithm just performs reduction steps as long as reduction steps are possible. The order in which reduction steps are chosen matters neither for termination nor for the canonical term finally obtained.

3 Propositions as Types

A great idea coming with computational type theory is the propositions as types principle. The principle says that propositions (i.e., logical statements) can be represented as types, and that the elements of those types can serve as proofs of the propositions. This simple approach to logic works incredibly well in practice and theory: It reduces proof checking to type checking, accommodates proofs as first-call values, and provides a rich form of logical reasoning known as intuitionistic reasoning.

The propositions as types principle is just perfect for *implications* $s \to t$ and *universal quantifications* $\forall x^s.\,t$. Both kind of propositions are accommodated as function types[1] and hence receive proofs as follows:

- A proof of an implication $s \to t$ is a function mapping every proof of the premise s to a proof of the conclusion t.
- A proof of an universal quantification $\forall x^s.t$ is a function mapping every element of the type of s to a proof of the proposition t.

The types for conjunctions $s \wedge t$ and disjunctions $s \vee t$ will be obtained with inductive type constructors such that a proof of $s \wedge t$ consists of a proof of s and a proof of t, and a proof of $s \vee t$ is either a proof of s or a proof of t. The proposition falsity having no proof will be expressed as an empty type $\bot$. With falsity we can express negations $\neg s$ as implications $s \to \bot$. The types for equations $s = t$ and existential quantifications $\exists x^s.t$ will be discussed in later chapters once we have introduced the conversion rule.

In this chapter you will see many terms describing proofs with lambda abstractions and matches. The construction of such proof terms is an incremental process that can be carried out efficiently in interaction with a proof assistant. On paper we will facilitate the construction of proof terms with proof diagrams.

[1]Note the notational coincidence.

3.1 Implication and Universal Quantification

We extend our type theory with a second universe $\mathbb{P} : \mathbb{T}$ of **propositional types**. The universe $\mathbb{P}$ is inhabited with all function types $\forall x^s.t$ where t is a propositional type:

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

We also accommodate $\mathbb{P}$ as a subuniverse of $\mathbb{T}$:

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

The subuniverse rule ensures that a function type $s \to t$ expressing an implication is in fact a proposition. We use the suggestive notation $\mathbb{P} \subseteq \mathbb{T}$ to say that $\mathbb{P}$ is a subuniverse of $\mathbb{T}$.

We can now write propositions using implications and universal quantifications (i.e., function types) and proofs using lambda abstractions and applications. For instance,

$$\forall X^{\mathbb{P}}.\ X \to X$$

is a proposition that has the proof $\lambda X^{\mathbb{P}} x^X.x$. Moreover,

$$\forall X^{\mathbb{P}}.\ X$$

is a proposition that has no proof (we will give an argument for this claim later). Here is yet another proposition

$$\forall XYZ^{\mathbb{P}}.\ (X \to Y) \to (Y \to Z) \to X \to Z$$

with a proof:

$$\lambda X^{\mathbb{P}}\, Y^{\mathbb{P}}\, Z^{\mathbb{P}}\, f^{X \to Y}\, g^{Y \to Z}\, x^X.\ g(fx)$$

Here are more examples of propositions and their proofs assuming that X, Y, and Z are propositional variables (i.e., variables of type $\mathbb{P}$):

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

We have omitted the types of the argument variables appearing in the lambda abstractions on the right since they can be derived from the propositions appearing on the left.

Our final examples express mobility laws for universal quantifiers:

$$\forall X^{\mathbb{T}}\, P^{\mathbb{P}}\, p^{X\to\mathbb{P}}.\ (\forall x.\, P \to px) \to (P \to \forall x.px) \qquad \lambda XPpfax.\, fxa$$
$$\forall X^{\mathbb{T}}\, P^{\mathbb{P}}\, p^{X\to\mathbb{P}}.\ (P \to \forall x.px) \to (\forall x.\, P \to px) \qquad \lambda XPpfxa.\, fax$$

Functions that yield propositions once all arguments are supplied are called **predicates**. In the above examples p is a unary predicate on the type X. In general, a predicate has a type $u_1 \to \cdots \to u_n \to \mathbb{P}$.

Exercise 3.1.1 (Exchange law)
Give a proof for the proposition $\forall XY^{\mathbb{T}}\forall p^{X\to Y\to\mathbb{P}}.\ (\forall xy.pxy) \to (\forall yx.pxy)$.

3.2 Falsity and Negation

A propositional constant $\bot$ having no proof will be helpful since together with implication it can express negations. The official name for $\bot$ is **falsity**. The natural idea for obtaining falsity is using an inductive type definition not declaring a value constructor:

$$\bot : \mathbb{P} \ ::= \ [\,]$$

Since $\bot$ has no value constructor, the design of computational type theory ensures that $\bot$ has no element. Moreover, we can define an **elimination function**

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

using an inductive function definition with no defining equation (since $\bot$ doesn't have a value constructor). The function $\mathsf{E}_\bot$ realizes the so-called **explosion rule** also known as **ex falso quodlibet principle**: Given a hypothetical proof of falsity, we can get a proof of everything.

We now define **negation** $\neg s$ as notation for an implication $s \to \bot$:

$$\neg s \quad \rightsquigarrow \quad s \to \bot$$

With this definition we have a proof of $\bot$ if we have a proof of s and $\neg s$. Thus, given a proof of $\neg s$, we can be sure that there is no proof of s. We say that we can **disprove** a proposition s if we can give a proof of $\neg s$. The situation that we have some proposition s and hypothetical proofs of both s and $\neg s$ is called a contradiction in mathematical language. A **hypothetical proof** is a proof using unvalidated assumptions (called hypotheses in this situation).

Figure 3.1 shows proofs of propositions involving negations. To understand the proofs, it is essential to see a negation $\neg s$ as an implication $s \to \bot$. Only the proof involving the elimination function $\mathsf{E}_\bot$ makes use of the special properties of falsity.

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

The variable X ranges over propositions.

Figure 3.1: Proofs for propositions involving negations

Also 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 3.1 are normal terms.

Coming from boolean logic, you may ask for a proof of $\neg\neg X \to X$. Such a proof does not exist in general in an intuitionistic proof system like the type-theoretic system we are exploring. However, such a proof exists if we assume the law of excluded middle familiar from ordinary mathematical reasoning. We will discuss this issue later in more detail.

Occasionally, it will be useful to have a propositional constant $\top$ having exactly one proof. The official name for $\top$ is **truth**. The natural idea for obtaining truth is using an inductive type definition declaring a single primitive value constructor:

$$\top : \mathbb{P} \ ::= \ \mathsf{I}$$

We can define an elimination function for $\top$ making explicit that I is the single element of $\top$:

$$\begin{aligned} \mathsf{E}_\top &: \ \forall p^{\top \to \mathbb{T}}.\, p\,\mathsf{I} \to \forall a^\top.\, pa \\ \mathsf{E}_\top\, pb\,\mathsf{I} &:= \ b \end{aligned}$$

In later chapters we will in fact see interesting uses of $\mathsf{E}_\top$.

Exercise 3.2.1 Show that $\forall X^{\mathbb{P}}.X$ has no proof. That is, disprove $\forall X^{\mathbb{P}}.X$. That is, prove $\neg\forall X^{\mathbb{P}}.X$.

3.3 Conjunction and Disjunction

Everyone in Computer Science is familiar with the boolean interpretation of conjunctions $s \wedge t$ and disjunctions $s \vee t$. In the type-theoretic interpretation, a conjunction $s \wedge t$ is a proposition whose proofs consist of a proof of s and a proof of t, and a disjunction $s \vee t$ is a proposition whose proofs consist of either a proof of s or a proof of t. This design is made explicit with two inductive type definitions:

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

The definitions introduce the following constructors:

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

With the type constructors '$\wedge$' and '$\vee$' we can form conjunctions $s \wedge t$ and disjunctions $s \vee t$ from given propositions s and t. With the value constructors C, L, and R we can construct proofs of conjunctions and disjunctions:

- If u is a proof of s and v is a proof of t, then the term $\mathsf{C}uv$ is a proof of the conjunction $s \wedge t$.
- If u is a proof of s, then the term $\mathsf{L}u$ is a proof of the disjunction $s \vee t$.
- If v is a proof of t, then the term $\mathsf{R}v$ is a proof of the disjunction $s \vee t$.

Note that we treat the propositional arguments of the value 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 must be derived from the surrounding context. This works well in practice.

The type constructors '$\wedge$' and '$\vee$' have the type $\mathbb{P} \to \mathbb{P} \to \mathbb{P}$, which qualifies them as predicates. We will call type constructors **inductive predicates** if their type qualifies them as predicates. Moreover, we will call value constructors obtaining values of propositions **proof constructors**. Using this language, we may say that disjunctions are accommodated with an inductive predicate coming with two proof constructors.

Proofs involving conjunctions and disjunctions will often make use of matches. Recall that matches are notation for applications of match functions obtained with inductive function definitions. For conjunctions and disjunctions, we will use the definitions appearing in Figure 3.2.

Figure 3.3 shows proofs of propositions involving conjunctions and disjunctions. The propositions formulate familiar logical laws. Note that we supply as subscripts

$$\textsc{match}\ s\ [\,\mathsf{C}\,xy \Rightarrow t\,] \quad\leadsto\quad \mathsf{M}_\wedge\,\_\,\_\,\_\,s\,(\lambda xy.t)$$

$$\mathsf{M}_\wedge : \forall XYZ^{\mathbb{P}}.\ X\wedge Y \to (X\to Y\to Z)\to Z$$
$$\mathsf{M}_\wedge\, XYZ\,(\mathsf{C}\,xy)\,f \ := \ fxy$$

$$\textsc{match}\ s\ [\,\mathsf{L}\,x \Rightarrow t_1 \mid \mathsf{R}\,y \Rightarrow t_2\,] \quad\leadsto\quad \mathsf{M}_\vee\,\_\,\_\,\_\,s\,(\lambda x.t_1)\,(\lambda y.t_2)$$

$$\mathsf{M}_\vee : \forall XYZ^{\mathbb{P}}.\ X\vee Y \to (X\to Z)\to (Y\to Z)\to Z$$
$$\mathsf{M}_\vee\, XYZ\,(\mathsf{L}\,x)\,fg \ := \ fx$$
$$\mathsf{M}_\vee\, XYZ\,(\mathsf{R}\,y)\,fg \ := \ gy$$

Figure 3.2: Matches for conjunctions and disjunctions

$X \to Y \to X \wedge Y$	C_{XY}
$X \to X \vee Y$	L_{XY}
$Y \to X \vee Y$	R_{XY}
$X \wedge Y \to X$	$\lambda a.\,\textsc{match}\ a\ [\,\mathsf{C}xy \Rightarrow x\,]$
$X \wedge Y \to Y$	$\lambda a.\,\textsc{match}\ a\ [\,\mathsf{C}xy \Rightarrow y\,]$
$X \wedge Y \to Y \wedge X$	$\lambda a.\,\textsc{match}\ a\ [\,\mathsf{C}xy \Rightarrow \mathsf{C}yx\,]$
$X \vee Y \to Y \vee X$	$\lambda a.\,\textsc{match}\ a\ [\,\mathsf{L}x \Rightarrow \mathsf{R}_{YX}\,x \mid \mathsf{R}y \Rightarrow \mathsf{L}_{YX}\,y\,]$

The variables X, Y, Z range over propositions.

Figure 3.3: Proofs for propositions involving conjunctions and disjunctions

the implicit arguments of the proof constructors C, L, and R when we think it is helpful.

Figure 3.4 shows proofs involving matches with nested patterns. Matches with **nested patterns** are a notational convenience for nested plain matches. For instance, the match

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

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

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

nesting a second plain match.

$$(X \wedge Y) \wedge Z \to X \wedge (Y \wedge Z)$$
$$\lambda a.\,\text{MATCH}\ a\ [\,\mathsf{C}(\mathsf{C}xy)z \Rightarrow \mathsf{C}x(\mathsf{C}yz)\,]$$

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

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

Figure 3.4: Proofs with nested patterns

$X \wedge Y \longleftrightarrow Y \wedge X$	$X \vee Y \longleftrightarrow Y \vee X$	*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 3.5: Equivalence laws for conjunctions and disjunctions

Exercise 3.3.1 Elaborate the proofs in Figure 3.4 such that they use nested plain matches. Moreover, annote the implicite arguments of the constructors C, L and R in all non-pattern occurrences.

3.4 Propositional Equivalence

We define **propositional equivalence** $s \longleftrightarrow t$ as notation for the conjunction of two implications:

$$s \longleftrightarrow t \quad \leadsto \quad (s \to t) \wedge (t \to s)$$

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 $s \longleftrightarrow t$, we can translate every proof of s into a proof of t, and every proof of t into a proof of s. Thus we know that s is provable if and only if t is provable.

Exercise 3.4.1 Give proofs for the equivalences shown in Figure 3.5 formulating well-known properties of conjunction and disjunction.

Exercise 3.4.2 Give proofs for the following propositions:

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

b) $\neg\neg\top \longleftrightarrow \top$

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

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

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

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

Equivalence (d) is known as **de Morgan law** for disjunctions. We don't ask for a proof of the de Morgan law for conjunctions since it requires the law of excluded middle. We call proposition (f) **Russell's law**. Russell's law will be used in a couple of prominent proofs.

Exercise 3.4.3 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 Notational Issues

Following Coq, we use the precedence order

$$\neg \quad \wedge \quad \vee \quad \longleftrightarrow \quad \to$$

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

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

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

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

3.6 Proof Term Construction using 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 patterns of matches. It turns out that the proof diagrams we have introduced in Chapter 1 provide a convenient information structure for constructing proof terms.

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

		$\neg(X \longleftrightarrow \neg X)$	intro
	$f : X \to \neg X$		
	$g : \neg X \to X$	$\bot$	assert
1		X	apply g
		$\neg X$	intro
	$x : X$	$\bot$	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

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

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 *intro* 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.
- 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 performed will 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 have Coq perform the proof

$$\begin{array}{lrl}
 & \forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}}.\ \neg\neg(\forall x.\, px) \to \forall x.\, \neg\neg px & \text{intro} \\
X : \mathbb{T},\ \ p : X \to \mathbb{P} & & \\
f : \neg\neg(\forall x.\, px) & & \\
x : X,\ \ g : \neg px & \bot & \text{apply } f \\
 & \neg(\forall x.\, px) & \text{intro} \\
f' : \forall x.\, px & \bot & g(f'x)
\end{array}$$

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

Figure 3.6: Proof diagram for a double negation law for universal quantification

actions with commands called *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:

$$\begin{array}{lrl}
 & \neg\neg X \to (X \to \neg X) \to \bot & \text{intro} \\
f : \neg\neg x & & \\
g : X \to \neg X & \bot & \text{apply } f \\
 & \neg x & \text{intro} \\
x : X & \bot & gxx
\end{array}$$

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 3.6 shows a proof diagram for a double negation law for universal quantification. Since universal quantifications are function types like implications, there are no new proof actions. Figure 3.7 shows a proof diagram using a **destructuring action** contributing a match in the proof term. The reason we did not see a destructuring action before is that so far matches were inserted by the intro action. Figure 3.8 gives a proof diagram for a distributivity law involving 6 subgoals. Note the symmetry in the proof term constructed. Figure 3.9 gives a proof diagram for a double negation law for implication. Note the use of the **exfalso action** realizing the explosion rule with an application of $\mathsf{E}_\bot$.

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

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

Figure 3.7: Proof diagram using a destructuring action

		$X\wedge(Y\vee Z)\longleftrightarrow(X\wedge Y)\vee(X\wedge Z)$	apply C
1	$x:X$	$X\wedge(Y\vee Z)\to(X\wedge Y)\vee(X\wedge Z)$	intro
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		$(X\wedge Y)\vee(X\wedge Z)\to X\wedge(Y\vee Z)$	intro
2.1	$x:X,\ y:Y$	$X\wedge(Y\vee Z)$	$\mathsf{C}x(\mathsf{L}y)$
2.2	$x:X,\ z:Z$	$X\wedge(Y\vee Z)$	$\mathsf{C}x(\mathsf{R}z)$

Proof term constructed:

$$\begin{aligned}&\mathsf{C}\ (\lambda a.\ \text{MATCH}\ a\ [\,\mathsf{C}x(\mathsf{L}y)\Rightarrow\mathsf{L}(\mathsf{C}xy)\mid\mathsf{C}x(\mathsf{R}z)\Rightarrow\ \mathsf{R}(\mathsf{C}xz)])\\&\quad(\lambda a.\ \text{MATCH}\ a\ [\,\mathsf{L}(\mathsf{C}xy)\Rightarrow\mathsf{C}x(\mathsf{L}y)\mid\mathsf{R}(\mathsf{C}xz)\Rightarrow\mathsf{C}x(\mathsf{R}z)\,])\end{aligned}$$

Figure 3.8: Proof diagram for a distributivity law

Exercise 3.6.1 Give proof diagrams and proof terms 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) $\neg\neg((\neg Y\to\neg X)\to X\to Y)$

e) $\neg\neg(X\vee\neg X)$

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

g) $\neg\neg\neg X\ \longleftrightarrow\ \neg X$

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

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

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

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

Proof term constructed:

$$\begin{array}{l} \mathsf{C}\ (\lambda fgh.\, f(\lambda f'.\, g(\lambda x.\, h(f'x)))) \\ \quad (\lambda fg.\, g(\lambda x.\, \mathsf{E}_\bot Y (f\,(\lambda h.\, hx)\,(\lambda y.\, g(\lambda x.y))))) \end{array}$$

Figure 3.9: Proof diagram for a double negation law for implication

Exercise 3.6.2 Give a proof diagram and a proof term 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 3.6.3 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 3.6.4 Prove $\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}} \forall Z^{\mathbb{P}}.\ (\forall x.px) \to Z \to \forall x.\, px \wedge Z$.

3.7 Impredicative Characterizations

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 say that $\bot$, $X \wedge Y$, and $X \vee Y$ can be characterized with just function types. The equivalences are known as **impredicative characterizations** of falsity, conjunction,

$$\bot \;\longleftrightarrow\; \forall Z^{\mathbb{P}}.\, Z$$
$$\mathsf{C}\ (\mathsf{E}_\bot(\forall Z^{\mathbb{P}}.\, Z))\ (\lambda f.\, f\bot)$$

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

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

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

Figure 3.10: Impredicative characterizations with proof terms

and disjunction. Figure 3.10 gives proof terms for the equivalences. The term impredicative refers to the fact that quantification over all propositions is used.

Exercise 3.7.1 Find an impredicative characterisation for $\top$.

3.8 Law of Excluded Middle

The propositions as types approach presented here yields a rich form of logical reasoning known as *intuitionistic reasoning*. Intuitionistic reasoning refines reasoning in Mathematics in that it does not build in the law of excluded middle. This way intuitionistic reasoning makes finer differences than the so-called classical reasoning used in Mathematics. Since type-theoretic logic can quantify over propositions, the **law of excluded middle** can be directly expressed as the proposition $\forall P^{\mathbb{P}}.\, P \vee \neg P$. Once we assume excluded middle, we can prove all the propositions we can prove in boolean logic.

Exercise 3.8.1 Let XM be the proposition $\forall P^{\mathbb{P}}.\, P \vee \neg P$ formalizing the law of excluded middle. Construct proof terms for the following propositions:

a) $\mathsf{XM} \to \forall P^{\mathbb{P}}.\, \neg\neg P \to P$	double negation law
b) $\mathsf{XM} \to \forall PQ^{\mathbb{P}}.\, \neg(P \wedge Q) \to \neg P \vee \neg Q$	de Morgan law
c) $\mathsf{XM} \to \forall PQ^{\mathbb{P}}.\, (\neg Q \to \neg P) \to P \to Q$	contraposition law
d) $\mathsf{XM} \to \forall PQ^{\mathbb{P}}.\, ((P \to Q) \to P) \to P$	Peirce's law

It turns out that the reverse directions of the above implications can also be shown intuitionistically. In other words, the laws given above are all propositionally equivalent to excluded middle. The proofs are not difficult.

3.9 Discussion

In this chapter we have seen that a computational type theory with dependent function types can express propositions as types and proofs as elements of propositional types. Function types provide for implications and universal quantifications, and falsity, conjunctions and disjunctions can be added using inductive type definitions. Universal quantification as obtained with the propositions as types approach is wonderfully general in that it can quantify over all values including functions and types. Since proofs are accommodated as first-class objects, one can even quantify over proofs. In the next few chapters we will see that the type-theoretic approach to logic scales elegantly to equations and existential quantifications as well as to inductive proofs over inductive types.

The propositions as types approach uses the typing rules of the underlying type theory as proofs rules. This elegant reuse reduces proof checking to type checking and greatly simplifies the implementation of proof assistants.

Proofs are obtained as terms obtained with lambda abstractions, applications, and matches. The resulting proof language is elegant and compact. The primitives of the language generalize familiar proof patterns: making assumptions, applying implicational assumptions, and case analysis on and destructuring of assumptions.

This chapter is the place where the reader should get fluent with lambda abstractions, matches, and dependent function types. We offer dozens of examples for exploration on paper and in interaction with the proof assistant. For proving on paper we offer proof diagrams facilitating the construction of proof terms with the necessary information structure. When we construct proof terms in interaction with a proof assistant, we in fact issue proof actions stepwise building the accompanying proof diagram.

In the system presented so far, only the typing rules are needed for proofs. Since there is no use of the reduction rules in this chapter, it is fine to not know them. This will change in the next chapter when we introduce the conversion rule relaxing type checking to type checking modulo definitional equality.

The details of the typing rules matter. What prevents a proof of falsity are the typing rules and the rules admitting inductive definitions. In this text, we mostly relax about the details of the typing rules since they are fully checked by the proof assistant. To be sure that something is well-typed or has a certain type, it is always a good idea to have it checked by the proof assistant. We expect that you train your command of the typing rules using the proof assistant.

4 Conversion Rule, Universe Hierarchy, and Elimination Restriction

We now introduce an additional typing rule called conversion rule liberating the typing discipline such that typing operates modulo computational equality of types. The conversion rule is needed so that equations and existential quantifications can be accommodated with propositional types. It also provides for abstraction techniques facilitating inductive proofs.

It is also time to spell out two essential restrictions of the typing discipline. First, we withdraw the self-membership $\mathbb{T} : \mathbb{T}$ and replace it with a cumulative universe hierarchy $\mathbb{T}_1 : \mathbb{T}_2 : \mathbb{T}_3 : \cdots$ and $\mathbb{T}_1 \subseteq \mathbb{T}_2 \subseteq \mathbb{T}_3 \subseteq \cdots$. Second, we introduce the so-called elimination restriction, restricting inductive function definitions such that the result type must be propositional if the type of a discriminating argument is propositional but not computational.

4.1 Conversion Rule

Recall the typing rules for applications and lambda abstractions from § 2.5.

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

The conversion rule is an additional typing rule relaxing typing by making it operate modulo computational equality of types:

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

The rule says that a term s has type t if s has type t' and t and t' are computationally equal (we use the notation $t \approx t'$ to say that two terms t and t' are computationally equal) (recall the definition of computational equality in § 2.9). Note that the conversion rule has a premise $\vdash t : \mathbb{T}$, which ensures that the term t describes a type.

Adding the conversion rule preserves the key properties of computational type theory. As before there is an algorithm that given a term decides whether the term

$$\begin{array}{lrl}
X:\mathbb{T},\ x:X,\ y:X & & \\
f:\forall p^{X\to\mathbb{P}}.\ px\to py & \forall p^{X\to\mathbb{P}}.\ py\to px & \text{intro} \\
p:X\to\mathbb{P} & py\to px & \text{conversion} \\
 & (\lambda z.\ pz\to px)\,y & \text{apply } f \\
 & (\lambda z.\ pz\to px)\,x & \text{conversion} \\
 & px\to px & \lambda a^{px}.a
\end{array}$$

$$\text{Proof term constructed:}\quad \lambda p.\,f(\lambda z.\,pz\to px)(\lambda a.a)$$

Figure 4.1: Proof diagram for Leibniz symmetry

is well-typed and if so derives a type of the term. The derived type is unique up to computational equality of types and minimal with respect to universe subtyping (e.g., $\mathbb{P}\subseteq\mathbb{T}$).

Exploiting the presence of the conversion rule, we can accommodate negation and propositional equivalence as defined functions:

$$\begin{array}{lcl}
\neg:\ \mathbb{P}\to\mathbb{P} & \qquad & \longleftrightarrow:\ \mathbb{P}\to\mathbb{P}\to\mathbb{P} \\
\neg X\ :=\ X\to\bot & & X\longleftrightarrow Y\ :=\ (X\to Y)\wedge(Y\to X)
\end{array}$$

The function definitions give us the constants $\neg$ and $\longleftrightarrow$ for the functions constructing negations and propositional equivalences.

The conversion rule provides proofs for propositions that are not provable without it. As example we choose a proposition we call **Leibniz symmetry**:

$$\forall X^{\mathbb{T}}\ \forall xy^{X}.\ \ (\forall p^{X\to\mathbb{P}}.\ px\to py)\to(\forall p^{X\to\mathbb{P}}.\ py\to px)$$

Leibniz symmetry says that if a value y satisfies every property a value x satisfies, then conversely x satisfies every property y satisfies. Figure 4.1 shows a proof diagram for Leibniz symmetry. The diagram involves two conversion steps

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

both of which are justified by β-reduction. The proof term constructed is

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

The two conversions are not visible in the proof term, but they appear with an application of the conversion rule needed for type checking the term. To explain the use of the conversion rule, we start with the typing for the first application of the variable f:

$$\vdash f(\lambda z.\,pz\to px):\ (\lambda z.\ pz\to px)\,x\to(\lambda z.\ pz\to px)\,y$$

Using the conversion rule we can switch to the typing

$$\vdash f(\lambda z.\, pz \to px) :\; (px \to px) \to (py \to px)$$

from which we obtain the typing

$$\vdash \lambda p.\, f(\lambda z.\, pz \to px)(\lambda a^{px}.a) :\; \forall p^{X \to \mathbb{P}}.\, py \to px$$

using the typing rules for applications and lambda abstractions.

4.2 Cumulative Universe Hierarchy

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

$$\frac{}{\vdash \mathbb{P} \subseteq \mathbb{T}} \qquad \frac{\vdash s : u \qquad \vdash u \subseteq u'}{\vdash s : u'} \qquad \frac{\vdash u : \mathbb{T} \qquad \vdash v \subseteq v'}{\vdash \forall x^u.v \subseteq \forall x^u.v'}$$

Note that third rule establishes subtyping of function types so that, for instance, we obtain the inclusion $(u \to \mathbb{P}) \subseteq (u \to \mathbb{T})$ for all types u.

Types are first class objects in computational 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 the self-membership $\mathbb{T} : \mathbb{T}$ yields a proof of falsity (see § 26.3). What works, however, is an infinite cumulative hierarchy of universes

$$\begin{aligned}&\mathbb{T}_1 : \mathbb{T}_2 : \mathbb{T}_3 : \cdots\\ &\mathbb{P} \subseteq \mathbb{T}_1 \subseteq \mathbb{T}_2 \subseteq \mathbb{T}_3 \subseteq \cdots\\ &\mathbb{P} : \mathbb{T}_2\end{aligned}$$

realized with the following typing rules:

$$\frac{i < j}{\vdash \mathbb{T}_i : \mathbb{T}_j} \qquad \frac{i \geq 2}{\vdash \mathbb{P} : \mathbb{T}_i} \qquad \frac{}{\mathbb{P} \subseteq \mathbb{T}_i} \qquad \frac{i < j}{\vdash \mathbb{T}_i \subseteq \mathbb{T}_j}$$

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

$$\frac{\vdash u : \mathbb{T}_i \qquad x : u \vdash v : \mathbb{P}}{\vdash \forall x^u.v \;:\; \mathbb{P}} \qquad \frac{\vdash u : \mathbb{T}_i \qquad \vdash v : \mathbb{T}_i}{\vdash \forall x^u.v : \mathbb{T}_i}$$

The rule for $\mathbb{P}$ says that the universe of propositions is closed under all quantifications including **big quantifications** quantifying over the types of universes. In contrast, a dependent function type $\forall x^{\mathbb{T}_i}.v$ where v 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, conjunctions, and disjunctions exploit this fact.

It is common practice to not annotate 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's type checking ensures that universe levels can be assigned consistently.

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

Following Coq, we have placed $\mathbb{P}$ in $\mathbb{T}_2$. This again is one of Coq's historical design decisions, placing $\mathbb{P}$ in $\mathbb{T}_1$ is also possible and would be simpler. In this case $\mathbb{P}$ could be understood as $\mathbb{T}_0$, the lowest universe level.

4.3 Elimination Restriction

Coq's type theory imposes the restriction that inductive functions discriminating on the proofs of an inductive proposition must have propositional result types, except if the inductive proposition is *computational*. We refer to this restriction as *elimination restriction*. We first give the necessary definitions and then explain why the elimination restriction is imposed.

An inductive propositional type $c : \mathbb{P}$ or $cs_1 \dots s_n : \mathbb{P}$ is **computational** if the type constructor c has at most one value constructor, and all nonparametric arguments of the value constructor have propositional types. Examples for computational propositions we have already seen are $\bot$, $\top$, and conjunctions $s \wedge t$.

We will refer to noncomputational inductive propositions as **sealing propositions**. The example of sealing propositions we have already seen are disjunctions $s \vee t$ (two proof constructors).

The **elimination restriction** applies to inductive function definitions and requires that the result type of a defined function must be propositional if the definition comes with a discriminating argument whose type is a sealing proposition.

If we look at the inductively defined match functions for conjunctions and disjunctions in Figure 3.2, we notice that we could type Z more generally as $\mathbb{T}$ for conjunctions, and that the elimination restriction prevents us from doing so for disjunctions. Moreover, we notice that the elimination function for $\bot$

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

defined in § 3.2 types Z with $\mathbb{T}$ rather than $\mathbb{P}$, which is in accordance with the elimination restriction since $\bot$ is a computational proposition.

We now explain one reason why the elimination restriction is imposed. An important requirement for Coq's type discipline is that assuming the law of excluded middle (§ 3.8)

$$\mathsf{XM} \;:=\; \forall P^{\mathbb{P}}.\, P \vee \neg P$$

must not lead to a proof of falsity. Formally, this means that the proposition

$$\mathsf{XM} \to \bot$$

must not be provable in Coq's type theory. It now turns out that XM implies that no proposition has more than one proof semantically, a property known as **proof irrelevance**. In fact, we may express proof irrelevance as a proposition

$$\mathsf{PI} \;:=\; \forall P^{\mathbb{P}} \forall p^{P \to \mathbb{T}} \forall ab^{P}.\, pa \to pb$$

and prove

$$\mathsf{XM} \to \mathsf{PI}$$

in Coq's type theory (see § 26.5).

Recall that the disjunction $\top \vee \top$ has two different canonical proof terms, $(\mathsf{L\,I})$ and $(\mathsf{R\,I})$. Proof irrelevance now says that $(\mathsf{L\,I})$ and $(\mathsf{R\,I})$ are indistinguishable semantically. Without the elimination restriction we could write the term

$$\textsc{match}\ (\mathsf{L\,I})\ [\,\mathsf{L}\,\_ \Rightarrow \bot \mid \mathsf{R}\,\_ \Rightarrow \top\,]$$

which is computationally equal to $\bot$. Using PI, this term is inhabited if the term

$$\textsc{match}\ (\mathsf{R\,I})\ [\,\mathsf{L}\,\_ \Rightarrow \bot \mid \mathsf{R}\,\_ \Rightarrow \top\,]$$

is inhabited. The term with $(\mathsf{R\,I})$ is computationally equal to $\top$ and thus inhabited by the conversion rule. Thus we would have a proof of $\mathsf{PI} \to \bot$ if the two matches for $(\mathsf{L\,I})$ and $(\mathsf{R\,I})$ would be well-typed, which, however, is prevented by the elimination restriction. Note that the type of the matches is Prop, which is not a propositional type.

We remark that the impredicativity of the universe $\mathbb{P}$ of propositions (§ 4.2) would also yield a proof of falsity if no elimination restriction was imposed (see Chapter 26).

You have now seen a rather delicate aspect of Coq's type theory. It arises from the fact that Coq's type theory reconciles the propositions as types approach with the assumption of proof irrelevance or, even stronger, with the law of excluded middle. It turns out that there are many good reasons for assuming proof irrelevance, even if the law of excluded middle is not needed.

If you work with Coq's type theory, it is not necessary that you understand the above arguments concerning proof irrelevance and excluded middle in detail. It suffices that you know about the elimination restriction. In any case, the proof assistant will ensure that the elimination restriction is observed.

It will turn out that the presence of certain computational propositions is crucial for the definition of many important functions. One such computational proposition is $\bot$. The other essential computational propositions are recursion types involving higher-order recursion (Chapters 19 and 20), and inductive equality types providing for casts (Chapter 23).

Exercise 4.3.1 One can define a computational falsity proposition with a recursive proof constructor:

$$F : \mathbb{P} \ ::= \ C(F)$$

We can define a computational eliminator for F similar to the falsity eliminator:

$$\begin{aligned} &E : \forall Z^{\mathbb{T}}.\ F \to Z \\ &E\,Z(Ca) \ := \ E\,Z a \end{aligned}$$

Thus we don't need inductive types with zero constructors to express falsity with computational elimination.

5 Leibniz Equality

We will now define propositional equality 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.

The conversion rule is absolutely essential for an adequate definition of propositional equality. It ensures that propositional equality subsumes computational equality, and that equational rewriting can be captured with a single typed constant.

There is much elegance and surprise in this chapter. Much of the technical essence of computational type theory will be exercised in modeling propositional equality. Students may need time to understand this beautiful construction.

5.1 Propositional Equality with Three Constants

With dependent function types and the conversion rule 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:

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

The constant eq allows us to write equations as propositional types. We treat X as an implicit argument and use the notations

$$\begin{aligned} s = t &\quad\rightsquigarrow\quad \mathsf{eq}\, st \\ s \neq t &\quad\rightsquigarrow\quad \neg\mathsf{eq}\, st \end{aligned}$$

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 rule, we can also prove with Q every equation $s = t$ where s and t are computationally equal. 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 term 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) $\forall x^{\mathsf{N}}.\ 0 \neq \mathsf{S}x$

$\top \neq \bot$	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

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

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

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

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

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

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

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

Using Leibniz symmetry from Section 4.1, 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 the-

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

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

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

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

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

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

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

Figure 5.2: Proofs of basic equational facts

ory. For instance, if we assume a constant of type $\bot$, we can prove everything (ex falso quodlibet) and our carefully constructed logical system collapses.

One solid justification we can have for a constant is that we can obtain it as a constructor of an inductive type definition. Inductive type 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 constant is that we can define it with one of the definition schemes coming with computational type theory (i.e., inductive function definitions, plain function definitions, plain constant definitions). The schemes are designed such that no proof of falsity can be introduced by the defined constants.

Here are plain function definitions justifying the constants for propositional equality:

$$\begin{aligned}
&\mathsf{eq} : \forall X^{\mathbb{T}}.\ X \to X \to \mathbb{P}\\
&\mathsf{eq}\,Xxy \ := \ \forall p^{X\to\mathbb{P}}.\ px \to py\\[1em]
&\mathsf{Q} : \forall X^{\mathbb{T}} \forall x.\ \mathsf{eq}\,X\,x\,x\\
&\mathsf{Q}\,Xx \ := \ \lambda pa.a\\[1em]
&\mathsf{R} : \forall X^{\mathbb{T}} \forall xy \forall p^{X\to\mathbb{P}}.\ \mathsf{eq}\,Xxy \to px \to py\\
&\mathsf{R}\,Xxypf \ := \ fp
\end{aligned}$$

The definitions are amazingly simply. Check them by hand and with Coq. Note that the conversion rule is needed to make use of the defining equation of eq. 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.

[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 a theorem, we want to emphasize its importance.

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

Figure 5.3: Constructors and eliminators for conjunctions and disjunctions

Conjunctions and disjunctions can also be accommodated with declared constants. Figure 5.3 shows the constants needed for conjunctions and disjunctions. We distinguish between **constructors** and **eliminators**. The constructors are obtained directly with the inductive type definitions we have seen for conjunctions and disjunctions. The eliminators can be obtained with inductive function definitions. We have seen definitions of the eliminators before as match functions in Figure 3.2.

Note that the types of the eliminators $\mathsf{E}_\wedge$ and $\mathsf{E}_\vee$ are related to the impredicative characterizations of conjunction and disjunction (Section 3.7).

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

Exercise 5.3.2 Prove commutativity of conjunction and disjunction just using the constructors and eliminators.

Exercise 5.3.3 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 propositional equivalence.

Exercise 5.3.4 Prove $\forall x^\top \forall y^\top.\ x = y$ using the elimination function $\mathsf{E}_\top$ from § 3.2.

6 Universal Inductive Eliminators

For inductive types we can define functions called eliminators that through their types provide general proof rules for case analysis and induction, and that through their defining equations provide schemes for defining functions on the underlying inductive type. Eliminators are the final step in the fascinating logical bootstrap accommodating the proofs in Chapter 1 inside computational type theory.

It turns out that one eliminator per inductive type suffices. This generality becomes possible through the use of return type functions and the flexibility provided by the conversion rule. Return type functions are similar to the return type predicates used with the rewriting rule for propositional equality.

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 Eliminator

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:

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

First look at the type of $\mathsf{E_B}$. 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 type function** p freely. We have seen the use of a return type function before with the replacement constant for propositional equality.

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

	$\forall x.\, x = \mathbf{T} \vee x = \mathbf{F}$	conversion
	$\forall x.\, (\lambda x.\, x = \mathbf{T} \vee x = \mathbf{F})\, x$	apply 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

Proof term constructed: $\mathsf{E}_\mathsf{B}\ (\lambda x.\ x = \mathbf{T} \vee x = \mathbf{F})\ (\mathsf{L}(\mathsf{Q}\,\mathbf{T}))\ (\mathsf{R}(\mathsf{Q}\,\mathbf{F}))$

Figure 6.1: Proof diagram for a boolean elimination

Now look at the defining equations of E_B. They are well-typed since the patterns $\mathsf{E}_\mathsf{B}\, pab\, \mathbf{T}$ and $\mathsf{E}_\mathsf{B}\, pab\, \mathbf{F}$ on the left instantiate the return type to $p\,\mathbf{T}$ and $p\,\mathbf{F}$, which are the types of the variables a and b, respectively.

First Example: Partial Proof Terms

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**

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

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 the conversion rule. 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

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

Figure 6.1 shows a proof diagram constructing this proof term. The diagram makes explicit the conversions handling the applications of the return type functions. 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: Kaminski's Equation

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

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

[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.

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

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

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 $z = \mathbf{T}$:

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

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

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) $x \,\&\, y = \mathbf{T} \;\longleftrightarrow\; x = \mathbf{T} \land y = \mathbf{T}.$

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

6.2 Eliminator 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

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

Then we can use $\mathsf{M_N}$ to do case analysis on numbers in proofs: To prove $\forall n.\, pn$, we prove a *base case* $p0$ and a *successor case* $\forall n.\, p(\mathsf{S}n)$. Defining $\mathsf{M_N}$ with an inductive function definition is straightforward:

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

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

Recursive Eliminator for Numbers

The type of the match eliminator for numbers 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:

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

There are two questions now: Can we define a **recursive eliminator** $\mathsf{E_N}$ with the given type, and does the type of $\mathsf{E_N}$ really suffice to do proofs by structural induction? The answer to both questions is pleasantly straightforward.

To define $\mathsf{E_N}$, we take the defining equations for $\mathsf{M_N}$ and obtain the additional argument for the inductive hypothesis of the continuation function f in the successor case with structural recursion:

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

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

First Example: Commutativity of Addition

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 $\mathsf{E_N}$:

$$\mathsf{E_N}\ (\lambda x.\ x + 0 = x)\ \ulcorner 0 + 0 = 0 \urcorner\ \ulcorner \forall x.\ x + 0 = x \to \mathsf{S}x + 0 = \mathsf{S}x \urcorner$$

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 an implicational premise in the successor case. Figure 6.2 shows a proof diagram for a proof term completing the partial proof term obtained with $\mathsf{E_N}$.

		$x+0=x$	conversion
		$(\lambda x.\,x+0=x)\,x$	apply $\mathsf{E_N}$
1		$(\lambda x.\,x+0=x)\,0$	conversion
		$0=0$	comp. eq.
2		$\forall x.\,(\lambda x.\,x+0=x)\,x \to (\lambda x.\,x+0=x)(\mathsf{S}x)$	conversion
		$\forall x.\,x+0=x \to \mathsf{S}x+0=\mathsf{S}x$	intro
	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. eq.

Proof term constructed:

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

Figure 6.2: Proof diagram for $x+0=x$

Second Example: Logically Decidable Equality

Our second example

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

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

$$\begin{array}{l}\mathsf{E_N}\ (\lambda x.\ \forall y.\,x=y \lor x\neq y)\\ \ulcorner\forall y.\,0=y\lor 0\neq y\urcorner\\ \ulcorner\forall x.\,(\forall y.\,x=y\lor x\neq y)\to\forall y.\,\mathsf{S}x=y\lor\mathsf{S}x\neq y\urcorner\end{array}$$

The base case follows with case analysis on y:

$$\begin{array}{l}\mathsf{M_N}\ (\lambda y.\ 0=y\lor 0\neq y)\\ \ulcorner 0=0\lor 0\neq 0\urcorner\\ \ulcorner\forall y.\,0=\mathsf{S}y\lor 0\neq\mathsf{S}y\urcorner\end{array}$$

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

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

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

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

Figure 6.3: Proof diagram with a quantified inductive hypothesis

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

We have described the 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 $\mathsf{M_N}$ with $\mathsf{E_N}$.

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

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

Exercise 6.2.4 Prove the following propositions in Coq using $\mathsf{E_N}$ and $\mathsf{M_N}$.

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 $\mathsf{eq_N} : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$ such that $\forall xy.\ x = y \longleftrightarrow \mathsf{eq_N}\, xy = \mathbf{T}$.
Prove the equivalence.

6.3 Eliminator for Pairs

Recall the inductive type definition for pairs from § 1.6 :

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

As before we use use the notations

$$\begin{aligned} s\times t &\quad\rightsquigarrow\quad \mathsf{Pair}\,st \\ (s,t) &\quad\rightsquigarrow\quad \mathsf{pair}\,\_\,\_\,st \end{aligned}$$

Following the scheme we have seen for booleans and numbers, we can define an eliminator for pairs as follows:

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

Exercise 6.3.1 Prove the following facts for pairs $a : X\times Y$ using the eliminator $\mathsf{E}_\times$:

a) $(\pi_1 a, \pi_2 a) = a$ $\qquad$ η-law

b) $\mathsf{swap}(\mathsf{swap}\ a)$ $\qquad$ involution law

Exercise 6.3.2 Use $\mathsf{E}_\times$ to write functions that agree with π_1, π_2, and swap (see § 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 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

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

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

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

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

Fact 6.4.1 $\mathsf{N} \neq \mathsf{B}$.

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 proof assistant. When we do the proof with Coq we can see that the techniques introduced so far smoothly scale to more involved proofs.

Exercise 6.4.2 Prove $\mathsf{B} \neq \top$ and $\mathsf{B} \neq \mathsf{B} \times \mathsf{B}$.

Exercise 6.4.3 Prove $\mathsf{B} \neq \mathbb{T}$.

Exercise 6.4.4 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 $\mathsf{B} = \mathsf{B} \times \top$.

6.5 Abstract Return Types

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

$$\begin{aligned} \mathsf{E}_\perp &: \ \forall Z^{\mathbb{T}}.\ \perp \to Z \\ \mathsf{E}_\mathsf{B} &: \ \forall p^{\mathsf{B}\to\mathbb{T}}.\ p\,\mathbf{T} \to p\,\mathbf{F} \to \forall x.px \end{aligned}$$

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:

$$\begin{aligned} \mathsf{E}_\perp\,\mathsf{N} &: \ \perp \to \mathsf{N} \\ \mathsf{E}_\perp\,(\mathsf{N} \to \mathsf{N}) &: \ \perp \to \mathsf{N} \to \mathsf{N} \\ \mathsf{E}_\perp\,(\mathsf{N} \to \mathsf{N} \to \mathsf{N}) &: \ \perp \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \end{aligned}$$

7 Case Study: Cantor Pairing

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

$E : \mathsf{N} \times \mathsf{N} \to \mathsf{N}$	*encode*
$D : \mathsf{N} \to \mathsf{N} \times \mathsf{N}$	*decode*

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

Existential quantifications can be captured with an inductive predicate

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

and the notation

$$\exists x^t.s \quad \rightsquigarrow \quad \mathsf{ex}\ t\,(\lambda x^t.\ s)$$

An existential quantification $\exists x^t.s$ says that there is some value of type t satisfying the predicate $\lambda x^t.\ s$. This suggests to define the inductive predicate ex with a single proof constructor

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

Simple as it is, this is all we need for the representation of existential quantifications as propositional types.

We will prove two prominent logical facts involving existential quantification: Russell's 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 power set theorem (there is no surjection from a set to its power set).

Given the computational type theory developed so far, the representation of existential quantifications is amazingly straightforward. Essential ingredients are inductive type definitions, dependent function types, lambda abstractions, and the conversion rule.

8.1 Inductive Definition and Basic Facts

Following the design laid out above, we accommodate existential quantification with the inductive type 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		$\neg(\exists x.px) \to \forall x.\neg px$	intro
	$f:\neg(\exists x.px),\ x^X,\ a:px$	$\bot$	apply f
		$\exists x.px$	apply $\mathsf{E}\,x$
		$(\lambda x.px)\,x$	conversion
		px	a
2		$(\forall x.\neg px) \to \neg(\exists x.px)$	intro with $\mathsf{M}_\exists$
	$f:\forall x.\neg px,\ x^X$		
	$a:(\lambda x.px)x$	$\bot$	apply fx
		px	conversion
		$(\lambda x.px)\,x$	a

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

Figure 8.1: Proof of existential de Morgan law with explicit conversions

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

$$\exists x^t.\,s \quad\rightsquigarrow\quad \mathsf{ex}\,(\lambda x^t.\,s)$$

Note that the use of the abstraction $\lambda x^t.\,s$ ensures that x is a local variable visible only in the term s.

Given a proof $\mathsf{E}uv$ of $\exists x^t.s$, we call u the **witness** and v the **certificate** of the proof. More generally, given a unary predicate $p^{X\to\mathbb{P}}$, we call values x^X satisfying p **witnesses** of p.

Note that the elimination restriction applies to the inductive predicate ex since the witness argument x^X of the proof constructor E is not a proof. Hence we cannot extract the witness of a proof of $\exists x.s$ in a computational context.

We define a simply typed match function for existential quantifications that suffices for all proofs in this chapter:

$$\mathsf{M}_\exists:\ \forall X^{\mathbb{T}}\,\forall p^{X\to\mathbb{P}}\,\forall Z^{\mathbb{P}}.\ \mathsf{ex}\,p \to (\forall x.\ px \to Z)\to Z$$
$$\mathsf{M}_\exists\,XpZ\,(\mathsf{E}xa)\,f \ := \ fxa$$

We will use match notation for applications of $\mathsf{M}_\exists$:

$$\textsc{match}\ s\ [\,\mathsf{E}xa \Rightarrow t\,] \quad\rightsquigarrow\quad \mathsf{M}_\exists\,\_\,\_\,\_\,s\,(\lambda xa.t)$$

Figure 8.1 shows a proof diagram and the constructed proof term for a de Morgan law for existential quantification. The proof diagram makes all conversions explicit so that you can see where they are needed.

In practice, it is not a good idea to make explicit inessential conversions like the ones in Figure 8.1. It is much preferable to think modulo conversion. Figure 8.2

	$X:\mathbb{T},\ p:X\to\mathbb{P}$	$\neg(\exists x.px) \longleftrightarrow \forall x.\neg px$	apply C
1		$\neg(\exists x.px) \to \forall x.\neg px$	intro
	$f:\neg(\exists x.px),\ x^X,\ a:px$	$\bot$	apply fx
		$\exists x.px$	$\mathsf{E}xa$
2		$(\forall x.\neg px) \to \neg(\exists x.px)$	intro with $\mathsf{M}_\exists$
	$f:\forall x.\neg px,\ x^X,\ a:px$	$\bot$	fxa

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

Figure 8.2: Proof of existential de Morgan law with implicit conversions

shows a proof diagram with implicit conversions constructing the same proof term. This is certainly a better presentation of the proof. The second diagram gives a fair representation of the interaction you will have with Coq. In fact, Coq will immediately reduce the first two β-redexes you see in Figure 8.1 as part of the proof actions introducing them. This way there will be no need for explicit conversion steps.

Exercise 8.1.1 Prove the following propositions with proof diagrams and give the resulting proof terms. Mark the proof actions involving implicit conversions.

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

b) $(\exists x.\ px \lor qx) \;\longleftrightarrow\; (\exists x.px) \lor (\exists x.qx)$.

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:

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

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

8.2 Barber Theorem

Nonexistence often get 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)
$\forall X^{\mathbb{T}}\,\forall p^{X\to X\to\mathbb{P}}.\ \neg\exists x\,\forall y.\, pxy \longleftrightarrow \neg pyy$.

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 type of the fixed point is falsity.

Do the proof with Coq.

9 Informative Types

Informative types combine computational and propositional information. They are usually obtained with computational variants of disjunctions ($s \vee t$) and existential quantifications ($\exists x.s$) called *sum types* ($s + t$) and *sigma types* ($\Sigma x.s$). Informative types are a unique feature of Coq's type theory having no equivalent in set-theoretic mathematics. With informative types one can describe computational situations that often don't have adequate descriptions in set-theoretic language.

Mathematics comes with a rich language for describing proofs at a high level of abstraction. Using this language, we can write mathematical proofs in such a way that they can be elaborated into formal proofs. The tactic level of the Coq proof assistant provides an abstraction layer for the elaboration of mathematical proofs making it possible to delegate to the proof assistant the details coming with proof terms.

It turns out that the idea of high-level proof extends to the construction of computational functions, provided they are specified with informative types. The *proof-style construction* of computational functions turns out to be very convenient in practice. Technically, the proof-style construction of computational functions comes at low cost since the tactic level of a proof assistant can be designed such that it addresses types in general, not just propositional types. Methodologically, the proof-style construction of a computational function is guided by an informative specification and uses high-level building blocks like induction following ideas familiar from proof construction. One first shows a for-all-exists lemma $\forall x^X \Sigma y^Y.pxy$ and then extracts a function $f^{X \to Y}$ and a correctness lemma $\forall x.\, px(fx)$.

9.1 Lead Examples

Consider the *propositional lemmas*

$$L_1 : \ \forall xy^{\mathsf{N}}.\ x = y \vee x \neq y$$
$$L_2 : \ \forall xy^{\mathsf{N}}\, \exists z.\ x + z = y \vee y + z = x$$

Both lemmas are propositional functions. Informally, the functions may be described as follows:

- Given two numbers x and y, L_1 decides whether they are equal or not and returns a proof certifying the decision.
- Given two numbers x and y, L_2 returns the distance between the numbers and a proof certifying the result.

Both lemmas have routine proofs proceeding by induction on x and case analysis on y.

We now ask how we can obtain computational functions

$$f_1 : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$$
$$f_2 : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$

satisfying the correctness lemma

$$C_1 : \ \forall xy^{\mathsf{N}}.\ \ f_1xy = \mathbf{T} \longleftrightarrow x = y$$
$$C_2 : \ \forall xy^{\mathsf{N}}.\ \ x + f_2xy = y \lor y + f_2xy = x$$

Because of the elimination restriction for disjunctions and existential quantifications, we cannot get computational functions from the lemmas L_1 and L_2. This is unfortunate since the proofs of L_1 and L_2 do contain information on how the computational functions can be constructed.

The situation changes if we prove the *computational lemmas*

$$F_1 : \ \forall xy^{\mathsf{N}}.\ \ (x = y) + (x \neq y)$$
$$F_2 : \ \forall xy^{\mathsf{N}}\,\Sigma z.\ \ (x + z = y) + (y + z = x)$$

using sum types (+) for the disjunctions ($\lor$) and a sigma type (Σ) for the existential quantification ($\exists$). Because sum type and sigma types are computational, there is no elimination restriction and we can easily obtain the computational functions f_1 and f_2 with their correctness lemmas C_1 and C_2 from the computational lemmas F_1 and F_2. We will see that the proof scripts for the propositional lemmas can be reused for the computational lemmas.

9.2 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 $\mathbb{T}$
$\forall$	$\to$	$\land$	$\longleftrightarrow$	$\lor$	$\exists$	propositional types in $\mathbb{P}$

For function types ($\forall$, $\to$) there is no difference. For conjunctions we have product types as computational counterpart, and for propositional equivalence we define the computational variant (**propositional equivalence of types**) as follows:

$$\begin{aligned} \Leftrightarrow &: \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ X \Leftrightarrow Y &:= (X \to Y) \times (Y \to X) \end{aligned}$$

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

The computational counterparts for disjunctions and existential quantifications are called **sum types** ($s + t$) and **sigma types** ($\Sigma x.s$). Their inductive definitions mimic the inductive definitions of disjunctions and existential quantifications by replacing 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:

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

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 called **variants**.

A value $\mathsf{E}\,pxc$ of a sigma type $\Sigma x.px$ may be seen as a **dependent pair** $(x, c)_p$. We speak of the **first** and **second component** of the pair (x and c). We may also refer to x as the **witness** and to c as the **certificate** of the pair. We may write $\mathsf{E}\,xc$ for $\mathsf{E}\,pxc$ if the type function is clear from the context.

While many 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 define simply typed **match functions** for sum types and sigma types following the definitions of the match functions for disjunctions and existential quantifications:

$$\begin{aligned}
&\mathsf{M}_+ : \ \forall XYZ^{\mathbb{T}}.\ X + Y \to (X \to Z) \to (Y \to Z) \to Z\\
&\quad \mathsf{M}_+\, XYZ\,(\mathsf{L}x)\, fg \ := \ fx\\
&\quad \mathsf{M}_+\, XYZ\,(\mathsf{R}y)\, fg \ := \ gy
\end{aligned}$$

$$\text{MATCH}\ s\ [\,\mathsf{L}x \Rightarrow t_1 \mid \mathsf{R}y \Rightarrow t_2\,] \quad \rightsquigarrow \quad \mathsf{M}_+\,\_\,\_\,\_\,s\,(\lambda x.t_1)\,(\lambda y.t_2)$$

$$\begin{aligned}
&\mathsf{M}_\Sigma : \ \forall X^{\mathbb{T}}\,\forall p^{X\to\mathbb{T}}\,\forall Z^{\mathbb{T}}.\ \mathsf{sig}\,p \to (\forall x.\, px \to Z) \to Z\\
&\quad \mathsf{M}_\Sigma\, XpZ\,(\mathsf{E}xa)\, f \ := \ fxa
\end{aligned}$$

$$\text{MATCH}\ s\ [\,\mathsf{E}xa \Rightarrow t\,] \quad \rightsquigarrow \quad \mathsf{M}_\Sigma\,\_\,\_\,\_\,s\,(\lambda xa.t)$$

Using the match functions, we can define so-called **squash functions**

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

from sums of propositions to disjunctions and from sigma types with propositional certificates to existential quantifications. Functions in the other direction cannot be defined because of the elimination restriction.

Exercise 9.2.1 Define the squash functions.

Exercise 9.2.2 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) $\mathsf{R}y = \mathsf{R}y' \to y = y'$.

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

Hint: The techniques used for the constructor laws for numbers (Figure 5.2) also work for sums.

9.3 Lemmas and Proofs

We use the words *lemma* and *proof* informally. Formally, lemmas are accommodated as constants. The idea is that mostly the type of a lemma matters for its use, not the details of its definition. We refer to the definition of a lemma as its *proof*. We speak of a **propositional lemma** if the type of the lemma is a propositional type, and of a **computational lemma** if the type of the lemma is not a propositional type. Propositional lemmas are typically accommodated as declared constants hiding their proofs. For computational lemmas it may be convenient to not hide their proofs so that they can contribute to computational equality.

Lemmas are in contrast to defined constants where the primary information about the constant is in the defining equations.

From now on, when we say *show* t or *prove* t we want to say that a term s of type t is to be constructed. When using *proof-oriented language*, it does not matter whether t is a propositional or a nonpropositional type. The view that one can prove any type is fully realized by the proof mode of the Coq proof assistant, which does not make a difference between propositional and nonpropositional types.

Also keep in mind that we will continue to use the words theorem, corollary, and fact as synonyms for lemma to transport information about the informal status of the lemma.

We are now ready to prove the two computational lemmas discussed in § 9.1. We shall be using proof-oriented language.

Fact 9.3.1 (Certifying equality decider)
$\forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y)$.

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.

Fact 9.3.2 (Distance)
$\forall xy^{\mathsf{N}}\Sigma z.\ (x + z = y) + (y + z = x)$.

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 $\Sigma z.\ (x + z = y) + (y + z = x)$. ∎

Fact 9.3.3 (Sum-sigma equivalence)
$\forall XY^{\mathbb{T}}.\ X + Y \Leftrightarrow \Sigma b.\,\text{IF } b \text{ THEN } X \text{ ELSE } Y$.

Proof Translation $\Rightarrow$: $\lambda a.\ \textsc{match}\ a\ [\,\mathsf{L}\,x \Rightarrow \mathsf{E}\,\mathbf{T}\,x \mid \mathsf{R}\,y \Rightarrow \mathsf{E}\,\mathbf{F}\,y\,]$.
Translation $\Leftarrow$: $\lambda a.\ \textsc{match}\ a\ [\,\mathsf{E}\,bc \Rightarrow (\textsc{if}\ b\ \textsc{then}\ \mathsf{L}\ \textsc{else}\ \mathsf{R})\ c\,]$. ∎

Note that type checking the translation functions of the proof makes heavy use of the conversion rule and computational equality.

Exercise 9.3.4 (Functional characterisations)
Prove the following propositional equivalences for sum types and sigma types:

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

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

Note that the equivalences are analogous to the impredicative characterisations of disjunctions and existential quantifications.

Exercise 9.3.5 Define functions as follows using proof-oriented language:

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

9.4 Projections and Skolem Equivalence

We assume a type function $p : X \to \mathbb{T}$ and define **projections** that yield the first and second component of a depend pair $a : \mathsf{sig}\,p$:

$$\begin{array}{llll} \pi_1 : \ \mathsf{sig}\,p \to X & & \pi_2 : \ \forall a^{\mathsf{sig}\,p}.\ p(\pi_1 a) & \\ \pi_1\,(\mathsf{E}\,x\,\_) \ := \ x & & \pi_2\,(\mathsf{E}\,xc) \ := \ c & \end{array}$$

Note that the type of π_2 is given using the function π_1. This acknowledges the fact that the type of the second component depends on the first component. Type checking the defining equation of π_2 requires conversion steps unfolding the definition of π_1.

Example 9.4.1 (Computational equality)
Let D be the function defined by Fact 9.3.2. Then, for instance, the equation $\pi_1(D\,3\,7) = 4$ holds by computational equality.

We will 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 computational function $f^{X\to Y}$ and a correctness proof $\forall x.\ px(fx)$. The definability of the two translations can be stated elegantly as a propositional equivalence between informative types.

Fact 9.4.2 (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 Skolem equivalence stated by 9.4.2 is of great practical importance in Coq's type theory. Often we will prove a computational lemma $\forall x\,\Sigma y.\ pxy$ to obtain a function $f^{X\to Y}$ satisfying $\forall x.\ px(fx)$.

That we use the term *Skolem equivalence* may be justified with the ressemblance of the equivalence and Skolem functions in first-order logic.

Exercise 9.4.3 Define a simply typed matching function

$$\forall X^{\mathbb{T}}\,\forall p^{X\to\mathbb{T}}\,\forall Z^{\mathbb{T}}.\ \mathsf{sig}\,p \to (\forall x.\ px \to Z) \to Z$$

using the projections π_1 and π_2.

Exercise 9.4.4 (Injectivity laws) Consider dependent pairs $a : \mathsf{sig}\,p$ for $p^{X\to\mathbb{T}}$. 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 general 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. In Chapter 23, we will show that the second injectivity law holds if the type X of the first component has an equality decider.

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.

Exercise 9.4.5 (Propositional Skolem equivalence)
Find out why the propositional version of the Skolem equivalence

$$\forall XY^{\mathbb{T}}\,\forall p^{X\to Y\to\mathbb{T}}.\;(\exists f\,\forall x.\,px(fx))\longleftrightarrow(\forall x\,\exists y.\,pxy)$$

is not provable. Convenience yourself that the unprovability persists even if the law of excluded is assumed. We offer the explanation that the difficulty is with proving the existence of the function f since functions must be constructed with computational principles.

9.5 Full Eliminator for Sigma Types

It turns out that some interesting results about sigma types cannot be shown using the projections. The simply typed match function will help neither since it is definable with the projections. To close the gap, we define a dependently typed eliminator for dependent pairs $a : \mathsf{sig}\,p$ obtained with a type function $p^{X\to\mathbb{T}}$:

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

Note that E_Σ is similar to the eliminator $\mathsf{E}_\times$ for pairs (§ 6.3).

Exercise 9.5.1 Consider dependent pairs $a : \mathsf{sig}\,p$ for a type function $p^{X\to\mathbb{T}}$.

a) Prove the **η-law** $\mathsf{E}\,(\pi_1 a)(\pi_2 a) = a$.

b) Express the projection functions with E_Σ and prove $\forall a.\,\pi_i\,a = t_i\,a$ for your terms t_i.

Exercise 9.5.2 (Distance)
Assume $D : \forall xy^{\mathsf{N}}\,\Sigma z.\;(x+z=y)+(y+z=x)$ and 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 a 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 9.5.3 (Products and sums as sigmas) Two types X and Y are **in bijection** if there are functions $f^{X\to Y}$ and $g^{Y\to X}$ inverting each other (i.e., $g(fx) = x$ and $f(gy) = y$).

a) Show that $X\times Y$ and $\mathsf{sig}\,(\lambda x^X.Y)$ are in bijection.

b) Show that $X+Y$ and $\mathsf{sig}\,(\lambda b^{\mathsf{B}}.\text{ IF } b \text{ THEN } X \text{ ELSE } Y)$ are in bijection.

9.6 Truncation

We now define an inductive operator $\mathcal{T} : \mathbb{T} \to \mathbb{P}$ mapping every type X to a proposition $\mathcal{T}(X)$ that is provable if and only if X is inhabited:

$$\mathcal{T}(X : \mathbb{T}) : \mathbb{P} \ ::= \ T(X)$$

We call $\mathcal{T}$ **truncation operator** and $\mathcal{T}(X)$ the **truncation of X**. Truncation deletes computational information and keeps propositional information. Note that the elimination restriction applies to truncation types. With truncation we can characterize disjunction and existential quantification using sum types and sigma types.

Fact 9.6.1

1. $(P \lor Q) \ \longleftrightarrow \ \mathcal{T}(P + Q)$
2. $(\exists x.px) \ \longleftrightarrow \ \mathcal{T}(\Sigma x.px)$
3. $\mathcal{T}(X) \longleftrightarrow (\exists x^X.\top)$

Fact 9.6.2 Let $X^{\mathbb{T}}$, $Y^{\mathbb{T}}$, and $P^{\mathbb{P}}$. Then:

1. $X \to \mathcal{T}(X)$
2. $P \longleftrightarrow \mathcal{T}(P)$
3. $(X \to Y) \to \mathcal{T}(X) \to \mathcal{T}(Y)$
4. $(X \to P) \to \mathcal{T}(X) \to P$

Exercise 9.6.3 Prove the above facts about truncation.

Exercise 9.6.4 Prove that double negated existential quantification agrees with double negated sigma quantification: $\neg\neg \mathsf{ex}\, p \longleftrightarrow \neg(\mathsf{sig}\, p \to \bot)$.

Exercise 9.6.5 Prove that double negated disjunction agrees with double negated sum: $\neg\neg(P \lor Q) \longleftrightarrow \neg(P + Q \to \bot)$.

10 Decision Types, Discrete Types, and Option Types

Every function definable in computational type theory is algorithmically computable. Thus we can prove within computational type theory that predicates are algorithmically decidable by characterizing them with decision functions. Decidability proofs in computational type theory are formal computability proofs avoiding the tediousness coming with an explicit model of computation such as Turing machines.

We call a predicate *decidable* if it can be characterized with a decision function within computational type theory. Decidable predicates are always algorithmically decidable. Moreover, decidable predicates are always logically decidable, which means that they computationally justify the law of excluded middle for their accompanying propositions (i.e., $\forall x.\ px \lor \neg px$).

Technically, decision functions are best realized as certifying deciders using special sum types called *decision types*. It turns out that the propagation laws for deciders (i.e., decision functions) mostly derive from the propagation laws for *decisions* (elements of decision types).

A *discrete type* is a type that comes with a decider for its equality predicate. Concrete data types like the booleans or the numbers always come equality deciders.

We also introduce option types, which are inductive types frequently used in functional programming and type theory. Option types are obtained with a type constructor that extends a given type with a single new element. Option types preserve discreteness.

10.1 Decision Types and Certifying Deciders

We define **decision types** as follows:

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

We call values of decision types **decisions**. A decision of type $\mathcal{D}(X)$ carries either an element of X or a proof that X is void. 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 satisfiability predicate for tests on numbers

$$\begin{aligned} \mathsf{tsat} &: \mathsf{N} \to \mathsf{B} \\ \mathsf{tsat}\, f &:= \exists n.\, fn = \mathbf{T} \end{aligned}$$

is not algorithmically decidable. Hence it cannot be shown in computational 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 10.1.1 (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 Let $F : \forall x. \mathcal{D}(px)$. We define a boolean decider $fx :=$ IF Fx THEN $\mathbf{T}$ 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 10.1.2 (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 10.1.3 Prove the claims of Fact 10.1.2.

Exercise 10.1.4 Define a function $\forall X^{\mathbb{T}} f^{X \to \mathsf{B}} x^X.\ \mathcal{D}(fx = \mathbf{T})$.

Exercise 10.1.5 Define functions as follows:

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

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

10.2 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 10.2.1 (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 9.3.1. The other claims all have straightforward proofs. We use $\mathsf{injective}\, f := \forall xx'.\ fx = fx' \to x = x'$. ■

Exercise 10.2.2 Proof the claims of Fact 10.2.1.

Exercise 10.2.3 Prove that a type has a certifying equality decider if and only if it has a boolean equality decider: $\forall X.\ \mathcal{E}(X) \Leftrightarrow \Sigma f^{X \to X \to \mathsf{B}}.\ \forall xy.\ x = y \longleftrightarrow fxy = \mathbf{T}$.

10.3 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 inductive type definition introduces the constructors

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

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

Proof Follows with the techniques used for the constructor laws for numbers (Figure 5.2). Exercise. ■

Fact 10.3.2 (Discreteness)
$\forall X^{\mathbb{T}}.\ \mathcal{E}(X) \to \mathcal{E}(\mathcal{O}(X))$.

Proof Exercise. ■

Exercise 10.3.3 Prove $\forall a^{\mathcal{O}(X)}.\ a \neq \emptyset \Leftrightarrow \Sigma x.\ a = {}^\circ x$.

Exercise 10.3.4 (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 10.3.5 (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 10.3.6 (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 Extensionality

Computational type theory does not fully determine equality of functions, propositions, and proofs. The missing commitment can be added through extensionality axioms.

11.1 Extensionality Axioms

Computational type theory fails to fully determine equality between functions, propositions, and proofs:

- Given two functions of the same type that agree on all elements, computational type theory does not prove that the functions are equal.
- Given two equivalent propositions, computational type theory does not prove that the propositions are equal.
- Given two proofs of the same proposition, computational type theory does not prove that the proofs are equal.

From a modeling perspective, it would be desirable to add the missing proof power for functions, propositions, and proofs. This can be done with three axioms expressible as propositions:

- **Function extensionality**
  FE $:= \ \forall X^{\mathbb{T}}\ \forall p^{X \to \mathbb{T}}.\ \forall f g^{\forall x.px}.\ (\forall x. fx = gx \to fx = gx) \to f = g$
- **Propositional extensionality**
  PE $:= \ \forall PQ^{\mathbb{P}}.\ (P \longleftrightarrow Q) \to P = Q$
- **Proof irrelevance**
  PI $:= \ \forall Q^{\mathbb{P}}.\ \forall ab^{Q}.\ a = b$

Function extensionality gives us the equality for functions we are used to from set-theoretic foundations. Together, function and propositional extensionality turn predicates $X \to \mathbb{P}$ into sets: Two predicates (i.e., sets) are equal if and only if they have the same witnesses (i.e., elements). Proof irrelevance tells that functions taking proofs as arguments don't depend on the particular proofs given. This way propositional arguments can play the role of preconditions. Moreover, dependent pair types sig p taken over predicates $p^{X \to \mathbb{P}}$ can model subtypes of X. Proof irrelevance also gives us dependent pair injectivity in the second component (§ 23.2).

There is general agreement that a computational type theory should be extensional, that is, prove FE, PE, and PI. In our case, we may assume FE, PE, and PI as axioms. There are general results saying that adding these extensionality axioms is consistent, that is, does not enable a proof of falsity. There is research underway aiming at a computational type theory integrating extensionality axioms in such a way that canonicity of the type theory is preserved. This is not the case in our setting since reduction of a term build with constants for the axioms may get stuck on one of the constants before a canonical term is reached.

Coq offers a facility that determines the axiomatic constants a constant depends on. Terms not depending on axiomatic constants are guaranteed to reduce to canonical terms.

Exercise 11.1.1 Prove $\mathsf{FE} \to \forall f^{\top\to\top}.\ f = \lambda a^{\top}.a$.

Exercise 11.1.2 Prove $\mathsf{FE} \to \mathsf{B} \neq (\top \to \top)$.

Exercise 11.1.3
Prove $\mathsf{FE} \to \forall f^{\mathsf{B}\to\mathsf{B}}.\ (f = \lambda b.\, b) \vee (f = \lambda b.\, !b) \vee (f = \lambda b.\, \mathbf{T}) \vee (f = \lambda b.\, \mathbf{F})$.

11.2 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:

$\emptyset$	$:=$	$\lambda x^X.\bot$	empty set
$p \cap q$	$:=$	$\lambda x^X.px \wedge qx$	intersection
$p \cup q$	$:=$	$\lambda x^X.px \vee qx$	union
$p - q$	$:=$	$\lambda x^X.px \wedge \neg qx$	difference

Exercise 11.2.1 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 11.2.2 Assume FE and PE and prove the following:

a) $(\forall x.\, x \in p \longleftrightarrow x \in q) \to p = q$ *set extensionality*

b) $p - (q \cup r) = (p - q) \cap (p - r)$

11.3 Proof Irrelevance

It turns out that PI is a straightforward consequence of PE.

We call a type **pure** if it has at most one element:

$$\mathsf{pure}\,(X^{\mathbb{T}}) \;:=\; \forall xy^X.\, x = y$$

Note that PI says that all propositions are pure.

Fact 11.3.1 $\bot$ and $\top$ are pure.

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

Fact 11.3.2 $\mathsf{PE} \to \mathsf{PI}$.

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. ■

Exercise 11.3.3 Prove that every pure type is discrete.

Exercise 11.3.4 Prove $\mathsf{FE} \to \mathsf{pure}\,(\top \to \top)$.

Exercise 11.3.5 Assume PI and $p^{X \to \mathbb{P}}$. Prove $\forall xy\, \forall ab.\; x = y \to (x, a)_p = (y, b)_p$.

Exercise 11.3.6 Suppose there is a function $f : (\top \vee \top) \to \mathsf{B}$ such that $f(\mathsf{L\,I}) = \mathbf{T}$ and $f(\mathsf{R\,I}) = \mathbf{F}$. Prove $\neg\,\mathsf{PI}$. Convince yourself that without the elimination restriction you could define a function f as assumed.

Exercise 11.3.7 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 elimination restriction you could define a function f as assumed.

11.4 Notes

We will always make explicit when we use extensionality assumptions. It turns out that most of the theory in this text does not require extensionality assumptions.

12 Excluded Middle and Double Negation

One of first laws of logic one learns in an introductory course on mathematics is excluded middle saying that a proposition is either true or false. On the other hand, computational type theory does not prove $P \vee \neg P$ for every proposition P. It turns out that most results in computational mathematics can be formulated such that they can be proved without assuming a law of excluded middle, and that such a constructive account gives more insight than a naive account using excluded middle. On the other hand, the law of excluded middle can be formulated with the proposition

$$\forall P^{\mathbb{P}}.\, P \vee \neg P$$

and assuming it in computational type theory is consistent and meaningful.

In this chapter, we study several characterizations of excluded middle and the special reasoning patterns provided by excluded middle. We show that these reasoning patterns are locally available for double negated claims without assuming excluded middle.

12.1 Characterizations of Excluded Middle

We formulate the law of excluded middle with the proposition

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

Computational type theory neither proves nor disproves XM. Thus it is interesting to assume XM and study its consequences. This study becomes most revealing if we assume XM only locally using implication.

There are several propositionally equivalent characterizations of excluded middle. Most amazing is may be Peirce's law that formulates excluded middle with just implication.

Fact 12.1.1 The following propositions are equivalent. That is, if we can prove one of them, we can prove all of them.

1. $\forall P^{\mathbb{P}}.\ P \vee \neg P$ *excluded middle*
2. $\forall P^{\mathbb{P}}.\ \neg\neg P \to P$ *double negation*
3. $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ (\neg P \to \neg Q) \to Q \to P$ *contraposition*
4. $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ ((P \to Q) \to P) \to P$ *Peirce's law*

Proof We prove the implications $1 \to 2 \to 3 \to 4 \to 1$.

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

$2 \to 3$. Assume $\neg P \to \neg Q$ and Q and show P. By (2) it suffices to show $\neg\neg P$. We assume $\neg P$ and show $\bot$. Follows from the assumptions.

$3 \to 4$. By (3) it suffices to show $\neg P \to \neg((P \to Q) \to P))$. Straightforward.

$4 \to 1$. By (4) with $P \mapsto (P \vee \neg P)$ and $Q \mapsto \bot$ we can assume $\neg(P \vee \neg P)$ and prove $P \vee \neg P$. We assume P and prove $\bot$. Straightforward since we have $\neg(P \vee \neg P)$. ∎

A common 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:

$$\mathsf{XM} \to (\neg P \to \bot) \to P$$

There is another characterization of excluded middle asserting existence of counterexamples, often used as tacit assumption in mathematical arguments.

Fact 12.1.2 (Counterexample) $\mathsf{XM} \longleftrightarrow \forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}}.\ (\forall x.px) \vee \exists x.\neg px.$

Proof Assume XM and $p^{X \to \mathbb{P}}$. By XM we assume $\neg\exists x.\neg px$ and prove $\forall x.px$. By the de Morgan law for existential quantification we have $\forall x.\neg\neg px$. The claim follows since XM implies the double negation law.

Now assume the right hand side and let P be a proposition. We prove $P \vee \neg P$. We choose $p := \lambda a^{\top}.P$. By the right hand side and conversion we have either $\forall a^{\top}.P$ or $\exists a^{\top}.\neg P$. In each case the claim follows. Note that choosing an inhabited type for X is essential. ∎

Figure 12.1 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

$$\begin{array}{rcll} \neg(P \vee Q) & \longleftrightarrow & \neg P \wedge \neg Q & \qquad \text{de Morgan} \\ \neg(\exists x.px) & \longleftrightarrow & \forall x.\neg px & \qquad \text{de Morgan} \end{array}$$

have constructive proofs.

Exercise 12.1.3

a) Prove the right-to-left directions of the equivalences in Figure 12.1.

b) Prove the left-to-right directions of the equivalences in Figure 12.1 using XM.

$\neg(P \wedge Q)$	$\longleftrightarrow$	$\neg P \vee \neg Q$	de Morgan
$\neg(\forall x.px)$	$\longleftrightarrow$	$\exists x.\neg px$	de Morgan
$(\neg P \to \neg Q)$	$\longleftrightarrow$	$(Q \to P)$	contraposition
$(P \to Q)$	$\longleftrightarrow$	$\neg P \vee Q$	classical implication

Figure 12.1: Prominent equivalences only provable with XM

Exercise 12.1.4 Prove the following equivalences possibly using XM. In each case find out which direction needs XM.

$$\begin{aligned} \neg(\exists x.\neg px) &\longleftrightarrow \forall x.px \\ \neg(\exists x.\neg px) &\longleftrightarrow \neg\neg\forall x.px \\ \neg\neg(\exists x.px) &\longleftrightarrow \neg\forall x.\neg px \end{aligned}$$

Exercise 12.1.5 Prove that the left-to-right direction of the de Morgan law for universal quantification implies XM:

$$(\forall P^{\mathbb{T}} \forall p^{P \to \mathbb{P}}.\ \neg(\forall x.px) \to (\exists x.\ \neg px)) \to \mathsf{XM}$$

Hint: Instantiate the de Morgan law with $P \vee \neg P$ and $\lambda\_.\bot$.

Exercise 12.1.6 Make sure you can prove the de Morgan laws for disjunction and existential quantification (not using XM).

Exercise 12.1.7 Explain why Peirce's law and the double negation law are independent in Coq's type theory.

Exercise 12.1.8 (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 $\exists x.\ px \to \forall y.py$. 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 12.1.2.

An informal proof may proceed as follows. Either everyone in the bar is drinking Whisky. Then we can pick any person for x. Otherwise, we pick a person for x not drinking Whisky, making the implication vacuously true.

There is a paper [21] on the drinker paradox suggesting that the drinker proposition does not imply excluded middle.

12.2 Double Negation

Given a proposition P, we call $\neg\neg P$ the **double negation** of P. 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,

$$\forall P^{\mathbb{P}}.\ \neg\neg(P \vee \neg P)$$

is provable. This metaproperty cannot be proved in Coq. However, for every instance a proof can be given in Coq. Moreover, for concrete propositional proof systems the translation of classical proofs into constructive proofs of the double negated claim can be formalized and verified (Glivenko's theorem 30.8.2).

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 an instance of the polymorphic identity function:

$$\neg\neg P \to (P \to \bot) \to \bot$$

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 12.2.1 The following **distribution laws** for double negation are provable:

$$\begin{aligned} \neg\neg(P \to Q) &\longleftrightarrow (\neg\neg P \to \neg\neg Q) \\ \neg\neg(P \wedge Q) &\longleftrightarrow \neg\neg P \wedge \neg\neg Q \\ \neg\neg\top &\longleftrightarrow \top \\ \neg\neg\bot &\longleftrightarrow \bot \end{aligned}$$

Exercise 12.2.2 Prove the equivalences of Fact 12.2.1.

Exercise 12.2.3 Prove the following propositions:

$$\begin{aligned} \neg(P \wedge Q) &\longleftrightarrow \neg\neg(\neg P \vee \neg Q) \\ (\neg P \to \neg Q) &\longleftrightarrow \neg\neg(Q \to P) \\ (\neg P \to \neg Q) &\longleftrightarrow (Q \to \neg\neg P) \\ (P \to Q) &\to \neg\neg(\neg P \vee Q) \end{aligned}$$

Exercise 12.2.4 Prove $\neg(\forall x.\neg px) \longleftrightarrow \neg\neg\exists x.px$.

Exercise 12.2.5 Prove the following implications:

$$\begin{aligned} \neg\neg P \vee \neg\neg Q &\to \neg\neg(P \vee Q) \\ (\exists x.\,\neg\neg px) &\to \neg\neg\exists x.px \\ \neg\neg(\forall x.px) &\to \forall x.\,\neg\neg px \end{aligned}$$

Convince yourself that the converse directions are not provable without excluded middle.

Exercise 12.2.6 Make sure you can prove the double negations of the following propositions:

$$\begin{gathered} P \vee \neg P \\ \neg\neg P \to P \\ \neg(P \wedge Q) \to \neg P \vee \neg Q \\ (\neg P \to \neg Q) \to Q \to P \\ ((P \to Q) \to P) \to P \\ (P \to Q) \to \neg P \vee Q \\ (P \to Q) \vee (Q \to P) \end{gathered}$$

12.3 Stable Propositions

We define **stable propositions** as follows:

$$\mathsf{stable}\ P^{\mathbb{P}} \;:=\; \neg\neg P \to P$$

We may see stable propositions as propositions where double negation elimination is possible. We will see that the XM-avoiding proof techniques for double negated propositions extend to stable propositions.

Fact 12.3.1 $\mathsf{XM} \longleftrightarrow \forall P^{\mathbb{P}}.\ \mathsf{stable}\,P$.

Fact 12.3.2 (Characterizations) The following propositions are equivalent:

1. $\mathsf{stable}\,P$
2. $P \longleftrightarrow \neg\neg P$
3. $\exists Q^{\mathbb{P}}.\ P \longleftrightarrow \neg Q$ *negative characterization*

Corollary 12.3.3 Negated propositions are stable: $\forall P^{\mathbb{P}}.\ \mathsf{stable}(\neg P)$.

Fact 12.3.4 Decidable propositions are stable: $\forall P^{\mathbb{P}}.\ \mathcal{D}(P) \to \mathsf{stable}\,P$.

Fact 12.3.5 $\top$ and $\bot$ are stable.

Fact 12.3.6 (Closure Rules)
Implication, conjunction, and universal quantification preserve stability:

1. $\mathsf{stable}\, Q \;\to\; \mathsf{stable}\,(P \to Q)$.
2. $\mathsf{stable}\, P \;\to\; \mathsf{stable}\, Q \;\to\; \mathsf{stable}\,(P \wedge Q)$.
3. $(\forall x.\, \mathsf{stable}\,(px)) \;\to\; \mathsf{stable}\,(\forall x. px)$.

Fact 12.3.7 (Extensionality) Stability is invariant under propositional equivalence: $(P \longleftrightarrow Q) \to \mathsf{stable}\, P \to \mathsf{stable}\, Q$.

Stable propositions matter since there are proof rules providing classical reasoning for stable claims.

Fact 12.3.8 (Classical reasoning rules for stable claims)

1. $\mathsf{stable}\, Q \to (P \vee \neg P \to Q) \to Q$.
2. $\mathsf{stable}\, Q \to \neg\neg P \to (P \to Q) \to Q$.

The first rule says that when we prove a stable claim, we can assume $P \vee \neg P$ for every proposition P. The second rule says that when we prove a stable claim, we can obtain P from a double negated assumption $\neg\neg P$.

Exercise 12.3.9 Prove the above facts. All proofs are straightforward.

Exercise 12.3.10 Prove the following classical reasoning rules for stable claims:

a) $\mathsf{stable}\, Q \to (P \to Q) \to (\neg P \to Q) \to Q$.
b) $\mathsf{stable}\, Q \to \neg(P_1 \wedge P_2) \to (\neg P_1 \vee \neg P_2 \to Q) \to Q$.
c) $\mathsf{stable}\, Q \to (\neg P_1 \to \neg P_2) \to ((P_2 \to P_1) \to Q) \to Q$.

Exercise 12.3.11 Prove $(\forall x.\, \mathsf{stable}\,(px)) \;\to\; \neg(\forall x. px) \;\longleftrightarrow\; \neg\neg\exists x.\, \neg px$.

Exercise 12.3.12 Prove $\mathsf{FE} \to \forall f g^{\mathsf{N}\to\mathsf{B}}.\ \mathsf{stable}(f = g)$.

Exercise 12.3.13 We define **classical variants** of conjunction, disjunction, and existential quantification:

$$\begin{array}{rcll} P \wedge_c Q & := & (P \to Q \to \bot) \to \bot & \qquad \neg(P \to \neg Q) \\ P \vee_c Q & := & (P \to \bot) \to (Q \to \bot) \to \bot & \qquad \neg P \to \neg\neg Q \\ \exists_c x. px & := & (\forall x. px \to \bot) \to \bot & \qquad \neg(\forall x. \neg px) \end{array}$$

The definitions are obtained from the impredicative characterisations of $\wedge$, $\vee$, and $\exists$ 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) $P \wedge Q \to P \wedge_c Q$ and $P \wedge_c Q \longleftrightarrow \neg\neg(P \wedge Q)$.

b) $P \vee Q \to P \vee_c Q$ and $P \vee_c Q \longleftrightarrow \neg\neg(P \vee Q)$.

c) $(\exists x.px) \to \exists_c x.px$ and $(\exists_c x.px) \longleftrightarrow \neg\neg(\exists x.px)$.

d) $P \vee_c \neg P$.

e) $\neg(P \wedge_c Q) \longleftrightarrow \neg P \vee_c \neg Q$.

f) $(\forall x.\ \mathsf{stable}\,(px)) \to \neg(\forall x.px) \longleftrightarrow \exists_c x.\neg px$.

g) $P \wedge_c Q$, $P \vee_c Q$, and $\exists_c x.px$ are stable.

12.4 Discrimination Lemma for Non-repeating Lists

Using XM, we can prove that every non-repeating list contains for every shorter list an element that is not in the shorter list:

$$\mathsf{XM} \to \forall X\, \forall AB^{\mathcal{L}(X)}.\ \mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \exists x.\ x \in A \wedge x \notin B$$

We speak of a *discriminating element* and the *classical discrimination lemma*. In the chapter on lists, we prove a computational version of the lemma (Fact 17.7.5), which replaces XM with an equality decider for the base type X. Here our main interest is in proving the *constructive discrimination lemma*

$$\forall X\, \forall AB^{\mathcal{L}(X)}.\ \mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \neg\neg\exists x.\ x \in A \wedge x \notin B$$

which assumes neither XM nor an equality decider. Note that the classical discrimination lemma is a trivial consequence of the constructive discrimination lemma. We may say that the constructive discrimination lemma is obtained from the classical discrimination lemma by eliminating the use of XM by weakening the existential claim with a double negation. Elimination technique for XM have useful applications.

We first prove the classical discrimination lemma. The proof is by induction on A and at some point we need to shorten B so that we can apply the inductive hypothesis. For the computational variant the shortening of B was done with an element removal function. This is not an option for the classical variant since no equality decider is available. However, we can still obtain a shortened list with the necessary property using existential quantification.

Lemma 12.4.1 (Shortening) $x \in B \to \exists B'.\ \mathsf{len}\,B' < \mathsf{len}\,B \wedge B \subseteq x :: B'$.

Proof By induction on B. If B is empty, the assumption $x \in B$ is contradictory. If $B = b :: B'$, the assumption provides for the case analysis $x = b \lor x \in B'$. If $x = b$, B' is the list we are looking for. Otherwise, we use the inductive hypothesis on x and B'. ■

Lemma 12.4.2 (Classical discrimination)
$\mathsf{XM} \to \forall AB^{\mathcal{L}(X)}.\ \mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \exists x.\ x \in A \land x \notin B.$

Proof By induction on A with B quantified. If A is empty, the assumption $\mathsf{len}\,B < \mathsf{len}\,A$ is contradictory. Otherwise, we have $A = a :: A'$. We use XM to do case analysis on $a \in B \lor a \notin B$. If $a \notin B$, we have found a discriminating element. Otherwise, we have $a \in B$. We use the shortening lemma to obtain B' such that $\mathsf{len}\,B' < \mathsf{len}\,A'$ and $B \subseteq a :: B'$. The inductive hypothesis now gives us $x \in A'$ such that $x \notin B'$. Thus $x \in A$ and $x \notin B$ since $a \notin A'$. ■

We observe that there is only a single use of XM. When we prove the constructive version with the double negated claim, we will exploit that XM is available for stable claims (Fact 12.3.8 (1)). We will use the rule formulated by Fact 12.3.8 (2) to erase the double negation from the inductive hypothesis so that we can harvest the witness.

Lemma 12.4.3 (Constructive discrimination)
$\forall AB^{\mathcal{L}(X)}.\ \mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \neg\neg\exists x.\ x \in A \land x \notin B.$

Proof By induction on A with B quantified. If A is empty, the assumption $\mathsf{len}\,B < \mathsf{len}\,A$ is contradictory. Otherwise, we have $A = a :: A'$. Since the claim is stable, we can do case analysis on $a \in B \lor a \notin B$ by Fact 12.3.8 (1). If $a \notin B$, we have found a discriminating element and finish the proof with $\forall P.\ P \to \neg\neg P$. Otherwise, we have $a \in B$. We use the shortening lemma to obtain B' such that $\mathsf{len}\,B' < \mathsf{len}\,A'$ and $B \subseteq a :: B'$. Using Fact 12.3.8 (2), the inductive hypothesis now gives us $x \in A'$ such that $x \notin B'$. Thus $x \in A$ and $x \notin B$ since $a \notin A'$. We finish the proof with $\forall P.\ P \to \neg\neg P$. ■

Exercise 12.4.4 Prove that the double negation of $\exists$ agrees with the double negation of Σ: $\neg\neg\mathsf{ex}\,p \longleftrightarrow \neg(\mathsf{sig}\,p \to \bot)$.

12.5 Definite Propositions

We define **definite propositions** as follows:

$$\mathsf{definite}\ P^{\mathbb{P}} \ := \ P \lor \neg P$$

We may see definite propositions as propositionally decided propositions. Computationally decided propositions are always propositionally decided, but bot necessarily vice versa. The structural properties of definite propositions are familiar from decided propositions.

Fact 12.5.1 $\mathsf{XM} \longleftrightarrow \forall P^{\mathbb{P}}.\ \mathsf{definite}\, P$.

Fact 12.5.2

1. Decidable propositions are definite: $\forall P^{\mathbb{P}}.\ \mathcal{D}(P) \to \mathsf{definite}\, P$.
2. Definite propositions are stable: $\forall P^{\mathbb{P}}.\ \mathsf{definite}\, P \to \mathsf{stable}\, P$.
3. $\top$ and $\bot$ are definite.
4. Definiteness is invariant under propositional equivalence.

Fact 12.5.3 (Closure Rules)
Implication, conjunction, disjunction, and negation preserve definiteness:

1. $\mathsf{definite}\ P \ \to\ \mathsf{definite}\ Q \ \to\ \mathsf{definite}\,(P \to Q)$.
2. $\mathsf{definite}\ P \ \to\ \mathsf{definite}\ Q \ \to\ \mathsf{definite}\,(P \wedge Q)$.
3. $\mathsf{definite}\ P \ \to\ \mathsf{definite}\ Q \ \to\ \mathsf{definite}\,(P \vee Q)$.
4. $\mathsf{definite}\ P \ \to\ \mathsf{definite}\,(\neg P)$.

Fact 12.5.4 (Definite de Morgan) $\mathsf{definite}\, P \vee \mathsf{definite}\, Q \ \to\ \neg(P \wedge Q) \ \longleftrightarrow\ \neg P \vee \neg Q$.

Exercise 12.5.5 Prove the above facts.

12.6 Variants of Excluded Middle

A stronger formulation of excluded middle is **truth value semantics**:

$$\mathsf{TVS} \ :=\ \forall P^{\mathbb{P}}.\ P = \top \vee P = \bot$$

TVS is equivalent to the conjunction of XM and PE.

Fact 12.6.1 $\mathsf{TVS} \longleftrightarrow \mathsf{XM} \wedge \mathsf{PE}$.

Proof We show $\mathsf{TVS} \to \mathsf{PE}$. Let $P \longleftrightarrow Q$. We apply TVS to P and Q. If they are both assigned $\bot$ or $\top$, we have $P = Q$. Otherwise we have $\top \longleftrightarrow \bot$, which is contradictory. The remaining implications $\mathsf{TVS} \to \mathsf{XM}$ and $\mathsf{XM} \wedge \mathsf{PE} \to \mathsf{TVS}$ are also straightforward. ∎

There are interesting weaker formulations of excluded middle. We consider two of them in exercises appearing below:

$\mathsf{WXM} \ :=\ \forall P^{\mathbb{P}}.\ \neg P \vee \neg\neg P$	**weak excluded middle**
$\mathsf{IXM} \ :=\ \forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ (P \to Q) \vee (Q \to P)$	**implicational excluded middle**

Altogether we have the following hierarchy: $\mathsf{TVS} \Rightarrow \mathsf{XM} \Rightarrow \mathsf{IXM} \Rightarrow \mathsf{WXM}$.

Exercise 12.6.2 Prove TVS $\longleftrightarrow \forall XYZ:\mathbb{P}.\ X = Y \lor X = Z \lor Y = Z$. Note that the equivalence characterizes TVS without using $\top$ and $\bot$.

Exercise 12.6.3 Prove 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 12.6.4 Prove $(\forall X^{\mathbb{T}}.\ X = \top \lor X = \bot) \to \bot$.

Exercise 12.6.5 (Weak excluded middle)

a) Prove XM $\to$ WXM.

b) Prove WXM $\longleftrightarrow \forall P^{\mathbb{P}}.\ \neg\neg P \lor \neg\neg\neg P$.

c) Prove WXM $\longleftrightarrow \forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ \neg(P \land Q) \to \neg P \lor \neg Q$.

Note that (c) says that WXM is equivalent to the de Morgan law for conjunction. We remark that computational type theory proves neither WXM nor WXM $\to$ XM.

Exercise 12.6.6 (Implicational excluded middle)

a) Prove XM $\to$ IXM.

b) Prove IXM $\to$ WXM.

c) Assuming that computational type theory does not prove WXM, argue that computational type theory proves neither IXM nor XM nor TVS.

We remark that computational type theory does not prove WXM. Neither does computational type theory prove any of the implications WXM $\to$ IXM, IXM $\to$ XM, and XM $\to$ TVS.

12.7 Notes

Proof systems not building in excluded middle are called *intuitionistic proof systems*, and proof systems building in excluded middle are called *classical proof systems*. The proof system coming with computational type theory is clearly an intuitionistic system. What we have seen in this chapter is that an intuitionistic proof system provides for a fine grained analysis of excluded middle. This is in contrast to a classical proof system that by construction does not support the study of excluded middle. It should be very clear from this chapter that an intuitionistic system provides for classical reasoning (i.e., reasoning with excluded middle) while a classical system does not provide for intuitionistic reasoning (i.e., reasoning without excluded middle).

Classical and intuitionistic proof systems have been studied for more than a century. That intuitionistic reasoning is not made explicit in current introductory teaching of mathematics has social reasons tracing back to early advocates of intuitionistic reasoning that also argued against the use of excluded middle.

13 Provability

A central notion of computational type theory and related systems is provability. A type (or more specifically a proposition) is *provable* if there is a term that type checks as a member of this type. Importantly, type checking is a decidable relation between terms that can be machine checked. We say that provability is a *verifiable relation*. Given the explanations in this text and the realization provided by the proof assistant Coq, we are on solid ground when we construct proofs.

In contrast to provability, unprovability is not a verifiable relation. Thus the proof assistant will, in general, not be able to certify that types are unprovable.

As it comes to unprovability, this text makes some strong assumptions that cannot be verified with the methods the text develops. The most prominent such assumption says that falsity is unprovable.

Recall that we call a type X *disprovable* if the type $X \to \bot$ is provable. If we trust in the assumption that falsity is unprovable, every disprovable type is unprovable. Thus disprovable types give us a class of types for which unprovability is verifiable up to the assumption that falsity is unprovable.

Types that are neither provable nor disprovable are called *independent types*. There are many independent types. In fact, the extensionality axioms from Chapter 11 and the different variants of excluded middle from Chapter 12 are all claimed independent. These claims are backed up by model-theoretic studies in the literature.

13.1 Provability Predicates

It will be helpful to assume an abstract **provability predicate**

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

With this trick $\mathsf{provable}\,(P)$ and $\neg\mathsf{provable}\,(P)$ are both propositions in computational type theory we can reason about. We define three standard notions for propositions and the assumed provability predicate:

$$\begin{aligned}
\mathsf{disprovable}\,(P) &:= \mathsf{provable}\,(\neg P)\\
\mathsf{consistent}\,(P) &:= \neg\mathsf{provable}\,(\neg P)\\
\mathsf{independent}\,(P) &:= \neg\mathsf{provable}\,(P) \land \neg\mathsf{provable}\,(\neg P)
\end{aligned}$$

With these definitions we can easily prove the following assumptions:

$$\begin{aligned}
\mathsf{independent}\,(P) &\to \mathsf{consistent}\,(P)\\
\mathsf{consistent}\,(P) &\to \neg\mathsf{disprovable}\,(P)\\
\mathsf{provable}\,(P) &\to \neg\mathsf{independent}\,(P)\\
\mathsf{disprovable}\,(P) &\to \neg\mathsf{provable}\,(P)
\end{aligned}$$

To show more, we make the following assumptions about the assumed provability predicate:

$$\begin{aligned}
\mathsf{PMP}: &\ \ \forall PQ.\ \ \mathsf{provable}\,(P \to Q) \to \mathsf{provable}\,(P) \to \mathsf{provable}\,(Q)\\
\mathsf{PI}: &\ \ \forall P.\ \ \mathsf{provable}\,(P \to P)\\
\mathsf{PK}: &\ \ \forall PQ.\ \mathsf{provable}\,(Q) \to \mathsf{provable}\,(P \to Q)\\
\mathsf{PC}: &\ \ \forall PQZ.\ \ \mathsf{provable}\,(P \to Q) \to \mathsf{provable}\,((Q \to Z) \to P \to Z)
\end{aligned}$$

Since the provability predicate coming with computational type theory satisfies these properties, we can expect that properties we can show for the assumed provability predicate also hold for the provability predicate coming with computational type theory.

Fact 13.1.1 (Consistency) The following propositions are equivalent:

1. $\neg\,\mathsf{provable}\,(\bot)$.
2. $\mathsf{consistent}\,(\neg\bot)$.
3. $\exists P.\,\mathsf{consistent}\,(P)$.
4. $\forall P.\,\mathsf{provable}\,(P) \to \mathsf{consistent}\,(P)$.

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

2 → 3. Trivial.

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

1 → 4. We assume that $\bot$ is unprovable, P is provable, and $\neg P$ is provable. By PMP 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. ■

Fact 13.1.2 (Transport)

1. $\mathsf{provable}(P \to Q)\ \to\ \neg\mathsf{provable}\,Q\ \to\ \neg\,\mathsf{provable}\,(P)$.
2. $\mathsf{provable}(P \to Q)\ \to\ \mathsf{consistent}\,(P)\ \to\ \mathsf{consistent}\,(Q)$.

Proof Claim 1 follows with PMP. Claim 2 follows with PC and (1). ■

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

Fact 13.1.3 (Sandwich) A proposition Z is independent if there exists a consistent proposition P and an unprovable proposition Q such that $P \to Z$ and $Z \to Q$ are provable: $\mathsf{consistent}\,(P) \to \neg\mathsf{provable}\,Q \to (P \to Z) \to (Z \to Q) \to \mathsf{independent}\,(Z)$.

Proof Follows with Fact 13.1.2. ■

Exercise 13.1.4 Show that the functions $\lambda P^{\mathbb{P}}.P$ and $\lambda P^{\mathbb{P}}.\top$ are provability predicates satisfying PMP, PI, PK, and PC.

Exercise 13.1.5 Let $P \to Q$ be provable. Show that P and Q are both independent if P is consistent and Q is unprovable.

Exercise 13.1.6 Assume that the provability predicate satisfies

$$\mathsf{PE}: \ \forall P^{\mathbb{P}}.\ \mathsf{provable}\,(\bot) \to \mathsf{provable}\,(P)$$

in addition to PMP, PI, PK, and PC. Prove $\neg\mathsf{provable}\,(\bot) \longleftrightarrow \neg\forall P^{\mathbb{P}}.\ \mathsf{provable}\,(P)$.

Exercise 13.1.7 (Hilbert style assumptions) The assumptions PI, PK, and PC can be obtained from the simpler assumptions

$$\begin{aligned} \mathsf{PK}': &\ \forall PQ.\ \mathsf{provable}\,(P \to Q \to P) \\ \mathsf{PS}: &\ \forall PQZ.\ \mathsf{provable}\,((P \to Q \to Z) \to (P \to Q) \to P \to Z) \end{aligned}$$

that will look familiar to people acquainted with propositional Hilbert systems. Prove PK, PI, and PC from the two assumptions above. PK and PI are easy. PC is difficult if you don't know the technique. You may follow the proof tree $S(S(KS)(S(KK)I))(KH)$. Hint: PI follows with the proof tree SKK.

The exercise was prompted by ideas of Jianlin Li in July 2020.

Exercise 13.1.8 We may consider more abstract provability predicates

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

where prop is an assumed type of propositions with an assumed constant

$$\mathsf{impl}: \ \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$$

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

14 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 functions for trichotomy and Euclidean division. There is much beauty in developing the theory of numbers from first principles.

14.1 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 $\mathsf{E_N}$ providing for inductive proofs on numbers:

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

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 14.1.1 (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 14.1.2 (Discreteness) N is a discrete type: $\forall xy^{\mathsf{N}}.\, \mathcal{D}(x = y)$.

Proof Fact 9.3.1. ■

Exercise 14.1.3 Show the constructor laws and discreteness using the eliminator and without using matches.

Exercise 14.1.4 (Double induction) Prove the following double induction principle for numbers (from Smullyan and Fitting [18]):

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

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 14.5.14.

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.

14.2 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 14.2.1 $(x + y) + z = x + (y + z)$ 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 14.2.2 (Injectivity) $x + y = x + z \to y = z$ and $x + y = x \to y = 0$.

Proof Both claims follow by induction on x. ∎

Exercise 14.2.3 Prove $x \neq x + \mathsf{S}y$.

14.3 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 14.3.1 Prove that multiplication is commutative and associative. Also prove that multiplication distributes over addition: $x \cdot (y + z) = x \cdot y + x \cdot z$.

14.4 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 14.4.1

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. ■

14.5 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 14.5.1 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 14.5.2 (Decidability) $\mathcal{D}(x \le y)$.

Proof Immediate with Fact 14.1.2. ■

Fact 14.5.3 $x \le y \to x + (y - x) = y$.

Proof By induction on x with y quantified. The base case is immediate with (1) of Fact 14.4.1. 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 14.5.4 $x \leq y \;\longleftrightarrow\; \exists k.\, x + k = y$.

Proof Direction $\rightarrow$ follows with Fact 14.5.3, direction $\leftarrow$ follows with Fact 14.4.1 (2). ■

Fact 14.5.5

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

Proof Claim 1 follows with Fact 14.4.1 (2). Claim 2 follows from (1). Claim 3 follows with Fact 14.4.1 (4). Claim 4 follows by case analysis on x and constructor disjointness.

Reflexivity follows with Fact 14.4.1 (3).

For transitivity, we assume $x + a = y$ and $y + b = z$ using Fact 14.5.4. 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 14.5.4. By (3) we have $a = 0$, and thus $x = y$. ■

Fact 14.5.6 (Strict transitivity)

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

Proof We show (1), (2) is similar. Using Fact 14.5.4, 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 14.5.5 (1). ■

Fact 14.5.7

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

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 14.4.1 (4). Claim 3 follows from (2). For Claim 4 we assume $x + a = y$ using Fact 14.5.4 and show $x \leq x + a + z$ using Fact 14.5.5 (1). Claim 5 follows from (4). ∎

Fact 14.5.8 $x - y \leq x$

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 14.5.5 (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 14.5.9 (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 14.5.6). ∎

We will not give examples for the use of complete induction here. Chapter 16 introduces a generalization of complete induction called size recursion that has everal important applications in this text.

Exercise 14.5.10 Prove $y > 0 \to y - \mathsf{S}x < y$.

Exercise 14.5.11 Prove $x + y \leq x + z \to y \leq z$.

Exercise 14.5.12 Define a boolean decider for $x \leq y$ and prove its correctness.

Exercise 14.5.13 Define a function $\forall xy.\ x \leq y \to \Sigma k.\ x + k = y$.

Exercise 14.5.14 Use the double induction operator from Exercise 14.1.4 to prove $\forall xy.\ (x \leq y) + (y < x)$. No further induction or lemma is necessary.

14.6 Trichotomy

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

Fact 14.6.1 (Trichotomy) $\forall xy^{\mathsf{N}}.\ (x < y) + (x = y) + (y < x)$.

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 14.5.1 is useful. ■

Fact 14.6.2 $x \le y \Leftrightarrow (x < y) + (x = y)$.

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 14.6.3 $(x \le y) + (y < x)$.

Proof By Facts 14.6.1 and 14.6.2. ■

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

Corollary 14.6.4 (Contraposition) $\neg(x > y) \to x \le y$.

Corollary 14.6.5 (Equality by Contradiction) $\neg(x < y) \to \neg(y < x) \to x = y$.

Proof Follows by contraposition and antisymmetry. ■

Fact 14.6.6 **Bounded quantification** preserves decidability:

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

Proof By induction on k and Fact 14.6.2. ■

Exercise 14.6.7 (Tightness) Prove $x \le y \le \mathsf{S}x \to x = y \lor y = \mathsf{S}x$.

Exercise 14.6.8 Formulate the contraposition corollaries as equivalences and prove them with the trichotomy operator.

Exercise 14.6.9 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.

14.7 Euclidean Division

The *Euclidean division theorem* says that for two numbers x and y there always exist unique numbers a and b such that $x = a \cdot \mathsf{S}y + b$ and $b \leq y$. We will construct a certifying function that given x and y computes a and b. We define

$$\delta xyab := x = a \cdot \mathsf{S}y + b \wedge b \leq y$$

Fact 14.7.1 (Σ-Totality) $\forall xy. \Sigma a \Sigma b.\, \delta xyab$.

Proof By induction on x with y fixed. In the base case $x = 0$ we choose $a = b = 0$. In the successor case we assume $x = a \cdot \mathsf{S}y + b \wedge b \leq y$ and show $\mathsf{S}x = a' \cdot \mathsf{S}y + b' \wedge b' \leq y$. If $b < y$, we choose $a' = a$ and $b' = \mathsf{S}y$. If $b = y$, we choose $a' = \mathsf{S}y$ and $b' = 0$. ■

Fact 14.7.1 gives us two functions $D, M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that

$$\forall xy.\, \delta xy(Dxy)(Mxy)$$

For instance, we have $D\,100\;3 = 25$ and $M\,100\;3 = 0$ by computational equality.

The algorithm underlying the proof of Fact 14.7.1 can be formulated explicitly with a function

$$\begin{aligned} &\Delta : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \times \mathsf{N} \\ &\Delta 0\, y := (0, 0) \\ &\Delta(\mathsf{S}x)y := \textsc{let}\ (a, b) := \Delta xy\ \textsc{in}\ \textsc{if}\ \ulcorner b = y \urcorner\ \textsc{then}\ (\mathsf{S}a, 0)\ \textsc{else}\ (a, \mathsf{S}b) \end{aligned}$$

Note the use of the **upper-corner notation** $\ulcorner b = y \urcorner$. It acts as a placeholder for an application an equality decider (boolean or informative). The use of the upper-corner notation is convenient since it saves us from naming the equality decider.

Fact 14.7.2 (Correctness) $\forall xy.\ \delta xy\,(\pi_1(\Delta xy))\,(\pi_2(\Delta xy))$.

Proof By induction on x with y fixed. The base case is straightforward. In the successor we assume $\Delta xy = (a, b)$. This gives us the inductive hypothesis $\delta xyab$. We now consider the two cases $b = y$ and $b \neq y$ and prove the claim $\delta(\mathsf{S}x)y\,(\pi_1(\Delta(\mathsf{S}x)y))\,(\pi_2(\Delta(\mathsf{S}x)y))$. ▪

Note that the proofs of Facts 14.7.1 and 14.7.2 are very similar. If you verify both proofs by hand, you will find the proof of Fact 14.7.1 simpler since it doesn't have to verify that Δ is doing the right thing. The common base for both proofs and the underlying algorithm is expressed by so-called recursion rules for δ appearing in the following fact.

Fact 14.7.3 (Recursion rules)

1. $\delta 0 y 0 0$.
2. $\delta xyab \to b \neq y \to \delta(\mathsf{S}x)ya(\mathsf{S}b)$.
3. $\delta xyab \to b = y \to \delta(\mathsf{S}x)y(\mathsf{S}a)0$.

If you have to reinvent the algorithm it seems best to start with the specification δ and the recursion rules. From this base the rest is easily derived.

Finally, we show the uniqueness of δ.

Fact 14.7.4 (Uniqueness) $\delta xyab \to \delta xya'b' \to a = a' \wedge b = b'$.

Proof Let $a \cdot \mathsf{S}y + b = a' \cdot \mathsf{S}y + b'$ and $b, b' \leq y$. Once we know $a = a'$, $b = b'$ follows by injectivity of addition (Fact 14.2.2). We show $a = a'$ by trichotomy. Without loss of generality we assume $a < a'$ and derive a contradiction. We have $a' = a + 1 + k$. Thus

$$a \cdot \mathsf{S}y + b = a' \cdot \mathsf{S}y + b' = a \cdot \mathsf{S}y + \mathsf{S}y + k \cdot \mathsf{S}y + b'$$

Thus $b = \mathsf{S}y + k \cdot \mathsf{S}y + b'$ by injectivity of addition (Fact 14.2.2). Contradiction with $b \leq y$. ▪

Doing the uniqueness proof formally starting from first principles is amazingly involved. Using Coq's automation tactics lia and nia, reading the first line of the proof outline suffices for constructing a formal proof.

Exercise 14.7.5 Prove $\forall nx.\ \mathcal{D}(n \mid x)$.

Exercise 14.7.6 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 14.7.7 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 14.7.8 Show that the functiond $Dxy := \pi_1(\Delta xy)$ and $Mxy := \pi_2(\Delta xy)$ satisfy the following equations:

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

14.8 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 [15] classical development from 1929.

15 Least Witnesses

The set-theoretic development of numbers comes with the prominent result that every nonempty set of numbers contains a least element. We will prove the type-theoretic version of this result using excluded middle. Moreover, we will show that a satisfiable predicate on numbers has a least witness if and only if the law of excluded middle holds. We will contrast this result with the construction of a computational least witness operator that, given a decidable predicate on numbers and some witness, yields the least witness.

15.1 Least Witness Predicate

In this chapter, p will denote a predicate $\mathsf{N} \to \mathbb{P}$ and n and k will denote numbers. We say that n is a **witness of p** if pn, and that **p is satisfiable** if $\exists x.px$. We define a **least witness predicate** as follows:

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

Fact 15.1.1

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 14.6.4) 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 $\neg pn$. ■

Fact 15.1.1 justifies a least witness procedure known as **linear search**, which tests pk for $k = 0, 1, 2, \ldots$ until the first k satisfying p is found. While this procedure can be realized easily in a procedural programming language, it is not an option in computational type theory where recursion must be structural.

Exercise 15.1.2 Prove the following equivalence:
$(x = a \cdot \mathsf{S}y + b \wedge b \leq y) \longleftrightarrow (\mathsf{least}\,(\lambda a.\,\mathsf{S}a \cdot \mathsf{S}y > x)\,a \wedge b = x - a \cdot \mathsf{S}y)$.

15.2 Least Witness Operator

It turns out that it is straightforward to construct a **least witness operator (LWO)**

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

using structural recursion. The trick is to rely on informative types and a clever worker function.

Lemma 15.2.1 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to \forall n.\ \mathsf{safe}\,pn + \Sigma k.\ \mathsf{least}\,pk.$

Proof By induction on n. The base case is obvious by Fact 15.1.1 (2). In the successor case we do case analysis on the inductive hypothesis. In the nontrivial case we have $\mathsf{safe}\,pn$ and do case analysis on pn. If we have pn, we have $\mathsf{least}\,pn$. If we have $\neg pn$, we have $\mathsf{safe}\,p(\mathsf{S}n)$ by Fact 15.1.1 (3). In both cases the claim $\mathsf{safe}\,p(\mathsf{S}n) + \Sigma k.\ \mathsf{least}\,pk$ follows. ■

Fact 15.2.2 (Least witness operator)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to (\Sigma n.\ pn) \to \Sigma k.\ \mathsf{least}\,pk.$

Proof Assume pn. By Lemma 15.2.1 we have either the claim or obtain the claim with pn and $\mathsf{safe}\,pn$. ■

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

Corollary 15.2.3 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to (\exists n.pn) \to \exists k.\ \mathsf{least}\,pk.$

Proof Given that we have to construct a proof, we can assume pn. This gives us $\Sigma n.pn$ and thus we can obtain a least witness with Fact 15.2.2. ■

Corollary 15.2.4 (Decidability)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\,\mathcal{D}(pn)) \to \forall n.\,\mathcal{D}(\mathsf{least}\,pn).$

Proof We show $\mathcal{D}(\mathsf{least}\,pn)$. If $\neg pn$, we have $\neg\mathsf{least}\,pn$. Otherwise we assume pn. Thus $\mathsf{least}\,pk$ for some k by Fact 15.2.2. 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 15.1.1). ■

Exercise 15.2.5 Prove that $x - y$ is the least z such that $x \le y + z$:
$x - y = z \;\longleftrightarrow\; \mathsf{least}\,(\lambda z.\,x \le y + z)\,z.$

15.3 Purely Computational Version

The reason the construction of a least witness operator in the previous section is so elegant is the combination of computational structure and correctness arguments made possible by the use of informative types. We will now consider a purely computational version

$$L : (\mathsf{N} \to \mathsf{B}) \to \mathsf{N} \to \mathsf{N}$$

of a least witness operator satisfying

$$\forall f^{\mathsf{N}\to\mathsf{B}}\, \forall n^{\mathsf{N}}.\ fn = \mathbf{T} \to \mathsf{least}\,(\lambda k.fk = \mathbf{T})\ (Lfn)$$

which is obtained from the informative operator by extracting the computational structure and deleting all propositional correctness information. We start with a worker function

$$\begin{aligned} L' :\ & (\mathsf{N} \to \mathsf{B}) \to \mathsf{N} \to \mathcal{O}(\mathsf{N}) \\ L'f\,0 \ :=\ & \emptyset \\ L'f\,(\mathsf{S}n) \ :=\ & \textsc{match}\ L'fn\ [\,{}^\circ x \Rightarrow {}^\circ x \mid \emptyset \Rightarrow \textsc{if}\ fn\ \textsc{then}\ {}^\circ n\ \textsc{else}\ \emptyset\ ] \end{aligned}$$

using an option type as result type and define the main function as follows:

$$\begin{aligned} L :\ & (\mathsf{N} \to \mathsf{B}) \to \mathsf{N} \to \mathsf{N} \\ Lfn \ :=\ & \textsc{match}\ L'fn\ [\,{}^\circ x \Rightarrow x \mid \emptyset \Rightarrow n\,] \end{aligned}$$

Check that L' and L in fact represent the computational information in Lemma 15.2.1 and Fact 15.2.2. Next we verify the correctness of L and L'.

Fact 15.3.1 (Correctness)

1. $\textsc{match}\ L'fn\ [\,{}^\circ x \Rightarrow \mathsf{least}\,(\lambda n.fn = \mathbf{T})\,x \mid \emptyset \Rightarrow \mathsf{safe}\,(\lambda k.fk = \mathbf{T})\,n\,]$
2. $fn = \mathbf{T} \to \mathsf{least}\,(\lambda k.fk = \mathbf{T})\ (Lfn)$

Proof Claim (1) follows by induction on n. Claim (2) follows with claim (1). ∎

15.4 Linear Search Version

With a standard trick it is possible to write a least witness operator performing a linear search. Recall that a linear search tests pk for $k = 0, 1, 2, \dots$ until the first k satisfying p is found. In computational type theory, we need a bound for the linear search, which can be provided by a witness. Given a witness n, we will recurse on n and increment k with each recursion step until we reach the first k with pk. If we reach $n = 0$, we return k without testing p. We start with an informative version of the linear search approach.

Lemma 15.4.1 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\mathcal{D}(pn)) \to \forall nk.\ p(n+k) \to \mathsf{safe}\, pk \to \Sigma x.\ \mathsf{least}\, px.$

Proof By induction on n with k quantified. For $n = 0$ the claim is trivial. For the successor case, we assume $p(\mathsf{S}n + k)$ and $\mathsf{safe}\, pk$, and show $\Sigma x.\ \mathsf{least}\, px$. We do case analysis on $\mathcal{D}(pk)$. If pk, we have $\mathsf{least}\, pk$, and thus the claim. Otherwise we have $\neg pk$. By the inductive hypothesis it suffices to show $p(n+\mathsf{S}k)$ and $\mathsf{safe}\, p(\mathsf{S}k)$. The first claim is straightforward, and the second claim follows with Fact 15.1.1 (3). ■

Fact 15.4.2 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall n.\mathcal{D}(pn)) \to (\Sigma n.\ pn) \to \Sigma x.\ \mathsf{least}\, px.$

Proof Follows with Lemma 15.4.1 using $k = 0$. ■

As before we can switch to a purely computational version. We define the worker function as follows:

$$\begin{aligned} L :\ & (\mathsf{N}\to\mathsf{B})\to\mathsf{N}\to\mathsf{N}\to\mathsf{N} \\ L\,f\,0\,k \ &:=\ k \\ L\,f\,(\mathsf{S}n)\,k \ &:=\ \textsc{if}\ fk\ \textsc{then}\ k\ \textsc{else}\ L\,f\,n\,(\mathsf{S}k) \end{aligned}$$

Fact 15.4.3 (Correctness) Let $f^{\mathsf{N}\to\mathsf{B}}$. Then:

1. $\forall nk.\ f(n+k) = \mathbf{T} \to \mathsf{safe}\,(\lambda x.fx = \mathbf{T})\,k \to \mathsf{least}\,(\lambda x.fx = \mathbf{T})\,(Lnk)$
2. $\forall n.\ fn = \mathbf{T} \to \mathsf{least}\,(\lambda x.fx = \mathbf{T})\,(Ln0)$

Proof Claim (1) follows by induction on n with k quantified. Claim (2) follows with Claim (1). ■

15.5 Least Witnesses and Excluded Middle

If we switch to a propositional least witness operator using $\exists$ in place of Σ, *logical decidability* of p suffices.

Lemma 15.5.1 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall k. pk \vee \neg pk) \to \forall n.\ pn \to \exists k.\ \mathsf{least}\, pk.$

Proof By induction on n. Similar to Lemma 15.2.1. ■

We can now show that the law of excluded middle holds if and only if every satisfiable predicate on the numbers has a least witness.

Fact 15.5.2 $(\forall P^{\mathbb{P}}.\, P \vee \neg P) \ \longleftrightarrow\ (\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\exists n.pn) \to (\exists n.\ \mathsf{least}\, pn)).$

Proof Direction $\to$ follows with Lemma 15.5.1. For direction $\leftarrow$ we pick a proposition P and prove $P \vee \neg P$. We now obtain the least witness n of the satisfiable predicate $pn := \textsc{match}\ n\ [\,0 \Rightarrow P \mid \mathsf{S}\_ \Rightarrow \top\,]$. 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 $p0$. ■

15.6 Conclusion

In this chapter we have seen that the construction of a certifying function can be considerably simpler than the separate construction of a purely computational function and the necessary correctness proof. At the example of the least witness operator we have also seen that in computational type theory unbounded algorithms need to be revised such that they can be realized with structural recursion. Moreover, we have seen that informative types may suggest elegant certifying functions that differ computationally from the standard algorithm.

16 Size Recursion

Size recursion generalizes structural recursion such that recursion is possible for all smaller arguments, where smaller augments are determined by a numeric size function. In contrast to structural recursion, where the arguments must come from an inductive type, size recursion accommodates arguments from any type. Nevertheless, a general size recursion operator can be defined with structural recursion on numbers.

Using size recursion and informative types, functions can be defined following recursion schemes expressible with size recursion. Often it is convenient to accommodate the underlying recursion scheme with a specialized recursion operator incorporating the desired case analysis, and encapsulating the necessary termination proofs. As first examples we will consider Euclidean division, greatest common divisors, and Fibonacci numbers.

Size recursion provides us with a flexible induction principle for proofs. In fact, proofs by induction on the size of objects are frequently used in mathematical developments.

We will consider procedural specifications of functions and construct satisfying functions using step-indexing. Step-indexing applies whenever the recursion of the specification can be interpreted as size recursion.

16.1 Basic Size Recursion Operator

The basic intuition for defining a recursive procedure f says that fx can be computed using recursive applications fy for every y smaller than x. Similarly, when we prove px, we may assume a proof for py for every y smaller than x. Both ideas can be formalized with a **size recursion operator** of the type

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

The requirement that y be smaller than x for recursive applications is formalized with a **size function** σ and the premise $\sigma y < \sigma x$. 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**

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

Size recursion generalizes structural recursion on numbers:

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

While structural recursion is confined to numbers and provides recursion only for the predecessor of the argument, size recursion works on arbitrary types and provides recursion for every y smaller than x, not just the predecessor.

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 14.5.9).

It turns out that a size recursion operator can be defined with structural recursion on numbers, following the idea we have already seen for complete induction. Given the step function, we can define an auxiliary function

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

by structural recursion on the upper bound n. By using the auxiliary function with the upper bound $\mathsf{S}(\sigma x)$, we can then obtain the target function $\forall x. px$.

Lemma 16.1.1 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 function asserted by structural recursion on n:

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

We can phrase the above proof also as an informal inductive proof leaving implicit the operator R. While more verbose than the formal proof, the informal proof seems easier to read for humans. Here we go:

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

Theorem 16.1.2 (Size Recursion)

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

Proof Straightforward with Lemma 16.1.1. ■

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 will make sure that the type function p gives us all the information we need for proofs 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, making it possible to prove concrete equations by computational equality.

Exercise 16.1.3 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

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

using the size recursion operator.

Exercise 16.1.4 Let f be a function $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ satisfying the following equations:

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

Prove the following using size recursion:

a) $\forall xy.\ fxy \le y$

b) $\forall xy\,\Sigma k.\ \ x = k \cdot \mathsf{S}y + fxy$

16.2 Specialized Size Recursion Operators

Often it is helpful to define specialized size recursion operators. For instance, we may define a size recursion operator for binary type functions.

Fact 16.2.1 (Binary size recursion)

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

Proof Size recursion on $X \times Y$ using the type function $\lambda a.\, p(\pi_1 a)(\pi_2 a)$ and the size function $\lambda a.\, \sigma(\pi_1 a)(\pi_2 a)$. ∎

Euclidean division counts how often $\mathsf{S}y$ can be subtracted from x without truncation. The recursive scheme behind this procedural characterization of division can be formalized with a recursion operator.

Fact 16.2.2 (Euclidean recursion)

$$\begin{aligned}&\forall y^{\mathsf{N}}\ \forall p^{\mathsf{N}\to\mathbb{T}}.\\&(\forall x.\ x \le y \to px) \to\\&(\forall x.\ x > y \to p(x - \mathsf{S}y) \to px) \to\\&\forall x.\ px\end{aligned}$$

Proof Size recursion on N using the size function $\lambda x.x$. ∎

With Euclidean recursion it is easy to define a Euclidean division function.

Fact 16.2.3 (Euclidean Division) $\forall xy\, \Sigma k.\ k \cdot \mathsf{S}y \le x < \mathsf{S}k \cdot \mathsf{S}y$.

Proof We prove $\forall x\, \Sigma k.\ k \cdot \mathsf{S}y \le x < \mathsf{S}k \cdot \mathsf{S}y$ by Euclidean recursion. This gives us two proof obligations:

$$\begin{aligned}&\forall x.\ x \le y \to \Sigma k.\ k \cdot \mathsf{S}y \le x < \mathsf{S}k \cdot \mathsf{S}y\\&\forall x.\ x > y \to k \cdot \mathsf{S}y \le (x - \mathsf{S}y) < \mathsf{S}k \cdot \mathsf{S}y \to \Sigma k.\ k \cdot \mathsf{S}y \le x < \mathsf{S}k \cdot \mathsf{S}y\end{aligned}$$

The first obligation follows with $k := 0$, and the second obligation follows with $k := \mathsf{S}k$. ∎

Note that the function Div constructed for Fact 16.2.3 can be executed with a proof assistant. Thus the function

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

is a computational division function $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ and equations like $\mathsf{div}\ 133\ 12 = 11$ can be verified by computation.

Our next example concerns greatest common divisors (GCDs). The basic computation rules for GCDs may be formalized as

$\gamma 0 y y$	*zero rule*
$\gamma x y z \to \gamma y x z$	*symmetry rule*
$x \le y \to \gamma x (y - x) z \to \gamma x y z$	*subtraction rule*

where a proposition $\gamma x y z$ says that z is the GCD of x and y. There will be no need for a formal definition of γ. We formalize the recursion scheme for obtaining GCDs with the given computation rules with a specialized size recursion operator.

Fact 16.2.4 (GCD recursion)

$$\begin{aligned}
&\forall p^{\mathsf{N} \to \mathsf{N} \to \mathbb{T}}.\\
&(\forall y.\ p0y) \to\\
&(\forall xy.\ pxy \to pyx) \to\\
&(\forall xy.\ x \le y \to px(y-x) \to pxy) \to\\
&\forall xy.\ pxy
\end{aligned}$$

Proof By binary size recursion on $x + y$ considering four disjoint cases: $x = 0$, $y = 0$, $x \le y$, and $y < x$. ■

Chapter 24 gives a more comprehensive account of GCDs.

Exercise 16.2.5 Construct functions as follows using Euclidean recursion:

a) $\forall x y^{\mathsf{N}}\ \Sigma k.\ \ k \le y \land \exists n.\ x = n \cdot \mathsf{S}y + k.$

b) $\forall x y^{\mathsf{N}}\ \Sigma ab.\ \ x = a \cdot \mathsf{S}y + b \land b \le y.$

Exercise 16.2.6 Define a function $g : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ computing GCDs. Use GCD recursion and follow the computation rules for GCDs. For the function to compute correctly no formal specification of the concrete GCD predicate is needed. Use the type function $\lambda xy.\,\mathsf{N}$ for the GCD recursion. Verify $g\ 21\ 49 = 7$ by computational equality.

$$\begin{aligned}
&\Phi : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N}\\
&\quad \Phi\, f\, 0 \;:=\; 0\\
&\quad \Phi\, f\, 1 \;:=\; 1\\
&\Phi\, f\, (\mathsf{SS}n) \;:=\; f\,n + f(\mathsf{S}n)
\end{aligned}$$

Figure 16.1: Procedural specification of the Fibonacci function

16.3 Procedural Specifications and Step Indexing

A famous procedural specification is the specification of the Fibonacci function

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

This specification is usually taken as a definition in mathematical texts. In a type theory restricting primitive recursion to strict structural recursion, this specification cannot serve as a definition, however.

We will present two constructions of a Fibonacci function, one using step-indexing and one using an iterative formulation. Our main interest is in the construction using step-indexing since the step-indexed construction generalizes to all procedural specifications whose recursion can be interpreted as size recursion.

We first formalize the procedural specification of the Fibonacci function with the higher-order function Φ defined in Figure 16.1. The function Φ models the recursion in the specification of the Fibonacci function with a **continuation function** f taking care of the recursive applications. We say that a function **f satisfies Φ** if

$$\forall n.\ f n = \Phi f n$$

We now observe that f satisfies the **procedural specification** Φ if and only if f satisfies the three equations specifying the Fibonacci function. Procedural specifications with a continuation function enable us to formalize recursive specifications without recursion.

Next we define a specialized recursion operator modeling the recursive structure of the procedural specification.

$$
\begin{aligned}
&g : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\\
&\qquad\quad g\,0\,\_ \;:=\; 0\\
&\quad g\,(\mathsf{S}\_)\,0 \;:=\; 0\\
&\quad g\,(\mathsf{S}\_)\,1 \;:=\; 1\\
&g\,(\mathsf{S}k)(\mathsf{SS}n) \;:=\; gkn + gk(\mathsf{S}n)
\end{aligned}
$$

$$
\begin{aligned}
&h : \mathsf{N} \to \mathsf{N}\\
&hn \;:=\; g(\mathsf{S}n)n
\end{aligned}
$$

Figure 16.2: Step-indexed definition of a function satisfying Φ

Fact 16.3.1 (Fib recursion)

$$
\begin{aligned}
&\forall p^{\mathsf{N}\to\mathbb{T}}.\\
&p0 \to\\
&p1 \to\\
&(\forall n.\ pn \to p(\mathsf{S}n) \to p(\mathsf{SS}n)) \to\\
&\forall n.\ pn
\end{aligned}
$$

Proof By size recursion on n. ■

Fact 16.3.2 (Uniqueness) All functions satisfying Φ agree.

Proof Let f and g satisfy Φ. Then $\forall n.\ fn = gn$ follows by Fib induction. Straightforward. ■

Next we construct a function satisfying Φ. The trick is to define an auxiliary function with an extra argument called a **step index**. The step index serves as an upper bound for the remaining recursion steps. The recursion of the auxiliary function is on the step index which is initialized as $\mathsf{S}n$. The necessary definitions are shown in Figure 16.2.

We first show that the step index is handled correctly by g.

Lemma 16.3.3 (Index independence) Let $k, k' > n$. Then $gkn = gk'n$.

Proof We prove $\forall n. \forall kk' > n.\ gkn = gk'n$ by fib induction. Straightforward. ■

Fact 16.3.4 (Existence) h satisfies Φ.

Proof We prove $hn = \Phi hn$ by distinguishing the cases $n = 0$, $n = 1$, and $n = \mathsf{SS}n$. The first two cases are straightforward, and the third case

$$g(\mathsf{SS}n)n + g(\mathsf{SS}n)(\mathsf{S}n) = g(\mathsf{S}n)n + g(\mathsf{SS}n)(\mathsf{S}n)$$

follows with Lemma 16.3.3. ∎

Exercise 16.3.5 (Iterative definition of a Fibonacci function) There is a different definition of a Fibonacci function using the auxiliary function

$$\begin{aligned} g &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ gab0 &:= a \\ gab(\mathsf{S}n) &:= gb(a+b)n \end{aligned}$$

The underlying idea is to start with the first two Fibonacci numbers and then iterate n-times to obtain the n-th Fibonacci number. For instance,

$$g\,0\,1\,5 = g\,1\,1\,4 = g\,1\,2\,3 = g\,2\,3\,2 = g\,3\,5\,1 = g\,5\,8\,0 = 5$$

a) Prove $gab(\mathsf{SS}n) = gabn + gab(\mathsf{S}n)$ by induction on n.

b) Prove that $g01$ satisfies Φ.

Exercise 16.3.6 (Greatest common divisors)
Define a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfying the equations

$$\begin{aligned} f0y &= y \\ f(\mathsf{S}x)0 &= \mathsf{S}x \\ f(\mathsf{S}x)(\mathsf{S}y) &= \begin{cases} f(\mathsf{S}x)(y-x) & \text{if } x \le y \\ f(x-y)(\mathsf{S}y) & \text{if } x > y \end{cases} \end{aligned}$$

using step-indexing.

a) Formalize the specification as a function Γ.

b) Define a recursion operator modeling the recursion of the specification.

c) Show that Γ is unique.

d) Define a function satisfying Γ using step-indexing.

e) Show index independence for the step-indexed auxiliary function.

f) Verify that your function satisfies Γ.

g) Convince yourself with examples that every function satisfying Γ computes greatest common divisors. A formal proof appears in § 24.6.

Exercise 16.3.7 (Greatest common divisors with remainders)

a) Construct functions r and g of type $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ such that:

$$rxy = \begin{cases} x & \text{if } x \le y \\ r(x - \mathsf{S}y)y & \text{if } x > y \end{cases}$$
$$gx0 = x$$
$$gx(\mathsf{S}y) = g(\mathsf{S}y)(rxy)$$

b) Show that both specifications are unique.

c) Convince yourself with examples that the specified functions compute remainders and greatest common divisors. Formal proofs appear in § 24.5.

16.4 Notes

The DNF solver appearing in § 31.4 is an interesting example for the use of a specialized size recursion operator (DNF recursion § 31.5) where the arguments are lists rather than numbers.

There are two main scenarios for the use of size recursion. In the more pleasant scenario (e.g., Euclidean division, GCDs, DNF solver), the desired function can be specified with an informative type and can be constructed with a specialized size recursion operator. No separate correctness proofs are needed in this case. In the less pleasant scenario (e.g., Fibonacci numbers), only a procedural specification of the function is available. In this case, a step-indexed construction of the function is needed. To verify the step-indexed construction (index independence), the corresponding size recursion operator is useful.

Given a procedural specification of a function, the corresponding size recursion operator may be seen as a termination proof for the specified procedure. The termination proof makes it possible to realize the procedure as a function following the recursive structure of the specification. With step-indexing it is routine to verify that the constructed function satisfies the specifying equations. While a construction with a specialized size recursion operator is also possible and in fact is less redundant, it does not provide for the verification of the constructed function. To attempt such a verification, the abstraction barrier provided by the recursion operator would have to be lifted so that the defining equations of the recursion operator can be used in inductive correctness proofs. In a later chapter (Chapter 20), we will consider a generalization of size recursion called *wellfounded recursion* where the recursion operator comes with an unfolding equation providing for the verification of the specifying equations at the level of the operator.

17 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.

17.1 Inductive Definition

A list represents a finite sequence $[x_1, \dots, x_n]$ of values. Formally, lists are obtained with two constructors **nil** and **cons**:

$$\begin{aligned} [] &\mapsto \mathsf{nil} \\ [x] &\mapsto \mathsf{cons}\ x\ \mathsf{nil} \\ [x, y] &\mapsto \mathsf{cons}\ x\ (\mathsf{cons}\ y\ \mathsf{nil}) \\ [x, y, z] &\mapsto \mathsf{cons}\ x\ (\mathsf{cons}\ y\ (\mathsf{cons}\ z\ \mathsf{nil})) \end{aligned}$$

The constructor nil provides the **empty list**. The constructor cons yields for a value x and a list $[x_1, \dots, x_n]$ the list $[x, x_1, \dots, x_n]$. Given a list cons x A, we call x the **head** and A the **tail** of the list. Given a list $[x_1, \dots, x_n]$, we call n the **length** of the list and $x_1, \dots, x_n$ the **elements** of the list. An element may appear more than once in a list. For instance, $[2, 2, 3]$ is a list of length 3 that has 2 elements.

Formally, lists are accommodated with an inductive type definition

$$\mathcal{L}(X : \mathbb{T}) : \mathbb{T} \ ::= \ \mathsf{nil} \mid \mathsf{cons}\,(X, \mathcal{L}(X))$$

introducing three constructors:

$$\begin{aligned} \mathcal{L} &: \mathbb{T} \to \mathbb{T} \\ \mathsf{nil} &: \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \\ \mathsf{cons} &: \forall X^{\mathbb{T}}.\ X \to \mathcal{L}(X) \to \mathcal{L}(X) \end{aligned}$$

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:

$$\begin{aligned} \mathsf{E}_{\mathcal{L}} :\ & \forall X^{\mathbb{T}}\, p^{\mathcal{L}(X)\to\mathbb{T}}.\ p\,[\,] \to (\forall x A.\ pA \to p(x :: A)) \to \forall A.\, pA \\ \mathsf{E}_{\mathcal{L}}\, Xpaf\,[\,] &:= a \\ \mathsf{E}_{\mathcal{L}}\, Xpaf\,(x :: A) &:= fxA(\mathsf{E}_{\mathcal{L}}\, XpafA) \end{aligned}$$

The eliminator provides for inductive proofs, recursive function definitions, and structural case analysis.

Fact 17.1.1 (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 14.1.1). Claim (4) corresponds to $\mathsf{S}n \neq n$ and follows by induction on A with x quantified. ■

Fact 17.1.2 (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 17.1.1. In case both lists are nonempty with heads x and y, an additional case analysis on $x = y$ is needed. ■

Exercise 17.1.3 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ \mathcal{D}(A = [\,])$.

Exercise 17.1.4 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ (A = [\,]) + \Sigma x B.\ A = x :: B$.

17.2 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:

$$\begin{aligned} \mathsf{len}\,[x_1,\dots,x_n] &= n && \textbf{length} \\ [x_1,\dots,x_m] \mathbin{+\!\!+} [y_1,\dots,y_n] &= [x_1,\dots,x_m,y_1,\dots,y_n] && \textbf{concatenation} \\ f@[x_1,\dots,x_n] &= [f@x_1,\dots,f@x_n] && \textbf{map} \end{aligned}$$

Formally, we define the operations as recursive functions:

$$\begin{aligned} &\mathsf{len} : \ \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathsf{N} \\ &\quad \mathsf{len}\,[] := 0 \\ &\quad \mathsf{len}\,(x :: A) := S\,(\mathsf{len}\,A) \end{aligned}$$

$$\begin{aligned} &\mathbin{+\!\!+} : \ \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathcal{L}(X) \to \mathcal{L}(X) \\ &\quad [] \mathbin{+\!\!+} B := B \\ &\quad (x :: A) \mathbin{+\!\!+} B := x :: (A \mathbin{+\!\!+} B) \end{aligned}$$

$$\begin{aligned} &@ : \ \forall XY^{\mathbb{T}}.\ (X \to Y) \to \mathcal{L}(X) \to \mathcal{L}(Y) \\ &\quad f@[] := [] \\ &\quad f@(x :: A) := fx :: (f@A) \end{aligned}$$

Note that in all three definitions we accommodate X as an implicit argument for readability.

Fact 17.2.1

1. $A \mathbin{+\!\!+} (B \mathbin{+\!\!+} C) = (A \mathbin{+\!\!+} B) \mathbin{+\!\!+} C$ (associativity)
2. $A \mathbin{+\!\!+} [] = A$
3. $\mathsf{len}\,(A \mathbin{+\!\!+} B) = \mathsf{len}\,A + \mathsf{len}\,B$
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. ∎

17.3 Membership

Informally, we may characterize **membership** in lists with the equivalence

$$x \in [x_1, \dots, x_n] \longleftrightarrow x = x_1 \lor \cdots \lor x = x_n \lor \bot$$

Formally, we define the **membership predicate** by structural recursion on lists:

$$\begin{aligned} (\in) :\ & \forall X^{\mathbb{T}}.\, X \to \mathcal{L}(X) \to \mathbb{P} \\ (x \in [\,]) \ &:=\ \bot \\ (x \in y :: A) \ &:=\ (x = y \lor x \in A) \end{aligned}$$

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 17.3.1 (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 14.6.6). Similarly, quantification over the elements of a list preserves decidability.

Fact 17.3.2 (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. $(\forall x.\, \mathcal{D}(px)) \to \mathcal{D}(\exists x.\, x \in A \land px)$.
3. $(\forall x.\, \mathcal{D}(px)) \to (\Sigma x.\, x \in A \land px) + (\forall x.\, x \in A \to \neg px)$.

Proof By induction on A. ∎

Fact 17.3.3 (Membership laws)

1. $x \in A +\!\!+ B \ \longleftrightarrow\ x \in A \lor x \in B$.
2. $x \in f@A \ \longleftrightarrow\ \exists a.\, a \in A \land x = fa$.

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 17.3.4 (Explicit Characterization)
$x \in A \ \longleftrightarrow\ \exists A_1 A_2.\, A = A_1 +\!\!+ x :: A_2$.

Proof Direction $\to$ follows by induction on A. Direction $\leftarrow$ follows by induction on A_1. ∎

Fact 17.3.5 (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 17.3.6
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 17.3.7 (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.

17.4 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 17.4.1 List inclusion $A \subseteq B$ is reflexive and transitive. List equivalence $A \equiv B$ is reflexive, symmetric, and transitive.

Fact 17.4.2 We have the following properties for membership, inclusion, and equivalence of lists.

$$
\begin{array}{ll}
x \notin [] & x \in [y] \longleftrightarrow x = y \\
[] \subseteq A & A \subseteq [] \to A = [] \\
x \in y :: A \to x \neq y \to x \in A & x \notin y :: A \to x \neq y \land x \notin A \\
A \subseteq B \to x \in A \to x \in B & A \equiv B \to x \in A \longleftrightarrow x \in B \\
A \subseteq B \to x :: A \subseteq x :: B & A \equiv B \to x :: A \equiv x :: B \\
A \subseteq B \to A \subseteq x :: B & x :: A \subseteq B \longleftrightarrow x \in B \land A \subseteq B \\
x :: A \subseteq x :: B \to x \notin A \to A \subseteq B & x :: A \subseteq [y] \longleftrightarrow x = y \land A \subseteq [y] \\
x :: A \equiv x :: x :: A & x :: y :: A \equiv y :: x :: A \\
x \in A \to A \equiv x :: A & \\
x \in A + B \;\longleftrightarrow\; x \in A \lor x \in B & \\
A \subseteq A' \to B \subseteq B' \to A + B \subseteq A' + B' & A + B \subseteq C \;\longleftrightarrow\; A \subseteq C \land B \subseteq C
\end{array}
$$

Proof Except for the membership fact for concatenation, which already appeared as Fact 17.3.3, all claims have straightforward proofs not using induction. ▪

Fact 17.4.3 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 17.3.1) and bounded quantification preserves decidability (Fact 17.3.2). ▪

17.5 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 17.5.1 (Compatibility laws for propositional equivalence)
Let $P \longleftrightarrow P'$ and $Q \longleftrightarrow Q'$. Then:

$$
\begin{array}{ccc}
P \land Q \longleftrightarrow P' \land Q' & P \lor Q \longleftrightarrow P' \lor Q' & (P \to Q) \longleftrightarrow (P' \to Q') \\
\neg P \longleftrightarrow \neg P' & & (P \longleftrightarrow Q) \longleftrightarrow (P' \longleftrightarrow Q')
\end{array}
$$

Fact 17.5.2 (Compatibility laws for list equivalence)
Let $A \equiv A'$ and $B \equiv B'$. Then:

$$
\begin{array}{ccc}
x \in A \longleftrightarrow x \in A' & A \subseteq B \longleftrightarrow A' \subseteq B' & A \equiv B \longleftrightarrow A' \equiv B' \\
x :: A \equiv x :: A' & A + B \equiv A' + B' & f@A \equiv f@A'
\end{array}
$$

Coq's setoid rewriting facility makes it possible to use the rewriting tactic for rewriting with equivalences, provided the necessary compatibility laws and equivalence relations have been registered with the facility. The compatibility laws for propositional equivalence are preregistered.

Exercise 17.5.3 Which of the compatibility laws are needed to justify rewriting the claim $\neg(x \in y :: (f@A) + B)$ with the equivalence $A \equiv A'$?

17.6 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 $A\backslash x$ for **element removal** as follows:

$$\begin{aligned} \backslash :\ & \mathcal{L}(X) \to X \to \mathcal{L}(X) \\ [\,]\backslash\_ \ &:=\ [\,] \\ (x :: A)\backslash y \ &:=\ \text{IF } \ulcorner x = y \urcorner \text{ THEN } A\backslash y \text{ ELSE } x :: (A\backslash y) \end{aligned}$$

Fact 17.6.1

1. $x \in A\backslash y \;\longleftrightarrow\; x \in A \wedge x \neq y$
2. $\mathsf{len}\,(A\backslash x) \leq \mathsf{len}\,A$
3. $x \in A \to \mathsf{len}\,(A\backslash x) < \mathsf{len}\,A.$
4. $x \notin A \to A\backslash x = A$

Proof By induction on A. ■

Exercise 17.6.2 Prove $x \in A \;\to\; A \equiv x :: (A \setminus x)$.

Exercise 17.6.3 Prove the following equations, which are useful in proofs:

1. $(x :: A)\backslash x = A\backslash x$
2. $x \neq y \to (y :: A)\backslash x = y :: (A\backslash x)$

17.7 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 17.7.1 (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 17.3.4. ∎

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 17.7.2 (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 17.3.1) for the cons case. ∎

Corollary 17.7.3 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 17.7.4 (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 17.7.6). 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 17.7.5 (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 17.6.1 (3). ∎

Corollary 17.7.6 Let A and B be lists over a discrete type X. Then:

1. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,A \leq \mathsf{len}\,B$.
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. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \leq \mathsf{len}\,A \to \mathsf{nrep}\,B$.
5. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \leq \mathsf{len}\,A \to B \equiv A$.

Proof Interestingly, all claims follow without induction from Facts 17.7.5, 17.7.1, and 17.7.3.

For (1), assume $\mathsf{len}\,A > \mathsf{len}\,B$ and derive a contradiction with Fact 17.7.5.

Claims (2) and (3) follow from Claim (1), where for (3) we assume $\mathsf{nrep}\,A$ and derive a contradiction (justified by Corollary 17.7.3).

For (4), we assume $\mathsf{rep}\,B$ and derive a contradiction (justified by Corollary 17.7.3). By Fact 17.7.1, 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 17.7.5. ∎

We remark that Corollary 17.7.6 (3) may be understood as a pigeonhole lemma.

Exercise 17.7.7 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 17.7.8 (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 17.7.6 and Exercise 17.7.7.

Exercise 17.7.9 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 17.7.10 (Partition) The proof of Corollary 17.7.3 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 17.7.11 (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 17.7.12 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) $\mathsf{nrep}\,(\mathsf{seq}\,nk)$

Exercise 17.7.13 (List of numbers) Prove that every non-repeating list of numbers of length $\mathsf{S}n$ contains a number $k \ge n$. Hint: Use $\mathsf{seq}\,0\,n$ from Exercise 17.7.12 and Corollary 17.7.6 (1). First prove $\forall nA.\ (\Sigma k \in A.\ k \ge n) + \forall k \in A.\ k < n$.

Exercise 17.7.14 (List reversal)
Define a list reversal function $\mathsf{rev} : \mathcal{L}(X) \to \mathcal{L}(X)$ and prove the following:

a) $\mathsf{rev}(A + B) = \mathsf{rev}\,B + \mathsf{rev}\,A$

b) $\mathsf{rev}(\mathsf{rev}\,A) = A$

c) $x \in A \longleftrightarrow x \in \mathsf{rev}\,A$

d) $\mathsf{nrep}\,A \to x \notin A \to \mathsf{nrep}(A + [x])$

e) $\mathsf{nrep}\,A \to \mathsf{nrep}(\mathsf{rev}\,A)$

f) Reverse list induction: $\forall p^{X \to \mathbb{T}}.\ p[] \to (\forall xA.\ p(A) \to p(A + [x])) \to \forall A.\, pA.$
Hint: By (a) it suffices to prove $\forall A.\ p(\mathsf{rev}\,A)$, which follows by induction on A.

17.8 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 17.7.6 (3)), and every list is equivalent to a non-repeating list (Fact 17.7.4).

We assume that lists are taken over a discrete type X and define a **cardinality function** as follows:

$$\begin{aligned} \mathsf{card} :\ & \mathcal{L}(X) \to \mathsf{N} \\ \mathsf{card}\,[] \ &:=\ 0 \\ \mathsf{card}(x :: A) \ &:=\ \textsc{if}\ \ulcorner x \in A \urcorner\ \textsc{then}\ \mathsf{card}\,A\ \textsc{else}\ \mathsf{S}(\mathsf{card}\,A) \end{aligned}$$

Note that we write $\ulcorner x \in A \urcorner$ for the application of the membership decider provided by Fact 17.3.1. We prove that the cardinality function agrees with the cardinalities provided by equivalent nonrepeating lists.

Fact 17.8.1 (Cardinality)

1. $\forall A \Sigma B.\ B \equiv A \land \mathsf{nrep}\,B \land \mathsf{len}\,B = \mathsf{card}\,A$.
2. $\mathsf{card}\,A = n \ \longleftrightarrow\ \exists B.\ B \equiv A \land \mathsf{nrep}\,B \land \mathsf{len}\,B = n$.

Proof Claim 1 follows by induction on A. Claim 2 follows with Claim 1 and Corollary 17.7.6 (2). ▪

Corollary 17.8.2

1. $\mathsf{card}\,A \leq \mathsf{len}\,A$
2. $A \subseteq B \ \to\ \mathsf{card}\,A \leq \mathsf{card}\,B$
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 17.8.1, Corollary 17.7.6, and Corollary 17.7.3. ▪

Exercise 17.8.3 (Cardinality predicate) We define a recursive cardinality predicate:

$$\begin{aligned}
&\mathsf{Card} : \mathcal{L}(X) \to X \to \mathbb{P}\\
&\mathsf{Card}\,[\,]\,0 := \top\\
&\mathsf{Card}\,[\,]\,(\mathsf{S}n) := \bot\\
&\mathsf{Card}\,(x :: A)\,0 := \bot\\
&\mathsf{Card}\,(x :: A)\,(\mathsf{S}n) := \text{IF } \ulcorner x \in A \urcorner \text{ THEN } \mathsf{Card}\,A\,(\mathsf{S}n) \text{ ELSE } \mathsf{Card}\,A\,n
\end{aligned}$$

Prove that the cardinality predicate agrees with the cardinality function:
$\forall An.\ \mathsf{Card}\,An \longleftrightarrow \mathsf{card}\,A = n$.

Exercise 17.8.4 (Disjointness predicate) We define **disjointness** of lists as follows:

$$\mathsf{disjoint}\,A\,B := \neg\exists x.\ x \in A \land x \in B$$

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.

17.9 Position-Element Mappings

The positions of a list $[x_1, \dots, x_n]$ are the numbers $0, \dots, n-1$. More formally, a number n is a **position** of a list A if $n < \mathsf{len}\,A$. 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.

$$\begin{aligned} x \in A \to \mathsf{sub}\,y A\,(\mathsf{pos}\,Ax) &= x \\ \mathsf{nrep}\,A \to n < \mathsf{len}\,A \to \mathsf{pos}\,A\,(\mathsf{sub}\,yAn) &= n \end{aligned}$$

The function pos uses 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 29 for constructing injections and bijections on finite types.

Here are the definitions of pos and sub we will use:

$$\begin{aligned} &\mathsf{pos} : \mathcal{L}(X) \to X \to \mathsf{N} \\ &\qquad \mathsf{pos}\,[\,]\,x := 0 \\ &\qquad \mathsf{pos}\,(a :: A)\,x := \text{IF } \ulcorner a = x \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{pos}\,Ax) \end{aligned}$$

$$\begin{aligned} &\mathsf{sub} : X \to \mathcal{L}(X) \to \mathsf{N} \to X \\ &\qquad \mathsf{sub}\,y\,[\,]\,n := y \\ &\qquad \mathsf{sub}\,y\,(a :: A)\,0 := a \\ &\qquad \mathsf{sub}\,y\,(a :: A)\,(\mathsf{S}n) := \mathsf{sub}\,yAn \end{aligned}$$

Fact 17.9.1 Let A be a list over a discrete type. Then:

1. $x \in A \;\to\; \mathsf{sub}\,a\,A\,(\mathsf{pos}\,Ax) = x$
2. $x \in A \;\to\; \mathsf{pos}\,Ax < \mathsf{len}\,A$
3. $n < \mathsf{len}\,A \;\to\; \mathsf{sub}\,aA\,n \in A$
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 17.9.2 Prove $(\forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathsf{N} \to X) \to \bot$.

Exercise 17.9.3 Let A and B be lists over a discrete type X. Prove the following:

a) $x \in A \to \mathsf{pos}\, Ax = \mathsf{pos}\, (A +\!\!\!+ B)x$

b) $x \in A \to y \in A \to \mathsf{pos}\, Ax = \mathsf{pos}\, Ay \to x = y$

Exercise 17.9.4 One can realize pos and sub with option types

$$\begin{aligned}\mathsf{pos} &: \mathcal{L}(X) \to X \to \mathcal{O}(\mathsf{N})\\ \mathsf{sub} &: \mathcal{L}(X) \to \mathsf{N} \to \mathcal{O}(X)\end{aligned}$$

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) $x \in A \to \Sigma n.\ \mathsf{pos}\, Ax = {}^\circ n$

b) $n < \mathsf{len}\, A \to \Sigma x.\ \mathsf{sub}\, An = {}^\circ x$

c) $\mathsf{pos}\, Ax = {}^\circ n \to \mathsf{sub}\, An = {}^\circ x$

d) $\mathsf{nrep}\, A \to \mathsf{sub}\, An = {}^\circ x \to \mathsf{pos}\, Ax = {}^\circ n$

e) $\mathsf{sub}\, An = {}^\circ x \to x \in A$

f) $\mathsf{pos}\, Ax = {}^\circ n \to n < \mathsf{len}\, A$

18 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.

18.1 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

$$\mathsf{exp} : \mathbb{T} \ ::= \ \mathsf{con}(\mathsf{N}) \mid \mathsf{add}(\mathsf{exp}, \mathsf{exp}) \mid \mathsf{sub}(\mathsf{exp}, \mathsf{exp})$$

To 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:

$$\begin{aligned}
&E : \mathsf{exp} \to \mathsf{N} \\
&\qquad E\,x \ := \ x \\
&E\,(e_1 + e_2) \ := \ E\,e_1 + E\,e_2 \\
&E\,(e_1 - e_2) \ := \ E\,e_1 - E\,e_2
\end{aligned}$$

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 18.1.1 Do the reduction $E(3 + 5 - 2) \succ^* 6$ step by step (at the equational level).

Exercise 18.1.2 Prove some of the constructor laws for expressions. For instance, show that `con` is injective and that `add` and `sub` are disjoint.

Exercise 18.1.3 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.

18.2 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:

$$R : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N})$$

$$\begin{aligned} R\ []\ A &:= A \\ R\ (0 :: x :: C)\ A &:= R\ C\ (x :: A) \\ R\ (1 :: C)\ (x_1 :: x_2 :: A) &:= R\ C\ (x_1 + x_2 :: A) \\ R\ (2 :: C)\ (x_1 :: x_2 :: A) &:= R\ C\ (x_1 - x_2 :: A) \\ R\ \_\ \_ &:= [] \end{aligned}$$

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 empty stack.

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 18.2.1 Do the reduction $R[0,3,0,5,2][] \succ^* [2]$ step by step (at the equational level).

18.3 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:

$$\begin{aligned} &\gamma : \mathsf{exp} \to \mathcal{L}(\mathsf{N}) \\ \gamma x \ &:= \ [0, x] \\ \gamma(e_1 + e_2) \ &:= \ \gamma e_2 \mathbin{+\!\!+} \gamma e_1 \mathbin{+\!\!+} [1] \\ \gamma(e_1 - e_2) \ &:= \ \gamma e_2 \mathbin{+\!\!+} \gamma e_1 \mathbin{+\!\!+} [2] \end{aligned}$$

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 18.3.1 (Correctness) $R\ (\gamma e \mathbin{+\!\!+} C)\ A = R\ C\ (Ee :: A)$.

Proof By induction on e. The case for addition proceeds as follows:

$$\begin{array}{rlr}
& R\ (\gamma(e_1 + e_2) +\!\!+ C)\ A & \\
= & R\ (\gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [1] +\!\!+ C)\ A & \text{definition } \gamma \\
= & R\ (\gamma e_1 +\!\!+ [1] +\!\!+ C)\ (Ee_2 :: A) & \text{inductive hypothesis} \\
= & R\ ([1] +\!\!+ C)\ (Ee_1 :: Ee_2 :: A) & \text{inductive hypothesis} \\
= & R\ C\ ((Ee_1 + Ee_2) :: A) & \text{definition } R \\
= & R\ C\ (E(e_1 + e_2) :: A) & \text{definition } E
\end{array}$$

The equational reasoning implicitly employs conversion and associativity for concatenation $+\!\!+$. The full details can be explored with Coq. ■

Corollary 18.3.2 $R\ (\gamma e)\ [] = [Ee]$.

Proof Theorem 18.3.1 with $C = A = []$. ■

Exercise 18.3.3 Do the reduction $\gamma(5-2) \succ^* [0,3,0,5,2]$ step by step (at the equational level).

Exercise 18.3.4 Explore the proof of the correctness theorem starting from the proof script in the accompanying Coq development.

18.4 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.

$$\begin{array}{rcl}
\multicolumn{3}{l}{\delta : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{exp}) \to \mathcal{L}(\mathsf{exp})} \\
\delta\ []\ A & := & A \\
\delta\ (0 :: x :: C)\ A & := & \delta\ C\ (x :: A) \\
\delta\ (1 :: C)\ (e_1 :: e_2 :: A) & := & \delta\ C\ (e_1 + e_2 :: A) \\
\delta\ (2 :: C)\ (e_1 :: e_2 :: A) & := & \delta\ C\ (e_1 - e_2 :: A) \\
\delta\ \_\ \_ & := & []
\end{array}$$

The correctness theorem for decompilation follows closely the correctness theorem for compilation.

Theorem 18.4.1 (Correctness) $\delta\ (\gamma e \mathbin{+\!\!+} C)\ B = \delta\ C\ (e :: B)$.

Proof By induction on e. The case for addition proceeds as follows:

$$\begin{aligned}
& \delta\ (\gamma(e_1 + e_2) \mathbin{+\!\!+} C)\ B && \\
={}& \delta\ (\gamma e_2 \mathbin{+\!\!+} \gamma e_1 \mathbin{+\!\!+} [1] \mathbin{+\!\!+} C)\ B && \text{definition } \gamma \\
={}& \delta\ (\gamma e_1 \mathbin{+\!\!+} [1] \mathbin{+\!\!+} C)\ (e_2 :: B) && \text{inductive hypothesis} \\
={}& \delta\ ([1] \mathbin{+\!\!+} C)\ (e_1 :: e_2 :: B) && \text{inductive hypothesis} \\
={}& \delta\ C\ ((e_1 + e_2) :: B) && \text{definition } \delta
\end{aligned}$$

The equational reasoning implicitly employs conversion and associativity for concatenation $\mathbin{+\!\!+}$. ■

Corollary 18.4.2 $\delta\ (\gamma e)\ [] = [e]$.

18.5 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 [16] and Burstall [4]. 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 [5], 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.

19 Existential Witness Operators

In this chapter, we will define an existential witness operator that for decidable predicates on numbers obtains a satisfying number from a given 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 from a propositional satisfiability proof it obtains a witness that can be used computationally. Existential witness operators are required for various computational constructions.

Given that the elimination restriction disallows computational access to the witness of an existential proof, the definition of a witness operator is not obvious. In fact, our definition will rely on higher-order structural recursion, a feature of inductive definitions we have not used before. The key idea is the use linear search types

$$T(n : \mathsf{N}) : \mathbb{P} \ ::= \ C\,(\neg pn \to T(\mathsf{S}n))$$

featuring a recursion through the right-hand side of a function type. Derivations of a linear search type 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 computational proposition, we can construct a derivation of $T0$ in propositional mode using the witness from the proof of $\exists n.pn$.

19.1 Linear Search Types

Recall from § 4.3 that a computational proposition is an inductive proposition exempted from the elimination restriction. Proofs of computational propositions can be decomposed in computation mode although they have been constructed in proof mode. Recursive computational propositions thus provide for computational recursion.

We fix a predicate $p : \mathsf{N} \to \mathbb{P}$ and define **linear search types** 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 linear search types are declared as propositions. 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 also 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 19.1.1 (Strict positivity) Assume that the inductive type definition $B : \mathbb{T} ::= C(B \to \bot)$ is admitted although it violates the strict positivity condition. Give a proof of falsity. Hint: First define a function $F : B \to \bot$ inductively.

Exercise 19.1.2 Higher-order recursion offers yet another possibility for defining an empty type:

$$A : \mathbb{P} \;::=\; C(\top \to A)$$

Define a function $A \to \bot$.

19.2 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

$$\forall n.\ pn \to Tn \tag{19.1}$$
$$\forall n.\ T(\mathsf{S}n) \to Tn \tag{19.2}$$

Using recursion on n, function (19.2) yields a function

$$\forall n.\ Tn \to T0 \tag{19.3}$$

Using function (19.1), we have a function $V : \forall n.\ pn \to T0$ as required.

Theorem 19.2.1 (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 elimination restriction since $T0$ is a proposition. Now W' yields a computational witness for p. ∎

Exercise 19.2.2 Point out where in the defining equation of W' it is exploited that linear search types are computational (i.e., no elimination restriction).

Exercise 19.2.3 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 $\mathsf{S}n$.

19.3 More Existential Witness Operators

Fact 19.3.1 (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 15.2.2 that gives us a least witness for a witness, or Fact 15.2.4 and Corollary 15.2.3 that tell us that $\mathsf{least}\,p$ is a decidable and satisfiable predicate. ∎

Corollary 19.3.2 (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 19.3.3 (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 19.3.4 (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 19.3.5 (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 19.3.6 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 19.3.7 Let p be a decidable predicate on numbers. Define a function $\forall n.\ Tn \to \Sigma k.\ k \geq n \wedge pk$.

Exercise 19.3.8 Construct a witness operator $(\exists x.\ px) \to (\Sigma x.\ px)$ for decidable predicates p on booleans. Exploit that there are only two a priori known candidates for a witness. Note that a computation elimination for $\bot$ is needed.

19.4 Eliminator and Existential Characterization

We define an eliminator for linear search types:

$$\begin{aligned} E_T &: \forall q^{\mathsf{N}\to\mathbb{T}}.\ (\forall n.\ (\neg pn \to q(\mathsf{S}n)) \to qn) \to \forall n.\ Tn \to qn \\ E_T\, q\, f\, n (C\varphi) &:= f n (\lambda h.\ E_T\, q\, f (\mathsf{S}n)(\varphi h)) \end{aligned}$$

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 19.4.1 Define W' with the eliminator E_T for T.

Exercise 19.4.2 (Existential characterization) Prove the following facts about linear search types.

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 19.4.3 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.

19.5 Notes

With linear search types we have seen computational propositions that go far beyond the inductive definitions we have seen so far. The proof constructor of linear search types 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.

20 Well-Founded Recursion

Well-founded recursion is provided with an operator

$$\mathsf{wf}(R) \to \forall p^{X \to \mathbb{T}}.\, (\forall x.\, (\forall y.\, Ryx \to py) \to px) \to \forall x.\, px$$

generalizing arithmetic size recursion such that recursion can descend along any well-founded relation. In addition, the well-founded recursion operator comes with an *unfolding equation* making it possible to prove for the target function the equations used for the definition of the step function. Well-foundedness of relations is defined constructively with *recursion types*

$$\mathcal{A}_R(x : X) : \mathbb{P} \ ::= \ \mathsf{C}\,(\forall y.\, Ryx \to \mathcal{A}_R y)$$

obtaining well-founded recursion from the higher-order recursion coming with inductive types. Being defined as computational propositions, recursion types mediate between proofs and computational recursion.

The way computational type theory accommodates definitions and proofs by general well-founded recursion is one of the highlights of computational type theory.

20.1 Recursion Types

We assume a binary relation $R^{X \to X \to \mathbb{P}}$ and pronounce the Ryx as **y below x**. We define the **recursion types** for R as follows:

$$\mathcal{A}_R(x : X) : \mathbb{P} \ ::= \ \mathsf{C}\,(\forall y.\, Ryx \to \mathcal{A}_R y)$$

and call the elements of recursion types **recursion certificates**. Note that recursion types are computational propositions. A recursion certificate of type $\mathcal{A}_R(x)$ justifies all recursions starting from x and descending on the relation R. That a recursion on a certificate of type $\mathcal{A}_R(x)$ terminates is ensured by the built-in termination property of computational type theory. Note that recursion types realize higher-order recursion.

We will harvest the recursion provided by recursion certificates with a **recursion operator**

$$\begin{aligned} W' :\ & \forall p^{X \to \mathbb{T}}.\, (\forall x.\, (\forall y.\, Ryx \to py) \to px) \to \forall x.\, \mathcal{A}_R x \to px \\ W' pFx(\mathsf{C}\varphi) \ :=\ & Fx(\lambda yr.\, W' pFy(\varphi yr)) \end{aligned}$$

Computationally, W' may be seen as an operator that obtains a function

$$\forall x.\, \mathcal{A}_R x \to px$$

from a **step function**

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

The step function describes a function $\forall x.px$ obtained with a **continuation function**

$$\forall y.\, Ryx \to py$$

providing recursion for all y below x. We also speak of **recursion guarded by R**.

We define **well-founded relations** as follows:

$$\mathsf{wf}(R^{X \to X \to \mathbb{P}}) \;:=\; \forall x.\, \mathcal{A}_R(x)$$

Note that a proof of a proposition $\mathsf{wf}(R)$ is a function that yields a recursion certificate $\mathcal{A}_R(x)$ for every x of the base type of R. For well-founded relations, we can specialize the recursion operator W' as follows:

$$\begin{aligned} &W : \mathsf{wf}(R) \to \forall p^{X \to \mathbb{T}}.\, (\forall x.\, (\forall y.\, Ryx \to py) \to px) \to \forall x.\, px \\ &WhpFx \;:=\; W'pFx(hx) \end{aligned}$$

We will refer to W' and W as **well-founded recursion operators**. Moreover, we will speak of **well-founded induction** if a proof is obtained with an application of W' or W.

It will become clear that W generalizes the size recursion operator. For one thing we will show that the order predicate $<^{\mathsf{N} \to \mathsf{N} \to \mathsf{N}}$ is a well-founded relation. Moreover, we will show that well-founded relations can elegantly absorb size functions.

The inductive predicates $\mathcal{A}_R$ are often called **accessibility predicates**. They inductively identify the **accessible values** of a relation as those values x for which all values y below (i.e., Ryx) are accessible. To start with, all terminal values of R are accessible in R. We have the equivalence

$$\mathcal{A}_R(x) \;\longleftrightarrow\; (\forall y.\, Ryx \to \mathcal{A}_R(y))$$

Note that the equivalence is much weaker than the inductive definition in that it doesn't provide recursion and in that it doesn't force an inductive interpretation of the predicate $\mathcal{A}_R$ (e.g., the full predicate would satisfy the equivalence).

We speak of recursion types $\mathcal{A}_R(x)$ rather than accessibility propositions $\mathcal{A}_R(x)$ to emphasize that the propositional types $\mathcal{A}_R(x)$ support computational recursion.

Fact 20.1.1 (Extensionality) Let R and R' be relations $X \to X \to \mathbb{P}$.
Then $(\forall xy.\, R'xy \to Rxy) \to \forall x.\, \mathcal{A}_R(x) \to \mathcal{A}_{R'}(x)$.

Proof By well-founded induction with W'. ∎

Exercise 20.1.2 Prove $\mathcal{A}_R(x) \longleftrightarrow (\forall y.\, Ryx \to \mathcal{A}_R(y))$ from first principles. Make sure you understand both directions of the proof.

Exercise 20.1.3 Prove $\mathcal{A}_R(x) \to \neg Rxx$.
Hint: Use well-founded induction with W'.

Exercise 20.1.4 Prove $Rxy \to Ryx \to \neg\mathcal{A}_R(x)$.

Exercise 20.1.5 Show that well-founded relations disallow infinite descend:
$\mathcal{A}_R(x) \to px \to \neg\forall x.\, px \to \exists y.\, py \wedge Ryx$.

20.2 Well-founded Relations

Fact 20.2.1 The order relation on numbers is well-founded.

Proof We prove the more general claim $\forall nx.\, x < n \to \mathcal{A}_<(x)$ by induction on the upper bound n. For $n = 0$ the premise $x < n$ is contradictory. For the successor case we assume $x < \mathsf{S}n$ and prove $\mathcal{A}_<(x)$. By the single constructor for $\mathcal{A}$ we assume $y < x$ and prove $\mathcal{A}_<(x)$. Follows by the inductive hypothesis since $y < n$. ∎

Given two relations $R^{X\to X\to\mathbb{P}}$ and $S^{Y\to Y\to\mathbb{P}}$, we define the **lexical product** $R \times S$ as a binary relation $X \times Y \to X \times Y \to \mathbb{P}$:

$$R \times S \;:=\; \lambda(x',y')\,(x,y)^{X\times Y}.Rx'x \vee x' = x \wedge Sy'y$$

Fact 20.2.2 (Lexical products) $\mathsf{wf}(R) \to \mathsf{wf}(S) \to \mathsf{wf}(S \times R)$.

Proof We prove $\forall xy.\, \mathcal{A}_{R\times S}(x,y)$ by nested well-founded induction on first x in R and then y in S. By the constructor for $\mathcal{A}_{R\times S}(x,y)$ we assume $Rx'x \vee x' = x \wedge Sy'y$ and prove $\mathcal{A}_{R\times S}(x',y')$. If $Rx'x$, the claim follows by the inductive hypothesis for x. If $x' = x \wedge Sy'y$, the claim is $\mathcal{A}_{R\times S}(x,y')$ and follows by the inductive hypothesis for y. ∎

The above proof is completely straightforward when carried out formally with the well-founded recursion operator W.

Another important construction for binary relations are **retracts**. Here one has a relation $R^{Y\to Y\to\mathbb{P}}$ and uses a function $\sigma^{X\to Y}$ to obtain a relation R_σ on X:

$$R_\sigma \;:=\; \lambda x'x.\, R(\sigma x')(\sigma x)$$

We will show that retracts of well-founded relations are well-founded. It will also turn out that well-founded recursion on a retract R_σ is exactly well-founded size recursion on R with the size function σ.

Fact 20.2.3 (Retracts) $\mathsf{wf}(R) \to \mathsf{wf}(R_\sigma)$.

Proof Let $R^{Y \to Y \to \mathbb{P}}$ and $\sigma^{X \to Y}$. We assume $\mathsf{wf}(R)$. It suffices to show

$$\forall yx.\, \sigma x = y \to \mathcal{A}_{R_\sigma}(x)$$

We show the lemma by well-founded induction on y and R. We assume $\sigma x = y$ and show $\mathcal{A}_{R_\sigma}(x)$. Using the constructor for $\mathcal{A}_{R_\sigma}(x)$, we assume $R(\sigma x')(\sigma x)$ and show $\mathcal{A}_{R_\sigma}(x')$. Follows with the inductive hypothesis for $\sigma x'$. ■

Corollary 20.2.4 (Well-founded size recursion)
Let $R^{Y \to Y \to \mathbb{P}}$ be well-founded and $\sigma^{X \to Y}$. Then:
$\forall p^{X \to \mathbb{T}}.\ (\forall x.\ (\forall x'.\ R(\sigma x')(\sigma x) \to px') \to px) \to \forall x.\, px.$

We now obtain the arithmetic size recursion operator from § 16.1 as a special case of the well-founded size recursion operator.

Corollary 20.2.5 (Arithmetic size recursion)
$\forall \sigma^{X \to \mathsf{N}}\ \forall p^{X \to \mathbb{T}}.\ (\forall x.\ (\forall x'. \sigma x' < \sigma x \to px') \to px) \to \forall x.\, px.$

Proof Follows with Corollary 20.2.4 and Fact 20.2.1. ■

There is a story here. We came up with retracts to have an elegant construction of the wellfounded size recursion operator appearing in Corollary 20.2.4. Note that conversion plays an important role in type checking the construction. The proof that retracts of well-founded relations are well-founded (Fact 20.2.3) is interesting in that it first sets up an intermediate that can be shown with well-founded recursion. The equational premise $\sigma x = y$ of the intermediate claim is needed so that the well-founded recursion is fully informed. Similar constructions will appear once we look at inversion operators for indexed inductive types.

Exercise 20.2.6 Prove $R \subseteq R' \to \mathsf{wf}(R') \to \mathsf{wf}(R)$ for all relations $R, R' : X \to X \to \mathbb{P}$. Tip: Use extensionality (Fact 20.1.1).

Exercise 20.2.7 Give two proofs for $\mathsf{wf}(\lambda xy.\, \mathsf{S}x = y)$: A direct proof by structural induction on numbers, and a proof exploiting that $\lambda xy.\, \mathsf{S}x = y$ is a sub-relation of the order relation on numbers.

20.3 Unfolding Equation

Assuming FE, we can prove the equation

$$WFx = Fx(\lambda yr.\, WFy)$$

for the well-founded recursion operator W. We will refer to this equation as **unfolding equation**. The equation makes it possible to prove that the function WF satisfies the equations underlying the definition of the guarded step function F. This is a major improvement over arithmetic size recursion where no such tool is available. For instance, the unfolding equation gives us the equation

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

for an Euclidean division function D defined with well-founded recursion on $<_\mathsf{N}$:

$$Dxy \;:=\; W(Fy)x$$

$$F : \mathsf{N} \to \forall x.\, (\forall x'.\, x' < x \to \mathsf{N}) \to \mathsf{N}$$

$$Fyxh \;:=\; \begin{cases} 0 & \text{if } x \le y \\ \mathsf{S}(h(x - \mathsf{S}y)\ulcorner x - \mathsf{S}y < x\urcorner) & \text{if } x > y \end{cases}$$

Note that the second argument y is treated as a parameter. Also note that the equation for D is obtained from the unfolding equation for W by computational equality.

We now prove the unfolding equation using FE. We first show the remarkable fact that under FE all recursion certificates are equal.

Lemma 20.3.1 (Pureness of recursion types)
Under FE, all recursion types are pure: $\mathsf{FE} \to \forall x\, \forall ab^{\mathcal{A}_R(x)}.\, a = b$.

Proof We prove

$$\forall x\, \forall a^{\mathcal{A}_R(x)}\, \forall bc^{\mathcal{A}_R(x)}.\, b = c$$

using W'. This gives us the claim $\forall bc^{\mathcal{A}_R(x)}.\, b = c$ and the inductive hypothesis

$$\forall x'.\, Rx'x \to \forall bc^{\mathcal{A}_R(x')}.\, b = c$$

We destructure b and c, which gives us the claim

$$C\varphi = C\varphi'$$

for $\varphi, \varphi' : \forall x'.Rx'x \to \mathcal{A}_R(x')$. By FE it suffices to show

$$\varphi x'r = \varphi' x'r$$

for $r^{Rx'x}$. Holds by the inductive hypothesis. ■

Fact 20.3.2 (Unfolding equation)
Let $R^{X\to X\to\mathbb{P}}$, $p^{X\to\mathbb{T}}$, and $F^{\forall x.\,(\forall x'.\,Rx'x\to px')\to px}$.
Then $\mathsf{FE}\to\mathsf{wf}(R)\to\forall x.\,WFx = Fx(\lambda x'r.\,WFx')$.

Proof We prove $WFx = Fx(\lambda x'r.\,WFx')$. We have

$$WFx = W'Fxa = W'Fx(C\varphi) = Fx(\lambda x'r.\,W'Fx'(\varphi x'r))$$

for some a and φ. Using FE, it now suffices to prove the equation

$$W'Fx'(\varphi x'r) = W'Fx'b$$

for some b. Holds by Lemma 20.3.1. ∎

For functions $f^{\forall x.px}$ and $F^{\forall x.\,(\forall x'.\,Rx'x\to px')\to px}$ we define

$$f \vDash F \;:=\; \forall x.\,fx = Fx(\lambda yr.fy)$$

and say that f **satisfies** F. Given this notation, we may write

$$\mathsf{FE}\to\mathsf{wf}(R)\to WF\vDash F$$

for Fact 20.3.2. We now prove that all functions satisfying a step function agree if FE is assumed and R is well-founded.

Fact 20.3.3 (Uniqueness)
Let $R^{X\to X\to\mathbb{P}}$, $p^{X\to\mathbb{T}}$, and $F^{\forall x.\,(\forall x'.\,Rx'x\to px')\to px}$.
Then $\mathsf{FE}\to\mathsf{wf}(R)\to(f\vDash F)\to(f'\vDash F)\to\forall x.\,fx = f'x$.

Proof We prove $\forall x.\,fx = f'x$ using W with R. Using the assumptions for f and f', we reduce the claim to $Fx(\lambda x'r.fx') = Fx(\lambda x'r.f'x')$. Using FE, we reduce that claim to $Rx'x\to fx' = f'x'$, an instance of the inductive hypothesis. ∎

Exercise 20.3.4 Note that the proof of Lemma 20.3.1 doubles the quantification of a. Verify that this is justified by the general law $(\forall a.\forall a.pa)\to\forall a.pa$.

20.4 Example: GCDs

Our second example for the use of well-founded recursion and the unfolding equation is the construction of a function computing GCDs (Chapter 24). We start with the procedural specification in Figure 20.1. We will construct a function $g^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfying the specification using W on the retract of $<_{\mathsf{N}}$ for the size function

$$\sigma:\ \mathsf{N}\times\mathsf{N}\to\mathsf{N}$$
$$\sigma(x,y) := x+y$$

$$
\begin{aligned}
&g : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\\
&\qquad g\,0\,y \;=\; y\\
&\quad g\,(\mathsf{S}x)\,0 \;=\; \mathsf{S}x\\
&g\,(\mathsf{S}x)\,(\mathsf{S}y) \;=\; \begin{cases} g\,(\mathsf{S}x)\,(y-x) & \text{if } x \le y\\ g\,(x-y)\,(\mathsf{S}y) & \text{if } x > y\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\text{guard conditions}\\
&x \le y \;\to\; \mathsf{S}x + (y - x) < \mathsf{S}x + \mathsf{S}y\\
&x > y \;\to\; (x-y) + \mathsf{S}y < \mathsf{S}x + \mathsf{S}y
\end{aligned}
$$

Figure 20.1: Recursive specification of a gcd function

The figure gives the guard conditions for the recursive calls adding the preconditions established by the conditional in the third specifying equation.

Given the specification in Figure 20.1, the formal definition of the guarded step function is straightforward:

$$
\begin{aligned}
&F :\; \forall c^{\mathsf{N}\times\mathsf{N}}.\,(\forall c'.\,\sigma c' < \sigma c \to \mathsf{N}) \to \mathsf{N}\\
&\quad F\,(0,y)\;\_ \;:=\; y\\
&\quad F\,(\mathsf{S}x,0)\;\_ \;:=\; \mathsf{S}x\\
&F\,(\mathsf{S}x,\mathsf{S}y)\,h \;:=\; \begin{cases} h\,(\mathsf{S}x,\, y-x)\,\ulcorner \mathsf{S}x + (y-x) < \mathsf{S}x + \mathsf{S}y \urcorner & \text{if } x \le y\\ h\,(x-y,\, \mathsf{S}y)\,\ulcorner (x-y) + \mathsf{S}y < \mathsf{S}x + \mathsf{S}y \urcorner & \text{if } x > y\end{cases}
\end{aligned}
$$

We now define the desired function

$$
\begin{aligned}
&g : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\\
&gxy \;:=\; WHF(x,y)
\end{aligned}
$$

using the recursion operator W and the function

$$H :\; \forall c^{\mathsf{N}\times\mathsf{N}}.\,\mathcal{A}_{(<_{\mathsf{N}})_\sigma}(c)$$

obtained with the functions for recursion certificates for numbers (Fact 20.2.1) and retracts (Fact 20.2.3). Each of the three specifying equations in Figure 20.1 can now be obtained as an instance of the unfolding equation (Fact 20.3.2).

In summary, we note that the construction of a function computing GCDs with a well-founded recursion operator is routine given the standard constructions for retracts and the order on numbers. Proving that the specifying equations are satisfied is straightforward using the unfolding equation and FE.

That the example can be done so nicely with the general retract construction is due to the fact that type checking is modulo computational equality. For instance, the given type of the step function F is computationally equal to

$$\forall c^{\mathsf{N}\times\mathsf{N}}.\ (\forall c'.\ (<_{\mathsf{N}})_\sigma\ c'c \to \mathsf{N}) \to \mathsf{N}$$

Checking the conversions underlying our presentation is tedious if done by hand but completely automatic in Coq.

Exercise 20.4.1 Construct a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfying the Ackermann equations (§ 1.4) using well-founded recursion for the lexical product $<_{\mathsf{N}} \times <_{\mathsf{N}}$.

20.5 Unfolding Equation without FE

We have seen a proof of the unfolding equation assuming FE. Alternatively, one can prove the unfolding equation assuming that the step function has a particular extensionality property. For concrete step function one can usually prove that they have this extensionality without using assumptions.

We assume a relation $R^{X\to X\to\mathbb{P}}$, a type function $p^{X\to\mathbb{T}}$, and a step function

$$F : \forall x.\ (\forall x'.\ Rx'x \to px') \to px$$

We define **extensionality** of F as follows:

$$\mathsf{ext}(F)\ :=\ \forall xhh'.\ (\forall yr.\ hyr = h'yr) \to Fxh = Fxh'$$

The property says that Fxh remains the same if h is replaced with a function agreeing with h. We have $\mathsf{FE} \to \mathsf{ext}(F)$. Thus all proofs assuming $\mathsf{ext}(F)$ yields proofs for the stronger assumption FE.

Fact 20.5.1 $\mathsf{ext}(F) \to \forall xaa'.\ W'Fxa = W'Fxa'$.

Proof We assume $\mathsf{ext}(F)$ and show $\forall x \forall a^{\mathcal{A}_R(x)}.\ \forall aa'.\ W'Fxa = W'Fxa'$ using W'. This give us the inductive hypothesis

$$\forall y\ \forall r^{Ryx}\ \forall aa'.\ W'Fya = W'Fya'$$

By destructuring we obtain the claim $W'Fx(C\varphi) = W'Fx(C\varphi')$ for two functions $\varphi, \varphi' : \forall y. Ryx \to \mathcal{A}_R(y)$. By reducing W' we obtain the claim

$$Fx(\lambda yr.\ W'Fy(\varphi yr)) = Fx(\lambda yr.\ W'Fy(\varphi' yr))$$

By the extensionality of F we now obtain the claim

$$W'Fy(\varphi yr) = W'Fy(\varphi' yr)$$

for r^{Ryx}, which is an instance of the inductive hypothesis. ■

Fact 20.5.2 (Unfolding equation)
Let R be well-founded. Then $\mathtt{ext}(F) \to \forall x.\, WFx = Fx(\lambda yr.\, WFy)$.

Proof We assume $\mathtt{ext}(F)$ and prove $WFx = Fx(\lambda yr.\, WFy)$. We have $WFx = W'Fx(C\varphi) = Fx(\lambda yr.\, W'Fy(\varphi yr))$. Extensionality of F' now gives us the claim $W'Fy(\varphi yr) = W'Fy(\varphi' yr)$, which follows by Fact 20.5.1. ■

Exercise 20.5.3 From the definition of extensionality for step function it seams clear that ordinary step functions are extensional. To prove that an ordinary step function is extensional, no induction is needed. It suffices to walk through the matches and confront the recursive calls.

a) Prove that the step function for Euclidean division is extensional (§ 20.3).

b) Prove that the step function for GCDs is extensional (§ 20.4).

c) Prove that the step function for the Ackermann equations is extensional (Exercise 20.4.1).

Exercise 20.5.4 Show that all functions satisfying an extensional step function for a well-founded relation agree.

20.6 Witness Operator

There is an elegant and instructive construction of an existential witness operator (Chapter 19) using recursion types. We assume a decidable predicate $p^{\mathsf{N}\to\mathbb{P}}$ and define a relation

$$Rxy \;:=\; x = \mathsf{S}y \wedge \neg py$$

on numbers. We would expect that p is satisfiable if and only if $\mathcal{A}_R$ is satisfiable. And given a certificate $\mathcal{A}_R(x)$, we can compute a witness of p doing a linear search starting from x using well-founded recursion.

Lemma 20.6.1 $p(x+y) \to \mathcal{A}_R(y)$.

Proof Induction on x with y quantified. The base case follows by falsity elimination. For the successor case, we assume $H : p(\mathsf{S}x + y)$ and prove $\mathcal{A}_R(y)$. Using the constructor for $\mathcal{A}_R$, we assume $\neg py$ and prove $\mathcal{A}_R(\mathsf{S}y)$. By the inductive hypothesis it suffices to show $p(x + \mathsf{S}y)$. Holds by H. ■

Lemma 20.6.2 $\mathcal{A}_R(x) \to \mathsf{sig}(p)$.

Proof By well-founded induction with W'. Using the decider for p, we have two cases. If px, we have $\mathsf{sig}(p)$. If $\neg px$, we have $R(\mathsf{S}x)x$ and thus the claim holds by the inductive hypothesis. ■

Fact 20.6.3 (Existential witness operator)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall x.\ \mathcal{D}(px)) \to \mathsf{ex}(p) \to \mathsf{sig}(p).$

Proof We assume a decidable and satisfiable predicate $p^{\mathsf{N}\to\mathbb{P}}$ and define R as above. By Lemma 20.6.2 it suffices to show $\mathcal{A}_R(0)$. We can now obtain a witness x for p. The claim follows with Lemma 20.6.2. ∎

We may see the construction of an existential witness operator in Chapter 19 as a specialization of the construction shown here where the general recursion types used here are replaced with special purpose linear search types.

20.7 Equations Package and Extraction

The results presented so far are such that, given a recursive specification of a function, we can obtain a function satisfying the specification, provided we can supply a well-founded relation and proofs for the resulting guard conditions (see Figure 20.1 for an example). Moreover, if we don't accept FE as an assumption, we need to prove that the specified step function is extensional as defined in §20.5.

The proof assistant Coq comes with a tool named *Equations package* making it possible to write recursive specifications and associate them with well-founded relations. The tool then automatically generates the resulting proof obligations. Once the user has provided the requested proofs for the specification, a function is defined and proofs are generated that the function satisfies the specifying equations. This uses the well-founded recursion operator and the generic proofs of the unfolding equation we have seen. One useful feature of Equations is the fact that one can specify functions with several arguments and with size recursion. Equations then does the necessary pairing and the retract construction, relieving the user from tedious coding.

Taken together, we can now define recursive functions where the termination conditions are much relaxed compared to strict structural recursion. In contrast to functions specified with strict structural recursion, the specifying equations are satisfied as propositional equations rather than as computational equations. Nevertheless, if we apply functions defined with well-founded recursion to concrete and fully specified arguments, reduction is possible and we get the accompanying computational equalities (e.g., $\mathsf{gcd}\,21\,56 \approx 7$).

This is a good place to mention Coq's extraction tool. Given a function specified in computational type theory, one would expect that one can extract related programs for functional programming languages. In Coq, such an extraction tool is available for all function definitions, and works particularly well for functions defined with Equations. The vision here is that one specifies and verifies functions in computational type theory and then extracts programs that are correct by construction. A flagship project using extraction is CompCert (compcert.org) where a verified compiler for a subset of the C programming language has been developed.

20.8 Padding and Simplification

Given a certificate $a : \mathcal{A}_R(x)$, we can obtain a computationally equal certificate $b : \mathcal{A}_R(x)$ that exhibits any number of applications of the constructor for certificates:

$$\begin{aligned} a &\approx Cx(\lambda yr.\, a') \\ a &\approx Cx(\lambda yr. Cy(\lambda y'r'.\, a'')) \end{aligned}$$

We formulate the idea with two functions

$$\begin{aligned} &D : \ \forall x.\, \mathcal{A}_R(x) \to \forall y.\, Ryx \to \mathcal{A}_R(y) \\ &Dxa \ := \ \text{MATCH } a \ [\, C\varphi \Rightarrow \varphi \,] \end{aligned}$$

$$\begin{aligned} &P : \ \mathsf{N} \to \forall x.\, \mathcal{A}_R(x) \to \mathcal{A}_R(x) \\ &\quad P\,0xa \ := \ a \\ &P\,(\mathsf{S}n)a \ := \ Cx(\lambda yr.\, Pny(Dxayr)) \end{aligned}$$

and refer to P as **padding function**. We have, for instance,

$$\begin{aligned} P(1+n)xa &\approx Cx(\lambda y_1r_1.\, Pny_1(Dxay_1r_1)) \\ P(2+n)xa &\approx Cx(\lambda y_1r_1.\, Cy_1(\lambda y_2r_2.\, Pny_2(Dy_1(Dxay_1r_1)y_2r_2))) \end{aligned}$$

The construction appears tricky and fragile on paper. When carried out with a proof assistant, the construction is fairly straightforward: Type checking helps with the definitions of D and P, and simplification automatically obtains the right hand sides of the two examples from the left hand sides.

When we simplify a term $P(k+n)xa$ where k is a concrete number and n, x, and a are variables, we obtain a term that needs at least $2k$ additional variables to be written down. Thus the example tells us that simplification may have to introduce an unbounded number of fresh variables.

The possibility for padding functions seems to be a unique feature of higher-order recursion.

Exercise 20.8.1 Write a padding function for linear search types (§19.1)

20.9 Classical Well-foundedness

Well-founded relations and well-founded induction are basic notions in set-theoretic foundations. The standard definition of well-foundedness in set-theoretic foundations asserts that all non-empty sets have minimal elements. The set-theoretic definition is rather different the computational definition based on recursion types. We will show that the two definitions are equivalent under XM, where sets will be expressed as unary predicates.

A meeting point of the computational and the set-theoretic world is well-founded induction. In both worlds a relation is well-founded if and only if it supports well-founded induction.

Fact 20.9.1 (Characterization by well-founded induction)
$\forall R^{X \to X \to \mathbb{P}}.\ \mathsf{wf}(R) \longleftrightarrow \forall p^{X \to \mathbb{P}}.\ (\forall x.\ (\forall y.\ Ryx \to py) \to px) \to \forall x. px.$

Proof Direction $\to$ follows with W. For the other direction, we instantiate p with $\mathcal{A}_R$. It remains to show $\forall x.\ (\forall y.\ Ryx \to \mathcal{A}_R y) \to \mathcal{A}_R x$, which is an instance of the type of the constructor for $\mathcal{A}_R$. ■

The characterization of well-foundedness with the principle of well-founded induction is very interesting since no inductive types and only a predicate $p^{X \to \mathbb{P}}$ is used. Thus the computational aspects of well-founded recursion are invisible. They are added by the presence of the inductive predicate $\mathcal{A}_R$ admitting computational elimination.

Next we establish a positive characterization of the non-well-founded elements of a relation. We define **progressive predicates** and **progressive elements** for a relation $R^{X \to X \to \mathbb{P}}$ as follows:

$$\begin{aligned} \mathsf{pro}_R(p^{X \to \mathbb{P}}) &:= \forall x.\ px \to \exists y.\ py \land Ryx \\ \mathsf{pro}_R(x^X) &:= \exists p.\ px \land \mathsf{pro}_R(p) \end{aligned}$$

Intuitively, progressive elements for a relation R are elements that have an infinite descent in R. Progressive predicates are defined such that every witness has an infinite descent in R. Progressive predicates generalize the frequently used notion of infinite descending chains.

Fact 20.9.2 (Disjointness) $\forall x.\ \mathcal{A}_R(x) \to \mathsf{pro}_R(x) \to \bot.$

Proof By well-founded induction with W'. We assume a progressive predicate p with px and derive a contradiction. By destructuring we obtain y such that py and Ryx. Thus $\mathsf{pro}_R(y)$. The inductive hypothesis now gives us a contradiction. ■

Fact 20.9.3 (Exhaustiveness) $\mathsf{XM} \to \forall x.\ \mathcal{A}_R(x) \lor \mathsf{pro}_R(x).$

Proof Using XM, we assume $\neg\mathcal{A}_R(x)$ and show $\mathsf{pro}_R(x)$. It suffices to show $\mathsf{pro}_R(\lambda z.\neg\mathcal{A}_R(z))$. We assume $\neg\mathcal{A}_R(z)$ and prove $\exists y.\ \neg\mathcal{A}_R(y) \land Ryz$. Using XM, we assume $H : \neg\exists y.\ \neg\mathcal{A}_R(y) \land Ryz$ and derive a contradiction. It suffices to prove $\mathcal{A}_R(z)$. We assume $Rz'z$ and prove $\mathcal{A}_R(z')$. Follows with H and XM. ■

Fact 20.9.4 (Characterization by absence of progressive elements)
$\mathsf{XM} \to (\mathsf{wf}(R) \longleftrightarrow \neg\exists x.\ \mathsf{pro}_R(x))$.

Proof For direction $\to$ we assume $\mathsf{wf}(R)$ and $\mathsf{pro}_R(x)$ and derive a contradiction. We have $\mathcal{A}_R(x)$. Contradiction by Fact 20.9.2.

For direction $\leftarrow$ we assume $\neg\exists x.\ \mathsf{pro}_R(x)$ and prove $\mathcal{A}_R(x)$. By Fact 20.9.3 we assume $\mathsf{pro}_R(x)$ and have a contradiction with the assumption. ■

We define the **minimal elements** in $R^{X\to X\to\mathbb{P}}$ and $p^{X\to\mathbb{P}}$ as follows:

$$\mathrm{min}_{R,p}(x) \;:=\; px \land \forall y.\ py \to \neg Ryx$$

Using XM, we show that a predicate is progressive if and only if it has no minimal element.

Fact 20.9.5 $\mathsf{XM} \to (\mathsf{pro}_R(p) \longleftrightarrow \neg\exists x.\ \mathrm{min}_{R,p}(x))$.

Proof For direction $\to$, we derive a contradiction from the assumptions $\mathsf{pro}_R(p)$, px, and $\forall y.\ py \to \neg Ryx$. Straightforward.

For direction $\leftarrow$, using XM, we derive a contradiction from the assumptions $\neg\exists x.\ \mathrm{min}_{R,p}(x)$, px, and $H : \neg\exists y.\ py \land Ryx$. We show $\mathrm{min}_{R,p}(x)$. We assume py and Ryx and derive a contradiction. Straightforward with H. ■

Next we show that R has no progressive element if and only if every satisfiable predicate has a minimal witness.

Fact 20.9.6 $\mathsf{XM} \to (\neg(\exists x.\ \mathsf{pro}_R(x)) \longleftrightarrow \forall p.\ (\exists x.\ px) \to \exists x.\ \mathrm{min}_{R,p}(x))$.

Proof For direction $\to$, we use XM and derive a contradiction from the assumptions $\neg\exists x.\ \mathsf{pro}_R(x)$, px, and $\neg\exists x.\ \mathrm{min}_{R,p}(x)$. With Fact 20.9.5 we have $\mathsf{pro}_R(p)$. Contradiction with $\neg\exists x.\ \mathsf{pro}_R(x)$.

For direction $\leftarrow$, we assume px and $\mathsf{pro}_R(p)$ and derive a contradiction. Fact 20.9.5 gives us $\neg\exists x.\ \mathrm{min}_{R,p}(x)$. Contradiction with the primary assumption. ■

We now have that a relation R is well-founded if and only if every satisfiable predicate has a minimal witness in R.

Fact 20.9.7 (Characterization by existence of minimal elements)
$\mathsf{XM} \to (\mathsf{wf}(R) \longleftrightarrow \forall p^{X\to\mathbb{P}}.\ (\exists x.\ px) \to \exists x.\ \mathrm{min}_{R,p}(x))$.

Proof Facts 20.9.4 and 20.9.6. ■

The above proofs gives us ample opportunity to contemplate about the role of XM in proofs. An interesting example is Fact 20.9.3, where XM is used to show that an element is either well-founded or progressive.

20.10 Transitive Closure

The **transitive closure** R^+ of a relation $R^{X \to X \to \mathbb{P}}$ is the minimal transitive relation containing R. There are different possibilities for defining R^+. We choose an inductive definition based on two rules:

$$\frac{Rxy}{R^+xy} \qquad\qquad \frac{R^+xy' \qquad Ry'y}{R^+xy}$$

We work with this format since it facilitates proving that taking the transitive closure of a well-founded relation yields a well-founded relation. Note that the inductive predicate behind R^+ has four parameters X, R, x, y, where X, R, x are uniform and y is non-uniform.

Fact 20.10.1 Let $R^{X \to X \to \mathbb{P}}$. Then $\mathsf{wf}(R) \to \mathsf{wf}(R^+)$.

Proof We assume $\mathsf{wf}(R)$ and prove $\forall y.\ \mathcal{A}_{R^+}(y)$ by well-founded induction on y and R. This gives us the induction hypothesis and the claim $\mathcal{A}_{R^+}(y)$. Using the constructor for recursion types we assume R^+xy and show $\mathcal{A}_{R^+}(x)$. If R^+xy is obtained from Rxy, the claim follows with the inductive hypothesis. Otherwise we have R^+xy' and $Ry'y$. The inductive hypothesis gives us $\mathcal{A}_{R^+}(y')$. Thus $\mathcal{A}_{R^+}(x)$ since R^+xy'. ■

Exercise 20.10.2 Prove that R^+ is transitive.
Hint: Assume R^+xy and prove $\forall z.\ R^+yz \to R^+xz$ by induction on R^+yz. First formulate and prove the necessary induction principle for R^+.

20.11 Notes

The inductive definition of the well-founded points of a relation appears in Aczel [1] in a set-theoretic setting. Nordström [17] adapts Aczel's definition to a constructive type theory without propositions and advocates functions recursing on recursion types. Balaa and Bertot [3] define a well-founded recursion operator in Coq and prove that it satisfies the unfolding equation. They suggest that Coq should support the construction of functions with a tool taking care of the tedious routine proofs coming with well-founded recursion, anticipating Coq's current Equations package.

21 Numeral Types as Indexed Inductive Types

This chapter is our first encounter with indexed inductive types. The type constructors of indexed inductive types come with nonparametric arguments called indices, which can be freely instantiated by the accompanying value constructors. Indices provide for the definition of fine-grained type families with powerful type checking. We consider an indexed family of numeral types as lead example in this chapter. A numeral type $\mathcal{N}(n)$ has n elements, which are obtained with a successor constructor $\forall n.\ \mathcal{N}(n) \to \mathcal{N}(\mathsf{S}n)$ and a zero constructor $\forall n.\ \mathcal{N}(\mathsf{S}n)$.

Indexed inductive types come with the technical challenge that intuitively obvious discriminations must be realized with the plain discriminations provided by the type theory. While it is interesting to see that plain discriminations suffice, carrying out the often involved translations by hand is tedious.

21.1 Numeral Types

We define an indexed family of **numeral types** $\mathcal{N}(n)$ such that $\mathcal{N}(n)$ has exactly n elements called **numerals**:

$$\begin{aligned} &\mathcal{N} : \mathsf{N} \to \mathbb{T} \ ::= \\ &\mid \mathsf{Z} : \forall n.\ \mathcal{N}(\mathsf{S}n) \\ &\mid \mathsf{U} : \forall n.\ \mathcal{N}(n) \to \mathcal{N}(\mathsf{S}n) \end{aligned}$$

We may think of $\mathcal{N}(\mathsf{S}n)$ as a type containing numerals for the numbers $0, \dots, n$. For instance, the elements of $\mathcal{N}(4)$ are the numerals

$$\mathsf{Z}\,3,\ \ \mathsf{U}(\mathsf{Z}\,2),\ \ \mathsf{U}(\mathsf{U}(\mathsf{Z}\,1)),\ \ \mathsf{U}(\mathsf{U}(\mathsf{U}(\mathsf{Z}\,0)))$$

representing the numbers 0, 1, 2, 3. We omit the first argument of U since it is determined by the second argument. The constructor U takes the role of the successor constructor for numbers, with the difference that each application of $\mathsf{U} : \forall n.\ \mathcal{N}(n) \to \mathcal{N}(\mathsf{S}n)$ raises the **level** of the numeral type. The constructor $\mathsf{Z} : \forall n.\ \mathcal{N}(\mathsf{S}n)$ gives us for every n the numeral for zero at level $\mathsf{S}n$. More generally, $\mathsf{U}^k(\mathsf{Z}\,n)$ gives us the numeral for k at level $k + \mathsf{S}n$.

Things become interesting once we define functions that discriminate on numerals. We start with the most general such function, the **universal eliminator for numerals**. The universal eliminator constructs a function

$$\forall n\, \forall a^{\mathcal{N}(n)}.\ pna$$

by discriminating on the numeral a. This leads to two cases, one for each value constructor. For Z, we need a value $p(\mathsf{S}n)(\mathsf{Z}\,n)$, and for U we need a value $p(\mathsf{S}n)(\mathsf{U}\,na)$. In the case for U, we can use recursion on the component numeral a to obtain a value pna. We formalize this reasoning with the type of the universal eliminator:

$$\begin{aligned}
&\forall p^{\forall n.\ \mathcal{N}(n)\to\mathbb{T}}.\\
&(\forall n.\ p(\mathsf{S}n)(\mathsf{Z}\,n)) \to\\
&(\forall na.\ pna \to p(\mathsf{S}n)(\mathsf{U}\,na)) \to\\
&\forall na.\ pna
\end{aligned}$$

The defining equations for the universal eliminator can now be written as follows:

$$\begin{aligned}
E\,pe_1e_2\,\underline{\ }\,(\mathsf{Z}\,n) &:= e_1n && : p(\mathsf{S}n)(\mathsf{Z})\\
E\,pe_1e_2\,\underline{\ }\,(\mathsf{U}\,na) &:= e_2na(E\,pe_1e_2\,na) && : p(\mathsf{S}n)(\mathsf{U}\,na)
\end{aligned}$$

Note that the **index argument** n of E is specified with an underline in the patterns. This meets a general requirement on patterns and accounts for the fact that index arguments are determined by the discriminating argument (the index argument is determined as $\mathsf{S}n$ in both equations).

Exercise 21.1.1 (Recursive numerals) One can represent numeral types as iterated option types: $\mathcal{N}(n) \cong \mathcal{O}^n(\perp)$. Formally, we define a recursive family of numeral types:

$$\begin{aligned}
\mathcal{F} &: \mathsf{N}\to\mathbb{T}\\
\mathcal{F}(0) &:= \perp\\
\mathcal{F}(\mathsf{S}n) &:= \mathcal{O}(\mathcal{F}(n))
\end{aligned}$$

Define a bijection between $\mathcal{N}(n)$ and $\mathcal{F}(n)$ comprising two functions

$$\begin{aligned}
f &: \forall n.\ \mathcal{N}(n) \to \mathcal{F}(n)\\
g &: \forall n.\ \mathcal{F}(n) \to \mathcal{N}(n)
\end{aligned}$$

inverting each other. Define the bijection such that the constructor Z maps to $\emptyset$ and the constructor U maps to $^\circ$. Prove the two roundtrip equations. Hints:

a) Define f by recursion on numerals.

b) Define g by recursion on n.

c) Prove $gn(fna) = a$ by induction on the numeral a (universal eliminator).

d) Prove $fn(gnc) = c$ by induction on n.

21.2 Index Condition and Inversion Operator

There is a substantial condition on the types of inductive functions discriminating on indexed inductive types we call **index condition**. It says that the index arguments of the **discriminating type** (the type of the discriminating argument) must be given as unconstrained variables. You may check that this is the case for the type of the universal eliminator for numerals given above (there the variable in index position is n). We refer to variables in index positions of discriminating types as **index variables**.

Here are two functional types where the index condition disallows the numeral argument as discriminating argument:

$$\mathcal{N}(0) \to \bot \tag{21.1}$$

$$\forall n\, \forall a^{\mathcal{N}(\mathsf{S}n)}.\ (a = \mathsf{Z}\,n) + (\Sigma a'.\, a = \mathsf{U}\,na') \tag{21.2}$$

Given our informal model of numerals, we should be able to construct both functions. We realize this intuition by defining a more general function we call **inversion operator**. The type of the inversion operator

$$\begin{aligned}\forall n\, \forall a^{\mathcal{N}(n)}.\ &\text{MATCH } n \text{ RETURN } \mathcal{N}(n) \to \mathbb{T}\\ &[\,0 \Rightarrow \lambda a.\,\bot\\ &\mid \mathsf{S}n \Rightarrow \lambda a.\ (a = \mathsf{Z}\,n) + (\Sigma a'.\, a = \mathsf{U}\,na')\\ &]\,a\end{aligned}$$

is such that we can discriminate on the numeral argument. If we instantiate the type of the inversion operator with $n = 0$, we obtain the type (21.1) up to conversion, and if we instantiate it with $n = \mathsf{S}n$, we obtain the type (21.1) up to conversion. Since the numeral argument a of the inversion operator is unconstrained, we can define the inversion operator by discrimination on a, which, after conversion, yields the straightforward subgoals

$$(\mathsf{Z}\,n = \mathsf{Z}\,n) + (\Sigma a'.\, \mathsf{Z}\,n = \mathsf{U}\,na')$$

$$(\mathsf{U}\,na = \mathsf{Z}\,n) + (\Sigma a'.\, \mathsf{U}\,na = \mathsf{U}\,na')$$

The type of the inversion operator is written with what we call a **reloading match**. The reloading of the numeral argument is needed so that the branches of the match can type check.

Exercise 21.2.1 Prove the following:

a) $\mathcal{N}(0) \to \bot$

b) $\forall n\, \forall a^{\mathcal{N}(\mathsf{S}n)}.\ a \neq \mathsf{Z}\,n \to \Sigma a'.\, a = \mathsf{U}\,na'$

Check your proofs with the proof assistant. Don't use automation tactics. Keep in mind that the index condition disallows discriminations on constrained indexed types. Convince yourself that the universal eliminator applies to the claims, but doesn't lead to proofs since the instantiations of the index argument are lost.

Exercise 21.2.2 (Predecessor) Define a function $P : \forall n.\ \mathcal{N}(\mathsf{SS}n)) \to \mathcal{N}(\mathsf{S}n)$ inverting the successor constructor: $\forall n \forall a^{\mathcal{N}(\mathsf{S}n)}.\ Pn(\mathsf{U}a) = a$.

Exercise 21.2.3 (Embedding numerals into numbers)
Define an injective function $f : \forall n.\ \mathcal{N}(n) \to \mathsf{N}$ embedding numerals into numbers. Prove that f is injective: $\forall n \forall ab^{\mathcal{N}(n)}.\ fna = fnb \to a = b$.
Hint: Do the injectivity proof by induction on a (using the universal eliminator) followed by inversion of b (using the inversion operator).

Exercise 21.2.4 Write the defining equations for the inversion operator. Convince yourself that the reloading match cannot be avoided.

21.3 Constructor Laws and Equality Decider

Next we show the constructor laws for numeral types. *Constructor disjointness*

$$\forall n\, \forall a^{\mathcal{N}(n)}.\ \mathsf{Z}n \neq \mathsf{U}na$$

follows with feq and the function

$$\lambda na^{\mathcal{N}(n)}.\ \textsc{match}\ a\ [\, \mathsf{Z}\,\_ \Rightarrow \bot \mid \mathsf{U}\,\_\,\_ \Rightarrow \top\,]$$

Injectivity of the value constructor U

$$\forall n\, \forall ab^{\mathcal{N}(n)}.\ \mathsf{U}na = \mathsf{U}nb \to a = b$$

will follow with feq and a function

$$f : \forall n.\ \mathcal{N}(\mathsf{S}n) \to \mathcal{N}(n) \qquad \text{such that} \quad fn(\mathsf{U}na) \approx a$$

The construction of such a function is not straightforward and has stunned the author more than once. The trick is to first construct a function

$$f_1 : \forall n.\ \mathcal{N}(\mathsf{S}n) \to \mathcal{O}(\mathcal{N}(n)) \qquad \text{such that} \quad f_1 n(\mathsf{U}na) \approx {}^\circ a$$

and to exploit that we have an injectivity operator

$$\forall X\, \forall xy^X.\ {}^\circ x = {}^\circ y \to x = y$$

for option types. We now obtain f_1 from a more general function

$$\begin{aligned} f_2 :\ & \forall n.\ \mathcal{N}(n) \to \text{MATCH } n\ [\,0 \Rightarrow \bot \mid \mathsf{S}n' \Rightarrow \mathcal{O}(\mathcal{N}(n'))\,] \\ f_2\,\_(\mathsf{Z}\_) \ & := \ \emptyset \\ f_2\,\_\,(\mathsf{U}\,n\,a) \ & := \ {}^\circ a \end{aligned}$$

as the instance $f_1 n := f_2(\mathsf{S}n)$.

Next we construct an equality decider

$$\forall n\,\forall ab^{\mathcal{N}(n)}.\ \mathcal{D}(a = b)$$

for numeral types. As usual, the decider is obtained by recursion on a, subsequent inversion of b (using the inversion operator), and the constructor laws for numerals.

Exercise 21.3.1 Do all of the above constructions with the proof assistant not using automation tactics. The most delicate construction is the proof of the injectivity of U. Try to understand every detail.

Exercise 21.3.2 Figure 5.2 gives a slick proof of the injectivity of the successor constructor for numbers. The technique also applies to sum types and to option types. Find out why the technique does not apply to the constructor U for numerals.

Exercise 21.3.3 (Listing) Define a function $\forall n.\ \mathcal{L}(\mathcal{N}(n))$ that yields for every n a nonrepeating list of length n containing all elements of $\mathcal{N}(n)$. Hint: Use induction on n and the constructor laws for numerals to show that the list is non-repeating.

Exercise 21.3.4 (Lifting) Define a function $L : \forall n.\ \mathcal{N}(n) \to \mathcal{N}(\mathsf{S}n)$ lifting numerals to the next level. For instance, L should satisfy $L\,5\,(\mathsf{U}(\mathsf{U}(\mathsf{Z}\,2))) \approx \mathsf{U}(\mathsf{U}(\mathsf{Z}\,3))$. Prove that L is injective: $\forall n \forall ab^{\mathcal{N}(n)}.\ Lna = Lnb \to a = b$.
Hint: Do the injectivity proof by induction on a (using the universal eliminator) followed by inversion of b (using the inversion operator).

21.4 Embedding Numerals into Numbers

We define a function mapping numerals to the numbers they represent:

$$\begin{aligned} N :\ & \forall n.\ \mathcal{N}(n) \to \mathsf{N} \\ N\,\_\,(\mathsf{Z}\,n) \ & := \ 0 \\ N\,\_\,(\mathsf{U}\,n\,a) \ & := \ \mathsf{S}(Nna) \end{aligned}$$

We would like to show that Nn reaches exactly the numbers smaller than n. We first show

$$\forall n\,\forall a^{\mathcal{N}(n)}.\ Nna < n \tag{21.3}$$

which follows by induction on a (using the universal eliminator for numerals).

Next we define a function inverting N:

$$\begin{aligned} B :\ & \mathsf{N} \to \forall n.\ \mathcal{N}(\mathsf{S}n) \\ B\,0\,n \ &:=\ \mathsf{Z}\,n \\ B\,(\mathsf{S}k)\,0 \ &:=\ \mathsf{Z}\,0 \\ B\,(\mathsf{S}k)\,(\mathsf{S}n) \ &:=\ \mathsf{U}\,(\mathsf{S}n)\,(Bkn) \end{aligned}$$

The idea is that Bkn yields the numeral for k in $\mathcal{N}(n)$. If k is too large (i.e., $k > n$), Bkn yields the largest numeral in $\mathcal{N}(\mathsf{S}n)$.

We can now show two roundtrip properties:

$$\forall n\, \forall a^{\mathcal{N}(\mathsf{S}n)}.\ B(N(\mathsf{S}n)a)n = a \tag{21.4}$$

$$\forall kn.\ k \le n \to N(\mathsf{S}n)(Bkn) = k \tag{21.5}$$

Note that (21.4) yields the injectivity of N. Moreover, (21.5) together with (21.3) yields the surjectivity of $N(\mathsf{S}n)$ for $\{0, \dots, n\}$.

The second roundtrip property (21.5) follows by a straightforward induction on k.

The first roundtrip property (21.4) needs more effort. It cannot be shown directly by induction on a since the index argument in the type of a is instantiated. However, it can be shown by induction on n and inversion of a using the inversion operator.

22 Inductive Derivation Systems

Inductive relations are relations defined with derivation rules such that an instance of an inductive relation holds if it is derivable with the rules defining the relation. Inductive relations are an important mathematical device for setting up proof systems for logical systems and formal execution rules for programming languages. Inductive relations are also the basic tool for setting up type systems.

It turns out that inductive relations can be modeled elegantly with indexed inductive type definitions, where the type constructor represents the relation and the value constructors represent the derivation rules. We present inductive relations and their formalization as indexed inductive types by discussing examples.

22.1 Derivation System for Comparisons

Consider the following *derivation rules* for *comparisons* of numbers:

$$\frac{}{x \mathrel{\dot<} \mathsf{S}x} \qquad\qquad \frac{x \mathrel{\dot<} y \qquad y \mathrel{\dot<} z}{x \mathrel{\dot<} z}$$

We may verbalize the rules as saying:

1. Every number is smaller than its successor.
2. If x is smaller than y and y is smaller than z, then x is smaller than z.

We may now ask the following questions:

- *Soundness:* Does every derivable comparison $x \mathrel{\dot<} y$ satisfy $x < y$?
- *Completeness:* Is every comparison $x \mathrel{\dot<} y$ such that $x < y$ derivable?

The answer to both questions is yes. Soundness for all derivable comparisons follows from the fact that for each of the two rules the *conclusion* (comparison below the line) is valid if the *premises* (comparisons above the line) are valid. To argue completeness, we need a recursive procedure that for $x < y$ constructs a derivation of $x \mathrel{\dot<} y$ (recursion on y does the job).

Derivations of comparisons are obtained by combining rules. Here are two different derivations of the comparison $3 \dot{<} 6$:

$$\dfrac{\dfrac{}{3 \dot{<} 4} \qquad \dfrac{\dfrac{}{4 \dot{<} 5} \qquad \dfrac{}{5 \dot{<} 6}}{4 \dot{<} 6}}{3 \dot{<} 6} \qquad\qquad \dfrac{\dfrac{\dfrac{}{3 \dot{<} 4} \qquad \dfrac{}{4 \dot{<} 5}}{3 \dot{<} 5} \qquad \dfrac{}{5 \dot{<} 6}}{3 \dot{<} 6}$$

Every line in the derivations represents the application of one of the two derivation rules. Note that the leaves of the derivation tree are all justified by the first rule, and that the inner nodes of the derivation tree are all justified by the second rule.

It turns out that derivation systems can be represented formally as indexed type families. For the derivation system for comparisons we employ a type constructor

$$\mathsf{L} : \mathsf{N} \to \mathsf{N} \to \mathbb{T}$$

to model the comparisons and two value constructors

$$\begin{aligned} \mathsf{L}_1 &: \ \forall x.\, \mathsf{L}x(\mathsf{S}x) \\ \mathsf{L}_2 &: \ \forall xyz.\, \mathsf{L}xy \to \mathsf{L}yz \to \mathsf{L}xz \end{aligned}$$

to model the derivation rules. Modeling derivation systems as indexed type families is a wonderful thing since it clarifies the many things left open by the informal presentation and also yields a powerful formal framework for derivation systems in general. Note that derivations now appear as terms describing values of derivation types $\mathsf{L}xy$. Here are examples:

$$\begin{aligned} \mathsf{L}_1\,4 &: \ \mathsf{L}\,4\,5 \\ \mathsf{L}_2\,4\,5\,6\,(\mathsf{L}_1\,4), (\mathsf{L}_1\,5)) &: \ \mathsf{L}\,4\,6 \\ \mathsf{L}_2\,3\,4\,6\,(\mathsf{L}_1\,3)\,(\mathsf{L}_2\,4\,5\,6\,(\mathsf{L}_1\,4), (\mathsf{L}_1\,5))) &: \ \mathsf{L}\,3\,6 \end{aligned}$$

When we look at the types of the value constructors, we see that we can declare the first argument of L as a nonuniform parameter, and that we must declare the second argument of L as an index:

$$\begin{aligned} &\mathsf{L}\,(x : \mathsf{N}) : \ \mathsf{N} \to \mathbb{T} \ := \\ &\mid \mathsf{L}_1 : \ \mathsf{L}x(\mathsf{S}x) \\ &\mid \mathsf{L}_2 : \ \forall yz.\, \mathsf{L}xy \to \mathsf{L}yz \to \mathsf{L}xz \end{aligned}$$

In principle, we could also declare the first argument of L as an index, but declaring it as a parameter avoids the index condition for the first argument.

We can now do formal proofs concerning the derivation system L. We first prove **completeness**:

$$\forall xy.\, x < y \to \mathsf{L}xy \tag{22.1}$$

The proof succeeds by induction on y with x quantified. The base case is trivial. For the successor case, we assume $x < \mathsf{S}y$ and prove $\mathsf{L}x(\mathsf{S}y)$. If $x = y$, we obtain $\mathsf{L}x(\mathsf{S}y)$ with L_1. If $x \neq y$, we have $x < y$. Hence we have $\mathsf{L}xy$ by the inductive hypothesis. By $\mathsf{L}1$ we have $\mathsf{L}\,y(\mathsf{S}y)$. The claim $\mathsf{L}\,x(\mathsf{S}y)$ follows with L_2.

For the soundness proof we need **induction on derivations**. Formally, we provide this induction with the universal eliminator for L, which has the type

$$\begin{aligned}
&\forall p^{\forall xy.\,\mathsf{L}xy \to \mathbb{T}}.\\
&(\forall x.\, px(\mathsf{S}x)(\mathsf{L}_1 x)) \to\\
&(\forall xzyab.\, pxya \to pyzb \to pxz(\mathsf{L}_2\, xyzab)) \to\\
&\forall xya.\, pxya
\end{aligned}$$

The defining equations of the eliminator discriminate on a and are obvious. Note that the type function p takes the nonuniform parameter x as an argument. This is necessary so that the second inductive hypothesis of the second clause of the eliminator can be provided.

We now prove **soundness**

$$\forall xy.\, \mathsf{L}xy \to x < y \tag{22.2}$$

by induction on the derivation of $\mathsf{L}xy$. This give us the proof obligations

$$\begin{gathered}
x < \mathsf{S}y\\
x < y \to y < z \to x < z
\end{gathered}$$

which are both obvious. Not that the obligations are rewritings of the two derivations rules replacing $\dot{<}$ with $<$. Informally, we can do the soundness proof by just showing that each of the two derivation rules is sound for $<$. This applies in general.

Exercise 22.1.1 (Constructor laws)

a) Prove $\mathsf{L}_1 x \neq \mathsf{L}_2 xy(\mathsf{S}x)ab$.

b) Prove that there are two different derivations of $\mathsf{L}\,3\,6$ (i.e., values of the type $\mathsf{L}\,3\,6$). Hint: Use a function $\forall xy.\, \mathsf{L}xy \to \mathsf{N}$ returning the length of the leftmost path of a derivation tree.

c) Define an inversion operator for L having the type

$$\forall xy.\, \mathsf{L}xy \to (y = \mathsf{S}x) + \Sigma z.\, \mathsf{L}xz \times \mathsf{L}zy$$

d) *Challenge.* Prove injectivity of L_2:

$$\forall xyz\, \forall aba'b'.\;\; \mathsf{L}_2 xzyab = \mathsf{L}_2 xzya'b' \to (a,b) = (a',b')$$

Use the inversion operator from (c) and dependent pair injectivity for numbers:

$$\forall p^{\mathsf{N}\to\mathbb{T}}\ \forall x\ \forall ab^{px}.\ (x,a)_p = (x,b)_p \to a = b$$

You find a detailed discussion of this proof in § 23.5. Dependent pair injectivity for numbers will be shown in § 23.3 once we have introduced inductive equality.

22.2 Derivation Unique Variant

We have seen a sound and complete derivation system for comparisons. There is much freedom in how we can choose the derivation rules for such a system. In practice, one only includes defining derivation rules that are not needed for completeness, since every defining rule adds a clause to each eliminator (and hence to each inductive proof on derivations).

In this section, we consider another derivation systems for comparisons, which is sound, complete, and derivation unique. Derivation uniqueness means there is at most one derivation per comparison.

This time we choose the following derivation rules:

$$\frac{}{0 \dot{<} \mathsf{S}y} \qquad\qquad \frac{x \dot{<} y}{\mathsf{S}x \dot{<} \mathsf{S}y}$$

Our intuition is that the base rule fixes the distance between the two numbers placing the left number at 0. The step rule then shifts the pair to the right. Soundness, completeness, and derivation uniqueness are straightforward with this intuition. Note that the current system is linear (that is, has at most one premise per rule).

Formally, we define the derivation system described above as follows:

$$\begin{aligned}&\mathsf{L} :\ \mathsf{N} \to \mathsf{N} \to \mathbb{T}\ :=\\ &\mid \mathsf{L}_1 :\ \forall y.\ \mathsf{L}\,0(\mathsf{S}y)\\ &\mid \mathsf{L}_2 :\ \forall xy.\ \mathsf{L}xy \to \mathsf{L}\,(\mathsf{S}x)(\mathsf{S}y)\end{aligned}$$

Clearly, both arguments of L must be accommodated as indices. The definition yields a universal eliminator of the type

$$\begin{aligned}&\forall p^{\forall xy.\ \mathsf{L}xy\to\mathbb{T}}.\\ &(\forall y.\ p0(\mathsf{S}y)(\mathsf{L}_1 y)) \to\\ &(\forall xya.\ pxya \to p(\mathsf{S}x)(\mathsf{S}y)(\mathsf{L}_2\, xya)) \to\\ &\forall xya.\ pxya\end{aligned}$$

Soundness

$$\forall xy.\, \mathsf{L}xy \to x < y$$

follows as before by induction on the derivation of $\mathsf{L}xy$, requiring soundness of each of the two rules. **Completeness**

$$\forall xy.\, x < y \to \mathsf{L}xy$$

is more interesting. This time we do an induction on x with y quantified, which after discrimination on y yields the obligations

$$\mathsf{L}\,0(\mathsf{S}y)$$
$$\mathsf{S}x < \mathsf{S}y \to \mathsf{L}\,(\mathsf{S}x)(\mathsf{S}y)$$

The first obligation follows with L_1, and the second obligation follows with L_2 and the inductive hypothesis.

For derivation uniqueness, we use an **inversion operator**

$$\begin{aligned}\forall xy\ \forall a^{\mathsf{L}xy}.\ \ &\textsc{match}\ x, y\ \textsc{return}\ \mathsf{L}xy \to \mathbb{T}\\ &\mid 0, \mathsf{S}y \Rightarrow \lambda a.\ a = \mathsf{L}_1 y\\ &\mid \mathsf{S}x, \mathsf{S}y \Rightarrow \lambda a.\ \Sigma a'.\ a = \mathsf{L}_2 xya'\\ &\mid \_, \_ \Rightarrow \lambda a. \bot\\ &]\, a\end{aligned}$$

which can be defined by discrimination on the derivation a. **Derivation uniqueness**

$$\forall xy\ \forall ab^{\mathsf{L}xy}.\ \ a = b$$

now follows by induction on a with b quantified followed by inversion of b.

Exercise 22.2.1 Elaborate the above definitions and proofs using a proof assistant. Make sure you understand every detail, especially as it comes to the inductive proofs. Define the inversion operator without using a smart match for x, y. Practice to come up with the types of the universal eliminator and the inversion operator without using notes.

Exercise 22.2.2 Change the above development such that the types $\mathsf{L}xy$ appear as propositions. Note the changes needed in the types of the universal eliminator and the inversion operator.

Exercise 22.2.3 Prove the following propositions using the inversion operator. Do not use soundness.

a) $\mathsf{L}x0 \to \bot$

b) $\mathsf{L}xx \to \bot$

Hint: (b) follows by induction on x.

Exercise 22.2.4 Prove that the constructor L_2 is injective in its third argument. Hint: Use derivation uniqueness.

Exercise 22.2.5 Given an equality decider for the derivation types $\mathsf{L}xy$. Hint: Use derivation uniqueness.

Exercise 22.2.6 Here is another derivation unique derivation system for comparisons:

$$\frac{}{x \dot{<} \mathsf{S}x} \qquad\qquad \frac{x \dot{<} y}{x \dot{<} \mathsf{S}y}$$

This time the base rule fixes the left number and the step rule increases the right number.

a) Formalize the system with an indexed inductive type family L. Accommodate the first argument as a uniform parameter.

b) Show completeness of the system. Hint: Induction on y suffices.

c) Define the universal eliminator for L using the prefix $\forall x\, \forall p^{\forall y.\, \mathsf{L}xy \to \mathbb{T}}$. Note that there is no need that the type function p takes the uniform parameter x as argument.

d) Show soundness for the system using the universal eliminator.

e) Try to formulate an inversion operator for L and note that there is a typing conflict for the first rule. The conflict comes from the non-linearity $\mathsf{L}x(\mathsf{S}x)$ in the type of L_1. We will resolve the conflict with a type cast in § 23.4 in Chapter 23 on inductive equality. Using an inversion operator with a type cast we will prove derivation uniqueness in § 23.4.

Exercise 22.2.7 (Even numbers) Formalize the derivation system

$$\frac{}{\mathsf{E}(0)} \qquad\qquad \frac{\mathsf{E}(n)}{\mathsf{E}(\mathsf{SS}n)}$$

with an indexed inductive type family $E^{\mathsf{N} \to \mathbb{T}}$.

a) Prove $\mathsf{E}(2 \cdot k)$.

b) Define a universal eliminator for E.

c) Prove $\mathsf{E}(n) \to \Sigma k.\, n = 2 \cdot k$.

d) Prove $\mathsf{E}(n) \Leftrightarrow \exists k.\, n = 2 \cdot k$.

e) Define an inversion operator for E providing cases for 0, 1, and $n \geq 2$.

f) Prove $\mathsf{E}(1) \to \bot$.

g) Prove $\mathsf{E}(\mathsf{SS}n) \to \mathsf{E}(n)$.

h) Prove $\mathsf{E}(\mathsf{S}n) \to \mathsf{E}(n) \to \bot$.

i) Prove derivation uniqueness for E.

j) Give an equality decider for the derivation types $\mathsf{E}(n)$.

22.3 Derivation Systems for GCDs

Greatest common divisors (GCDs) can be determined with three derivation rules:

$$\frac{}{\mathsf{G}\,0yy} \qquad \frac{\mathsf{G}\,xyz}{\mathsf{G}\,yxz} \qquad \frac{x \le y \qquad \mathsf{G}\,x(y-x)z}{\mathsf{G}\,xyz}$$

The rules yield an indexed type family $\mathsf{G} : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathbb{P}$ with three indices. One can show that $\mathsf{G}\,xyz$ is derivable if and only if z is the GCD of x and y. A comprehensive development of GCDs including the inductive definitions discussed here appears in Chapter 24. Here we will show that G is functional and that G is the least gcd relation. We will obtain these results without using the arithmetic gcd predicate.

A **gcd relation** is a predicate $p^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathbb{P}}$ satisfying the following conditions for all numbers x, y, z:

1. $p0yy$ *zero rule*
2. $pyxz \to pxyz$ *symmetry rule*
3. $x \le y \to px(y-x)z \to pxyz$ *subtraction rule*

Note that the conditions for gcd relations correspond exactly to the derivation rules for G.

Fact 22.3.1 G is a least gcd relation.

Proof The rules defining G agree with the conditions for gcd predicates. Hence G is a gcd predicate. To show that G is a least gcd predicate, we assume a gcd predicate p and prove $\forall xyz.\, \mathsf{G}\,xyz \to pxyz$ by induction on the derivation $\mathsf{G}\,xyz$. The proof obligations generated by the induction are the gcd relation conditions for p. ■

In Chapter 24 we will see results saying that every gcd relation is total and that the arithmetic gcd predicate is a functional gcd relation. Together with Fact 22.3.1 this implies that G agrees with the arithmetic gcd predicate. The agreement in turn implies that G is functional.

Can we prove that G is functional just relying on methods for indexed inductive families and not using the agreement with the arithmetic gcd predicate? The best such proof the author knows uses a deterministic variant G' of G defined with the following rules:

$$\frac{}{\mathsf{G}'\,0yy} \qquad\qquad \frac{}{\mathsf{G}'\,(\mathsf{S}x)0(\mathsf{S}x)}$$

$$\frac{x \le y \qquad \mathsf{G}'\,(\mathsf{S}x)(y-x)z}{\mathsf{G}'\,(\mathsf{S}x)(\mathsf{S}y)z} \qquad\qquad \frac{y < x \qquad \mathsf{G}'\,(x-y)(\mathsf{S}y)z}{\mathsf{G}'\,(\mathsf{S}x)(\mathsf{S}y)z}$$

We will show that G' is a functional gcd relation. Since G is a least gcd relation, we then have $\mathsf{G} \subseteq \mathsf{G}'$, which gives us the functionality of G.

To show the necessary properties of G', we shall use an inversion operator with the type:

$$\begin{aligned}\forall xyz\,\forall a^{\mathsf{G}'xyz}.\ & \textsc{match}\ x, y\\ & [\,0,\ y \Rightarrow z = y\\ & \mid \mathsf{S}x,\ 0 \Rightarrow z = \mathsf{S}x\\ & \mid \mathsf{S}x,\ \mathsf{S}y \Rightarrow \textsc{if}\ \ulcorner x \le y\urcorner\ \textsc{then}\ \mathsf{G}'\,(\mathsf{S}x)(y-x)z\ \textsc{else}\ \mathsf{G}'\,(x-y)(\mathsf{S}y)z\\ &]\end{aligned}$$

Defining such an operator is routine.

Fact 22.3.2 (Commutativity) $\forall xyz.\ \mathsf{G}'\,xyz \to \mathsf{G}'\,yxz$.

Proof By induction on the derivation of $\mathsf{G}'\,xyz$. The interesting case is the third rule for G' with $x = y$, where we have to show $G'\,0(\mathsf{S}x)z \to G'(\mathsf{S}x)(\mathsf{S}x)z$. By inversion of $G'\,0(\mathsf{S}x)z$ we obtain $z = \mathsf{S}x$, which gives us the claim $G'(\mathsf{S}x)(\mathsf{S}x)(\mathsf{S}x)$, which follows with the third and fourth rule for G'. ■

Fact 22.3.3 G' is a gcd relation.

Proof The first condition is the first rule fo G'. The second condition is Fact 22.3.2. The third condition follows by case analysis on x and y and third rule for G'. ■

Fact 22.3.4 G' is functional.

Proof We show $\forall xyzz'.\ \mathsf{G}'\,xyz \to \mathsf{G}'\,xyz' \to z = z'$ by induction on the derivation of $\mathsf{G}'\,xyz$ and inversion of $\mathsf{G}'\,xyz$. All four obligations are straightforward. ■

Fact 22.3.5 $\mathsf{G} \subseteq \mathsf{G}'$. Hence G is functional.

Proof Since G is a least gcd relation (Fact 22.3.1) and G' is a gcd relation (Fact 22.3.3), we have $\mathsf{G} \subseteq \mathsf{G}'$. Hence G is functional since G' is functional (Fact 22.3.4). ■

There is the general result that all gcd relations are total (Fact 24.1.2). Hence G and G' agree.

22.4 Regular Expressions

Regular expressions are patterns for strings used in text search. There is a relation $A \vdash s$ saying that a string A *satisfies* a regular expression s. One also speaks of a regular expression *matching a string*. We are considering regular expressions here since the satisfaction relation $A \vdash s$ has an elegant and natural definition with derivation rules.

We represent *strings* as lists of numbers, and *regular expressions* with an inductive type realizing the BNF

$$s, t : \mathsf{exp} \ ::= \ x \mid \mathbf{0} \mid \mathbf{1} \mid s + t \mid s \cdot t \mid s^* \qquad (x : \mathsf{N})$$

We model the satisfaction relation $A \vdash s$ with an indexed inductive type family

$$\vdash \, : \ \mathcal{L}(\mathsf{N}) \to \mathsf{exp} \to \mathbb{T}$$

providing value constructors for the following rules:

$$\frac{}{[x] \vdash x} \qquad \frac{}{[\,] \vdash \mathbf{1}} \qquad \frac{A \vdash s}{A \vdash s + t} \qquad \frac{A \vdash t}{A \vdash s + t}$$

$$\frac{A \vdash s \qquad B \vdash t}{A + B \vdash s \cdot t} \qquad \frac{}{[\,] \vdash s^*} \qquad \frac{A \vdash s \qquad B \vdash s^*}{A + B \vdash s^*}$$

Note that both arguments of $\vdash$ are indices. Concrete instances of the satisfaction relation, for instance,

$$[1, 2, 2] \vdash 1 \cdot 2^*$$

can be shown with just constructor applications. **Inclusion** and **equivalence** of regular expressions are defined as follows:

$$\begin{aligned} s \subseteq t &:= \ \forall A.\, A \vdash s \ \to \ A \vdash t \\ s \equiv t &:= \ \forall A.\, A \vdash s \ \longleftrightarrow \ A \vdash t \end{aligned}$$

An easy to show inclusion is

$$s \subseteq s^* \tag{22.3}$$

(only constructor applications are needed). More challenging is the inclusion

$$s^* \cdot s^* \subseteq s^* \tag{22.4}$$

We need an inversion function

$$A \vdash s \cdot t \to \Sigma A_1 A_2.\ A = A_1 + A_2 \ \times \ A_1 \vdash s \ \times \ A_2 \vdash t \tag{22.5}$$

and a lemma

$$A \vdash s^* \;\rightarrow\; B \vdash s^* \;\rightarrow\; A +\!\!+ B \vdash s^* \tag{22.6}$$

The inversion function can be obtained as an instance of a more general **inversion operator**

$$\begin{aligned}\forall As.\ A \vdash s \;\rightarrow\; & \textsc{match}\ s\\ & [\, x \Rightarrow A = [x]\\ & \mid \mathbf{0} \Rightarrow \bot\\ & \mid \mathbf{1} \Rightarrow A = []\\ & \mid u + v \Rightarrow (A \vdash u) + (A \vdash v)\\ & \mid u \cdot v \Rightarrow \Sigma A_1 A_2.\ (A = A_1 +\!\!+ A_2) \times (A_1 \vdash u) \times (A_2 \vdash v)\\ & \mid u^* \Rightarrow (A = []) + \Sigma A_1 A_2.\ (A = A_1 +\!\!+ A_2) \times (A_1 \vdash u) \times (A_2 \vdash u^*)\\ &]\end{aligned}$$

which can be defined by discrimination on $A \vdash s$. Note that the index s determines a single rule except for s^*.

We now come to the proof of lemma (22.6). The proof is by induction on the derivation $A \vdash s^*$ with B fixed. There are two cases. If $A = []$, the claim is trivial. Otherwise $A = A_1 +\!\!+ A_2$, $A_1 \vdash s$, and $A_2 \vdash s^*$. Since $A_2 \vdash s^*$ is obtained by a subderivation, the inductive hypothesis gives us $A_2 +\!\!+ B \vdash s^*$. Hence $A_1 +\!\!+ A_2 +\!\!+ B \vdash s^*$ by the second rule for s^*.

The above induction is informal. It can be made formal with an universal eliminator for $A \vdash s$ and a reformulation of the claim as follows:

$$\forall As.\, A \vdash s \;\rightarrow\; \textsc{match}\ s\ [\, s^* \Rightarrow B \vdash s^* \;\rightarrow\; A +\!\!+ B \vdash s^* \mid \_ \Rightarrow \top \,]$$

The reformulation provides an unconstrained inductive premises $A \vdash s$ so that no information is lost by the application of the universal eliminator. Defining the universal eliminator with a type function $\forall As.\ A \vdash s \rightarrow \mathbb{T}$ is routine. We remark that a weaker eliminator with a type function $\mathcal{L}(\mathsf{N}) \rightarrow \mathtt{exp} \rightarrow \mathbb{T}$ suffices.

We now have (22.4). A straightforward consequence is

$$s^* \cdot s^* \equiv s^*$$

A less obvious consequence is the equivalence

$$(s^*)^* \equiv s^* \tag{22.7}$$

saying that the star operation is idempotent. Given (22.3), it suffices to show

$$A \vdash (s^*)^* \rightarrow A \vdash s^* \tag{22.8}$$

The proof is by induction on $A \vdash (s^*)^*$. If $A = []$, the claim is obvious. Otherwise, we assume $A_1 \vdash s^*$ and $A_2 \vdash (s^*)^*$, and show $A_1 \mathbin{+\!\!+} A_2 \vdash s^*$. The inductive hypothesis gives us $A_2 \vdash s^*$, which gives us the claim using (22.6).

The above proof is informal since the inductive premise $A \vdash (s^*)^*$ is constrained. A formal proof succeeds with the reformulation

$$\forall A s.\ A \vdash s \to \text{MATCH}\ s\ [\ (s^*)^* \Rightarrow A \vdash s^* \mid \_ \Rightarrow \top\ ]$$

Exercise 22.4.1 Define a certifying solver $\forall s.\ (\Sigma A.\ A \vdash s) + (\forall A.\ A \vdash s \to \bot)$. Hint: Straightforward with recursion on s.

Exercise 22.4.2 After reading this section, do the following with a proof assistant.

a) Define a universal eliminator for $A \vdash s$.
b) Define an inversion operator for $A \vdash s$.
c) Prove $s^* \cdot s^* \equiv s^*$.
d) Prove $(s^*)^* \equiv s^*$.

Exercise 22.4.3 (Denotational semantics) The informal semantics for regular expressions described in textbooks can be formalized as a recursive function on regular expressions that assigns languages to regular expressions. We represent languages as type functions $\mathcal{L}(\mathsf{N}) \to \mathbb{T}$ and capture the semantics with a function

$$\mathcal{R} : \mathsf{exp} \to \mathcal{L}(\mathsf{N}) \to \mathbb{T}$$

defined as follows:

$$\begin{aligned}
\mathcal{R}\,x\,A &:= (A = [x])\\
\mathcal{R}\,\mathbf{0}\,A &:= \bot\\
\mathcal{R}\,\mathbf{1}\,A &:= (A = [])\\
\mathcal{R}\,(s+t)\,A &:= \mathcal{R}sA + \mathcal{R}tA\\
\mathcal{R}\,(s\cdot t)\,A &:= \Sigma A_1 A_2.\ (A = A_1 \mathbin{+\!\!+} A_2) \times \mathcal{R}sA_1 \times \mathcal{R}tA_2\\
\mathcal{R}\,(s^*)\,A &:= \Sigma n.\ \mathcal{P}\,(\mathcal{R}s)nA
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}\,\varphi\,0\,A &:= (A = [])\\
\mathcal{P}\,\varphi\,(\mathsf{S}n)\,A &:= \Sigma A_1 A_2.\ (A = A_1 \mathbin{+\!\!+} A_2) \times \varphi A_1 \times \mathcal{P}\,\varphi\,n\,A
\end{aligned}$$

a) Prove $\mathcal{R}\,sA \Leftrightarrow A \vdash s$.
b) We have represented languages as type functions $\mathcal{L}(\mathsf{N}) \to \mathbb{T}$. A representation as predicates $\mathcal{L}(\mathsf{N}) \to \mathbb{P}$ would be more faithful to the literature. Rewrite the definitions of $\vdash$ and $\mathcal{R}$ accordingly and show their equivalence.

22.5 Decidability of Regular Expression Matching

We will now construct a decider for $A \vdash s$. The decidability of $A \vdash s$ is not obvious. We will formalize a decision procedure based on Brzozowski derivatives.

A function $D : \mathsf{N} \to \mathsf{exp} \to \mathsf{exp}$ is a **derivation function** if

$$\forall x A s.\ \ x :: A \vdash s \Leftrightarrow A \vdash Dxs$$

In words we may say that a string $x :: A$ satisfies a regular expression s if and only if A satisfies the **derivative** Dxs. If we have a decider $\forall s.\ \mathcal{D}([] \vdash s)$ and in addition a derivation function, we have a decider for $A \vdash s$.

Fact 22.5.1 $\forall s.\ \mathcal{D}([] \vdash s)$.

Proof By induction on s. For $\mathbf{1}$ and s^* we have a positive answer, and for x and $\mathbf{0}$ we have a negative answer using the inversion function. For $s + t$ and $s \cdot t$ we rely on the inductive hypotheses for the constituents. ■

Fact 22.5.2 $\forall A s.\ \mathcal{D}(A \vdash s)$ provided we have a derivation function.

Proof By recursion on A using Fact 22.5.1 in the base case and the derivation function in the cons case. ■

We define a derivation function D as follows:

$$\begin{aligned}
&D :\ \mathsf{N} \to \mathsf{exp} \to \mathsf{exp}\\
&\qquad Dxy \ := \ \text{IF } \ulcorner x = y \urcorner \text{ THEN } \mathbf{1} \text{ ELSE } \mathbf{0}\\
&\qquad Dx\,\mathbf{0} \ := \ \mathbf{0}\\
&\qquad Dx\,\mathbf{1} \ := \ \mathbf{0}\\
&\qquad Dx\,(s+t) \ := \ Dxs + Dxt\\
&\qquad Dx\,(s \cdot t) \ := \ \text{IF } \ulcorner [] \vdash s \urcorner \text{ THEN } Dxs \cdot t + Dxt \text{ ELSE } Dxs \cdot t\\
&\qquad Dx\,(s^*) \ := \ Dxs \cdot s^*
\end{aligned}$$

It remains to show that D is a derivation function. For this proof we need one essential lemma for star expressions.

Lemma 22.5.3 (Eager star inversion)
$\forall x A s.\ \ x :: A \vdash s^* \ \to\ \Sigma A_1 A_2.\ \ A = A_1 + A_2 \times x :: A_1 \vdash s \times A_2 \vdash s^*$.

Proof By induction on the derivation of $x :: A \vdash s^*$. Only the second rule for star expressions applies. Hence we have $x :: A = A_1 + A_2$ and subderivations $A_1 \vdash s$ and $A_2 \vdash s^*$. If $A_1 = []$, we have $A_2 = x :: A$ and the claim follows by the inductive hypothesis. Otherwise, we have $A_1 := x :: A_1'$, which gives us the claim.

The formal proof follows this outline but works on a reformulation of the claim providing an unconstrained inductive premise. ■

Theorem 22.5.4 (Derivation) $\forall xAs.\; x :: A \vdash s \Leftrightarrow A \vdash Dxs.$

Proof By induction on s. All cases but the direction $\Rightarrow$ for s^* follow with the inversion operator and case analysis. The direction $\Rightarrow$ for s^* follows with the eager star inversion lemma 22.5.3. ■

Corollary 22.5.5 $\forall As.\; \mathcal{D}(A \vdash s).$

Proof Follows with Fact 22.5.2 and Theorem 22.5.4. ■

Exercise 22.5.6 In the light of eager star inversion (Lemma 22.5.3), it seems possible to replace the second derivation rule for star expressions with the more restrictive rule

$$\frac{x :: A \vdash s \qquad B \vdash s^*}{x :: A +\!\!+ B \vdash s^*}$$

Define an inductive family $A \dot{\vdash} s$ adopting the more restrictive rule and show that it is intertranslatable with $A \vdash s$: $\forall As.\; A \dot{\vdash} s \Leftrightarrow A \vdash s.$

22.6 Post Correspondence Problem

Many problems in computer science have elegant specifications using inductive relations. As an example we consider the Post correspondence problem (PCP), a prominent undecidable problem providing a base for undecidability proofs. The problem involves cards with an upper and a lower string. Given a list C of cards, one has to decide whether there is a nonempty list $D \subseteq C$ such that the concatenation of all upper strings equals the concatenation of all lower strings. For instance, assuming the binary alphabet $\{a, b\}$, the list

$$C = [a/\epsilon,\; b/a,\; \epsilon/bb]$$

has the solution

$$D = [\epsilon/bb,\; b/a,\; b/a,\; a/\epsilon,\; a/\epsilon]$$

On the other hand,

$$C' = [a/\epsilon,\; b/a]$$

has no solution.

We formalize PCP over the binary alphabet B with an inductive predicate

$$\mathsf{post} : \mathcal{L}(\mathcal{L}(\mathsf{B}) \times \mathcal{L}(\mathsf{B})) \to \mathcal{L}(\mathsf{B}) \to \mathcal{L}(\mathsf{B}) \to \mathbb{P}$$

defined with the rules

$$\frac{(A,B) \in C}{\textsf{post}\ C\,A\,B} \qquad\qquad \frac{(A,B) \in C \qquad \textsf{post}\ C\,A'\,B'}{\textsf{post}\ C\,(A +\!\!\!+ A')\,(B +\!\!\!+ B')}$$

Note that $\textsf{post}\,CAB$ is derivable if there is a nonempty list $D \subseteq C$ of cards such that the concatenation of the upper strings of D is A and the concatenation of the lower strings of D is B. Undecidability of PCP over a binary alphabet now means that there is no computable function

$$\forall C.\ \mathcal{D}(\exists A.\ \textsf{post}\,CAA) \tag{22.9}$$

Since Coq's type theory can only define computable functions, we can conclude that no function of type (22.9) is definable.

23 Inductive Equality

Inductive equality extends Leibniz equality with eliminators discriminating on identity proofs. The definitions are such that inductive identities appear as computational propositions enabling reducible casts between computational types.

There is an important equivalence between uniqueness of identity proofs (UIP) and injectivity of dependent pairs (DPI) (i.e., injectivity of the second projection). As it turns out, UIP holds for discrete types (Hedberg's theorem) but is unprovable in computational type theory in general

Hedberg's theorem is of practical importance since it yields injectivity of dependent pairs and reducibility of identity casts for discrete types, two features that are essential for inversion lemmas for indexed inductive types.

The proofs in this chapter are of surprising beauty. They are obtained with dependently typed algebraic reasoning about identity proofs and often require tricky generalizations.

23.1 Basic Definitions

Recall the definition of inductive equality:

$$\begin{aligned}&\mathsf{eq}\,(X:\mathbb{T},\ x:X):X\to\mathbb{P}\ ::=\\&\mid \mathsf{q}:\ \mathsf{eq}\ X\ x\ x\end{aligned}$$

We treat the argument X of the constructors eq and q as implicit argument and write $s = t$ for $\mathsf{eq}\,st$. Moreover, we call propositions $s = t$ **identities**, and refer to proofs of identities $s = t$ as **paths** from s to t.

Note that identities $s = t$ are computational propositions. This provides for expressivity we cannot obtain with Leibniz equality. We define two eliminators for identities

$$\begin{aligned}&C:\ \forall X^{\mathbb{T}}\,\forall x^X\,\forall p^{X\to\mathbb{T}}\,\forall y.\ x = y \to px \to py\\&C\,Xxp\,\_\,(\mathsf{q}\_)\,a\ :=\ a\end{aligned}$$

$$\begin{aligned}&\mathcal{J}:\ \forall X^{\mathbb{T}}\,\forall x^X\,\forall p^{\forall y.\ x=y\to\mathbb{T}}.\ px(\mathsf{q}x)\to\forall ye.\ pye\\&\mathcal{J}\,Xxpa\,\_\,(\mathsf{q}\_)\ :=\ a\end{aligned}$$

called **cast operator** and **full eliminator**. Note that the cast operator is the canonical index eliminator for identities. For C we treat the first four arguments as implicit arguments, and for $\mathcal{J}$ the first two arguments.

We call applications of the cast operator **casts**. A cast $C_p e a$ with $e^{x=y}$ changes the type of a from px to py for every admissible type function p. We have

$$C(\mathsf{q}x)a \;\approx\; a$$

and say that trivial casts $C(\mathsf{q}x)a$ can be **discharged**. We also have

$$\forall p^{X\to\mathbb{T}}\,\forall e^{x=y}\forall a^{px}.\;C_p e a \;\approx\; \mathcal{J}(\lambda y\_.py)aye$$

which says that the cast eliminator can be expressed with the full eliminator.

Inductive quality as defined here is stronger than the Leibniz equality considered in Chapter 5. The constructors of the inductive definition give us the constants eq and q, and with the cast operator we can easily define the constant for the rewriting law. Inductive equality comes with two essential generalizations over Leibniz equality: Rewriting can now take place at the universe $\mathbb{T}$ using the cast operator, and both the cast operator and the full eliminator come with computation rules. We will make essential use of both features in this chapter.

We remark that equality in Coq is defined as inductive equality and that the full eliminator $\mathcal{J}$ corresponds exactly to Coq's matches for identities.

The laws for propositional equality can be seen as operators on paths. It turns out that that these operators have elegant algebraic definitions using casts:

$$\begin{aligned}
&\sigma : \; x = y \to y = x\\
&\sigma e \;:=\; C_{(\lambda y.y=x)}\,e\,(\mathsf{q}x)
\end{aligned}$$

$$\begin{aligned}
&\tau : \; x = y \to y = z \to x = z\\
&\tau e \;:=\; C_{(\lambda y.y=z\to x=z)}\,e\,(\lambda e.e)
\end{aligned}$$

$$\begin{aligned}
&\varphi : \; x = y \to fx = fy\\
&\varphi e \;:=\; C_{(\lambda y.fx=fy)}\,e\,(\mathsf{q}(fx))
\end{aligned}$$

It also turns out that these operators satisfy familiar looking algebraic laws.

Exercise 23.1.1 Prove the following algebraic laws for casts and identities $e^{x=y}$.

a) $Ce(\mathsf{q}x) = e$

b) $Cee = \mathsf{q}y$

In each case, determine a suitable type function for the cast.

Exercise 23.1.2 (Groupoid operations on paths)
Prove the following algebraic laws for σ and τ:

a) $\sigma(\sigma e) = e$

b) $\tau e_1(\tau e_2 e_3) = \tau(\tau e_1 e_2)e_3$

c) $\tau e(\sigma e) = \mathsf{q}x$

Note that σ and τ give identity proofs a group-like structure: τ is an associative operation and σ obtains inverse elements.

Exercise 23.1.3 (Impredicative characterization)
Prove $x = y \longleftrightarrow \forall p^{X \to \mathbb{P}}.\ px \to py$ for inductive identities. Note that the equivalence says that inductive identities agree with Leibniz identities (§ 5.3).

23.2 Uniqueness of Identity Proofs

We will now show that the following properties of types are equivalent:

$\mathsf{UIP}(X) :=$	$\forall xy^X\ \forall ee'^{x=y}.\ e = e'$	*uniqueness of identity proofs*
$\mathsf{UIP}'(X) :=$	$\forall x^X\ \forall e^{x=x}.\ e = \mathsf{q}x$	*u. of trivial identiy proofs*
$\mathsf{K}(X) :=$	$\forall x\ \forall p^{x=x\to\mathbb{P}}.\ p(\mathsf{q}x) \to \forall e.pe$	*Streicher's K*
$\mathsf{CD}(X) :=$	$\forall p^{X\to\mathbb{T}}\ \forall x\ \forall a^{px}\ \forall e^{x=x}.\ Cea = a$	*cast discharge*
$\mathsf{DPI}(X) :=$	$\forall p^{X\to\mathbb{T}}\ \forall xuv.\ (x,u)_p = (x,v)_p \to u = v$	*dependent pair injectivity*

The flagship property is UIP (uniqueness of identity proofs), saying that identities have at most one proof. What is fascinating is that UIP is equivalent to DPI (dependent pair injectivity), saying that the second projection for dependent pairs is injective. While UIP is all about identity proofs, DPI doesn't even mention identity proofs. There is a famous result by Hofmann and Streicher [12] saying that computational type theory does not prove UIP. Given the equivalence with DPI, this result is quite surprising. On the other hand, there is Hedberg's theorem [11] (§ 23.3) saying that UIP holds for all discrete types. We remark that UIP is an immediate consequence of proof irrelevance.

We now show the above equivalence by proving enough implications. The proofs are interesting in that they need clever generalization steps to harvest the power of the identity eliminators $\mathcal{J}$ and C. Finding the right generalizations requires insight and practice.[1]

Fact 23.2.1 $\mathsf{UIP}(X) \to \mathsf{UIP}'(X)$.

Proof Instantiate $\mathsf{UIP}(X)$ with $y := x$ and $e' := \mathsf{q}x$. ■

[1]We acknowledge the help of Gaëtan Gilbert, (Coq Club, November 13, 2020).

Fact 23.2.2 $\mathsf{UIP}'(X) \to \mathsf{K}(X)$.

Proof Instantiate $\mathsf{UIP}'(X)$ with e from $\mathsf{K}(X)$ and rewrite. ∎

Fact 23.2.3 $\mathsf{K}(X) \to \mathsf{CD}(X)$.

Proof Apply $\mathsf{K}(X)$ to $\forall e^{x=x}.\ Cea = a$. ∎

Fact 23.2.4 $\mathsf{CD}(X) \to \mathsf{DPI}(X)$.

Proof Assume $\mathsf{CD}(X)$ and $p^{X\to\mathbb{T}}$. We obtain the claim with backward reasoning:

$$\begin{array}{rr}
\forall xuv.\ (x,u)_p = (x,v)_p \to u = v & \text{by instantiation} \\
\forall ab^{\mathsf{sig}\,p}.\ a = b \to \forall e^{\pi_1 a = \pi_1 b}.\ Ce(\pi_2 a) = \pi_2 b & \text{by elimination on } a = b \\
\forall a^{\mathsf{sig}\,p}\ \forall e^{\pi_1 a = \pi_1 a}.\ Ce(\pi_2 a) = \pi_2 a & \text{by CD}
\end{array}$$ ∎

Fact 23.2.5 $\mathsf{DPI}(X) \to \mathsf{UIP}'(X)$.

Proof Assume $\mathsf{DPI}(X)$. We obtain the claim with backward reasoning:

$$\begin{array}{rr}
\forall e^{x=x}.\ e = \mathsf{q}x & \text{by DPI} \\
\forall e^{x=x}.\ (x,e)_{\mathsf{eq}\,x} = (x,\mathsf{q}x)_{\mathsf{eq}\,x} & \text{by instantiation} \\
\forall e^{x=y}.\ (y,e)_{\mathsf{eq}\,x} = (x,\mathsf{q}x)_{\mathsf{eq}\,x} & \text{by } \mathcal{J}
\end{array}$$ ∎

Fact 23.2.6 $\mathsf{UIP}'(X) \to \mathsf{UIP}(X)$.

Proof Assume $\mathsf{UIP}'(X)$. We obtain the claim with backward reasoning:

$$\begin{array}{rr}
\forall e' e^{x=y}.\ e = e' & \text{by } \mathcal{J} \text{ on } e' \\
\forall e^{x=x}.\ e = \mathsf{q}x & \text{by UIP}'
\end{array}$$ ∎

Theorem 23.2.7 $\mathsf{UIP}(X)$, $\mathsf{UIP}'(X)$, $\mathsf{K}(X)$, $\mathsf{CD}(X)$, and $\mathsf{DPI}(X)$ are equivalent.

Proof Immediate by the preceding facts. ∎

Exercise 23.2.8 Verify the above proofs with a proof assistant to appreciate the subtleties.

Exercise 23.2.9 Give direct proofs for the following implications: $\mathsf{UIP}(X) \to \mathsf{K}(X)$, $\mathsf{K}(X) \to \mathsf{UIP}'(X)$, and $\mathsf{CD}(X) \to \mathsf{UIP}'(X)$.

Exercise 23.2.10 Prove that dependent pair types are discrete if their component types are discrete: $\forall X\, \forall p^{X\to\mathbb{T}}.\ \mathcal{E}(X) \to (\forall x.\ \mathcal{E}(px)) \to \mathcal{E}(\mathsf{sig}\,p)$.

23.3 Hedberg's Theorem

We will now prove Hedberg's theorem [11]. Hedberg's theorem says that all discrete types satisfy UIP. Hedberg's theorem is important in practice since it says that the second projection for dependent pair types is injective if the first components are numbers.

The proof of Hedberg's theorem consists of two lemmas, which are connected with a clever abstraction we call Hedberg functions. In algebraic speak one may see a Hedberg function a polymorphic constant endo-function on paths.

Definition 23.3.1 A function $f : \forall xy^X.\ x = y \to x = y$ is a **Hedberg function for X** if $\forall xy^X\ \forall ee'^{x=y}.\ fe = fe'$.

Lemma 23.3.2 (Hedberg) Every type that has a Hedberg function satisfies UIP.

Proof Let $f : \forall xy^X.\ x = y \to x = y$ be a Hedberg function for X. We treat x, y as implicit arguments and prove the equation

$$\forall xy\ \forall e^{x=y}.\ \ \tau(fe)(\sigma(f(\mathsf{q}y))) = e$$

We first destructure e, which reduces the claim to

$$\tau(f(\mathsf{q}x))(\sigma(f(\mathsf{q}x))) = \mathsf{q}x$$

which is an instance of equation (c) shown in Exercise 23.1.2.

Now let $e, e' : x = y$. We show $e = e'$. Using the above equation twice, we have

$$e = \tau(fe)(\sigma(f(\mathsf{q}y))) = \tau(fe')(\sigma(f(\mathsf{q}y))) = e'$$

since $fe = fe'$ since f is a Hedberg function. ■

Lemma 23.3.3 Every discrete type has a Hedberg function.

Proof Let d be an equality decider for X. We define a Hedberg function for X as follows:

$$fxye\ :=\ \text{IF } dxy \text{ is } \mathsf{L}\hat{e} \text{ THEN } \hat{e} \text{ ELSE } e$$

We need to show $fxye = fxye'$. If $dxy = \mathsf{L}\hat{e}$, both sides are $\hat{e}$. Otherwise, we have $e : x = y$ and $x \neq y$, which is contradictory. ■

Theorem 23.3.4 (Hedberg) Every discrete type satisfies UIP.

Proof Lemma 23.3.3 and Lemma 23.3.2. ■

Corollary 23.3.5 Every discrete type satisfies DPI.

Proof Theorems 23.3.4 and 23.2.7. ∎

Exercise 23.3.6 Prove Hedberg's theorem with the weaker assumption that equality on X is propositionally decidable: $\forall xy^X.\ x = y \lor x \neq y$.

Exercise 23.3.7 Prove Hedberg's theorem with the weaker assumption that identities on X are propositionally decidable: $\forall xy^X.\ x = y \lor x \neq y$.

Exercise 23.3.8 Construct a Hedberg function for X assuming FE and stability of equality on X: $\forall xy^X.\ \neg\neg(x = y) \to x = y$.

Exercise 23.3.9 Assume FE and show that $\mathsf{N} \to \mathsf{B}$ satisfies UIP.
Hint: Use Exercises 23.3.8 and 12.3.12.

23.4 Inversion with Casts

Sometimes a full inversion operator for an indexed inductive type family can only be expressed with a cast. As example we consider derivation types for comparisons $x < y$ defined as follows:

$$\begin{aligned}
&\mathsf{L}\,(x:\mathsf{N}) :\ \mathsf{N} \to \mathbb{T}\ ::= \\
&\mid \mathsf{L}_1 :\ \mathsf{L}x(\mathsf{S}x) \\
&\mid \mathsf{L}_2 :\ \forall y.\ \mathsf{L}xy \to \mathsf{L}x(\mathsf{S}y)
\end{aligned}$$

The type of the inversion operator for L can be expressed as

$$\begin{aligned}
\forall xy\ \forall a^{\mathsf{L}xy}.\ &\textsc{match}\ y\ \textsc{return}\ \mathsf{L}xy \to \mathbb{T} \\
&[\,0 \Rightarrow \lambda a.\ \bot \\
&\mid \mathsf{S}y' \Rightarrow \lambda a^{\mathsf{L}x(\mathsf{S}y')}.\ (\Sigma e^{y'=x}.\ Cea = \mathsf{L}_1 x) + (\Sigma a'.\ a = \mathsf{L}_2 xy'a') \\
&]\,a
\end{aligned}$$

The formulation of the type follows the pattern we have seen before, except that there is a cast in the branch for L_1:

$$\Sigma e^{y'=x}.\ Cea = \mathsf{L}_1 x$$

The cast is necessary since a has the type $\mathsf{L}x(\mathsf{S}y')$ while $\mathsf{L}_1 x$ has the type $\mathsf{L}x(\mathsf{S}x)$. A formulation without a cast seems impossible. The defining equations for the inversion operator discriminate on a, as usual, which yields the obligations

$$\begin{aligned}
&\Sigma e^{x=x}.\ Ce(\mathsf{L}_1 x) = \mathsf{L}_1 x \\
&\Sigma a'.\ \mathsf{L}_2 xy'a = \mathsf{L}_2 xy'a'
\end{aligned}$$

The first obligation follows with cast discharge and UIP for numbers. The second obligation is trivial.

We need the inversion operator to show derivation uniqueness of L. As it turns our, we need an additional fact about L:

$$\mathsf{L}\,xx \to \bot \tag{23.1}$$

This fact follows from a more semantic fact

$$\mathsf{L}\,xy \to x < y \tag{23.2}$$

which follows by induction on $\mathsf{L}\,xy$. We don't have a direct proof of (23.1).

We now prove derivation uniqueness

$$\forall xy\ \forall ab^{\mathsf{L}\,xy}.\ a = b$$

for L following the usual scheme (induction on a with b quantified followed by inversion of b). This gives four cases, where the contradictory cases follow with (23.1). The two remaining cases

$$\forall b^{\mathsf{L}\,x(\mathsf{S}x)}\ \forall e^{x=x}.\ Ceb = b$$
$$\mathsf{L}_2\,xya' = \mathsf{L}_2\,xyb'$$

follow with UIP for numbers and the inductive hypothesis, respectively.

We can also define an *index inversion operator for* L

$$\forall xy\ \forall a^{\mathsf{L}\,xy}.\ \textsc{match}\ y\ [\,0 \Rightarrow \bot \mid \mathsf{S}y' \Rightarrow x \neq y' \to \mathsf{L}\,xy'\,]$$

by discriminating on a.

Exercise 23.4.1 The proof sketches described above involve sophisticated type checking and considerable technical detail, more than can be certified reliably on paper. Use the proof assistant to verify the above proof sketches.

23.5 Constructor Injectivity with DPI

We present another inversion fact that can only be verified with UIP for numbers. This time we need DPI for numbers. We consider the indexed type family

$$\begin{aligned}
&\mathsf{K}\,(x : \mathsf{N}) :\ \mathsf{N} \to \mathbb{T}\ ::= \\
&\mid \mathsf{K}_1 :\ \mathsf{K}\,x(\mathsf{S}x) \\
&\mid \mathsf{K}_2 :\ \forall zy.\ \mathsf{K}\,xz \to \mathsf{K}\,zy \to \mathsf{K}\,xy
\end{aligned}$$

which provides a derivation system for arithmetic comparisons $x < y$ taking transitivity as a rule. Obviously, K is not derivation unique. We would like to show that the value constructor K_2 is injective:

$$\forall a^{\mathsf{K}xz}\ \forall b^{\mathsf{K}zy}.\ \ \mathsf{K}_2xzyab = \mathsf{K}_2xzya'b' \to (a,b) = (a',b') \tag{23.3}$$

We will do this with a customized index inversion operator

$$\mathsf{K}_{\mathsf{inv}} :\ \forall xy.\, \mathsf{K}xy \to (y = \mathsf{S}x) + (\Sigma z.\, \mathsf{K}xz \times \mathsf{K}zy)$$

satisfying

$$\mathsf{K}_{\mathsf{inv}}\, xy(\mathsf{K}_2xzyab) \approx \mathsf{R}\,(z,(a,b))$$

(R is one of the two value constructors for sums). Defining the inversion operator $\mathsf{K}_{\mathsf{inv}}$ is routine. We now prove (23.3) by applying $\mathsf{K}_{\mathsf{inv}}$ using feq to both sides of the assumed equation of (23.3), which yields

$$\mathsf{R}\,(z,(a,b)) = \mathsf{R}\,(z,(a',b'))$$

Now the injectivity of the sum constructor R (a routine proof) yields

$$(z,(a,b)) = (z,(a',b'))$$

which yields $(a,b) = (a',b')$ with DPI for numbers.

The proof will also go through with a simplified inversion operator $\mathsf{K}_{\mathsf{inv}}$ where in the sum type is replaced with the option type $\mathcal{O}(\Sigma z.\, \mathsf{K}xz \times \mathsf{K}zy)$. However, the use of a dependent pair type seems unavoidable, suggesting that injectivity of K_2 cannot be shown without DPI.

Exercise 23.5.1 Prove injectivity of the constructors for sum using feq.

Exercise 23.5.2 Prove injectivity of K_2 using a customized inversion operator employing an option type rather than a sum type.

Exercise 23.5.3 Prove injectivity of K_2 with the dependent elimination tactic of Coq's Equations package.

Exercise 23.5.4 Define the full inversion operator for K.

Exercise 23.5.5 Prove $\mathsf{K}xy \Leftrightarrow x < y$.

Exercise 23.5.6 Prove that there is no function $\forall xy.\, \mathsf{K}xy \to \Sigma z.\, \mathsf{K}xz \times \mathsf{K}zy$.

23.6 Inductive Equality at Type

We define an inductive equality type at the level of general types

$$\begin{aligned}&\mathsf{id}\,(X:\mathbb{T},\; x:X):X\to\mathbb{T} \;::=\\ &\mid \mathsf{I}:\;\mathsf{id}\;X\;x\;x\end{aligned}$$

and ask how propositional inductive equality and **computational inductive equality** are related. In turns out that we can go back and forth between proofs of propositional identities $x = y$ and derivations of general identities $\mathsf{id}\,xy$, and that UIP at one level implies UIP at the other level. We learn from this example that assumptions concerning only the propositional level (i.e., UIP) may leak out to the computational level and render nonpropositional types inhabited that seem to be unconnected to the propositional level.

First, we observe that we can define transfer functions

$$\begin{aligned}&\uparrow:\;\forall X\,\forall xy^X\,\forall e^{x=y}.\;\mathsf{id}\,xy\\ &\downarrow:\;\forall X\,\forall xy^X\,\forall a^{\mathsf{id}\,xy}.\;x=y\end{aligned}$$

such that $\uparrow(\mathsf{q}x)\approx\mathsf{I}x$ and $\downarrow(\mathsf{I}x)\approx\mathsf{q}x$ for all x, and $\downarrow(\uparrow e)=e$ and $\uparrow(\downarrow a)=a$ for all e and a. We can also define a function

$$\varphi:\forall XY\,\forall f^{X\to Y}\forall xx'^X.\;\mathsf{id}\,xx'\to\mathsf{id}\,(fx)(fx')$$

Fact 23.6.1 $\mathsf{UIP}\,X\to\forall xy^X\,\forall ab^{\mathsf{id}\,xy}.\;\mathsf{id}\,ab.$

Proof We assume $\mathsf{UIP}\,X$ and $x,y:X$ and $a,b:\mathsf{id}\,xy$. We show $\mathsf{id}\,ab$. It suffices to show

$$\mathsf{id}\,(\uparrow(\downarrow a))(\uparrow(\downarrow b))$$

By φ it suffices to show $\mathsf{id}\,(\downarrow a)(\downarrow b)$. By $\uparrow$ it suffices to show $\downarrow a=\downarrow b$, which holds by the assumption $\mathsf{UIP}\,X$. ■

Exercise 23.6.2 Prove the converse direction of Fact 23.6.1.

Exercise 23.6.3 Prove Hedberg's theorem for general inductive equality. Do not make use of propositional types.

Exercise 23.6.4 Formulate the various UIP characterizations for general inductive equality and prove their equivalence. Make sure that you don't use propositional types. Note that the proofs from the propositional level carry over to the general level.

23.7 Notes

The dependently typed algebra of identity proofs identified by Hofmann and Streicher [12] plays an important role in homotopy type theory [20], a recent branch of type theory where identities are accommodated as nonpropositional types and UIP is inconsistent with the so-called univalence assumption. Our proof of Hedberg's theorem follows the presentation of Kraus et al. [14]. That basic type theory cannot prove UIP was discovered by Hofmann and Streicher [12] in 1994 based on a so-called groupoid interpretation.

24 Case Study: GCDs

We study the construction of functions computing greatest common divisors. The main tools are size recursion and indexed inductive predicates.

24.1 GCD Predicates

We start with an abstract class of predicates satisfying three basic rules providing for the recursive computation of greatest common divisors (GCDs).

Definition 24.1.1 A **gcd predicate** is a predicate $p^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathbb{P}}$ satisfying the following conditions for all numbers x, y, z:

1. $p0yy$ *zero rule*
2. $pyxz \to pxyz$ *symmetry rule*
3. $x \le y \to px(y-x)z \to pxyz$ *subtraction rule*

A proposition $pxyz$ may be read as saying that z is the GCD of x and y. We refer to the above propositions as rules to highlight their computational interpretation:

1. The GCD of 0 and y is y.
2. The GCD of x and y is the GCD of y and x.
3. The GCD of x and y is the GCD of x and $y - x$ provide $x \le y$.

Fact 24.1.2 (Totality) Let p be a gcd predicate. Then $\forall xy\,\Sigma z.\ pxyz$.

Proof By size recursion on $x + y$. If $x = 0$ or $y = 0$ the claim follows with the zero and the symmetry rule. Otherwise we assume without loss of generality that $x \le y$. Since $x+(y-x) < x+y$, the induction hypothesis gives us $px(y-x)z$ for some z. The claim follows with the subtraction rule. ∎

The rules characterizing gcd predicates have a special form called *Horn clause format*. Due to this format we can inductively define a **least gcd predicate**:

$$\begin{aligned}
&G : \mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathbb{P} ::= \\
&\mid G_1 : \forall y.\ G0yy \\
&\mid G_2 : \forall xy.\ Gyxz \to Gxyz \\
&\mid G_3 : \forall xyz.\ x \le y \to Gx(y-x)z \to Gxyz
\end{aligned}$$

Fact 24.1.3 G is a least gcd predicate.

Proof By definition G is a gcd predicate. Let p be some gcd predicate. Then $\forall xy.\ Gxyz \to pxyz$ follows by induction on the derivation $Gxyz$. ∎

We will show that G agrees with every functional gcd predicate. We recall a few standard definitions for predicates $X \to Y \to Z \to \mathbb{P}$ and functions $X \to Y \to Z$:

- **p is total** if $\forall xy\,\exists z.\ pxyz$. $\quad$ total p
- **p is functional** if $\forall xyzz'.\ pxyz \to pxyz' \to z = z'$. $\quad$ fun p
- **p is subsumed by p'** if $\forall xyz.\ pxyz \to p'xyz$. $\quad$ $p \subseteq p'$
- **p agrees with p'** if $\forall xyz.\ pxyz \longleftrightarrow p'xyz$. $\quad$ $p \equiv p'$
- **f respects p** if $\forall xy.\ pxy(fxy)$. $\quad$ $f \subseteq p$
- **f agrees with p** if $\forall xyz.\ pxyz \longleftrightarrow fxy = z$. $\quad$ $f \equiv p$
- **f agrees with f'** if $\forall xy.\ fxy = f'xy$. $\quad$ $f \equiv f'$

We remark that the definitions are given for the case where p has three arguments and f has two arguments. The standard case would be p with 2 arguments and f with one argument. The cases where p has more than one argument may then be obtained with pairing and unpairing.

Fact 24.1.4

1. $p \subseteq p' \to \mathsf{fun}\,p' \to \mathsf{fun}\,p$.
2. $\mathsf{total}\,p \to \mathsf{fun}\,p' \to p \subseteq p' \to p \equiv p'$.
3. $\mathsf{fun}\,p \to f \subseteq p \to f \equiv p$.
4. $f \equiv p \to \mathsf{fun}\,p$.

Fact 24.1.5 G agrees with every functional gcd predicate.

Proof Follows with Facts 24.1.3, 24.1.2, and 24.1.4 (2). ∎

Fact 24.1.6 There is a function that respects every gcd predicate.

Proof Since G is a least gcd predicate (Fact 24.1.3) it suffices to show that there is a function that respects G. Follows by Fact 24.1.2. ∎

Fact 24.1.7 There is a function that agrees with every functional gcd predicate.

Proof Follows with Facts 24.1.6 and 24.1.4 (3). ∎

24.2 GCD Recursion

Given a gcd predicate p, Fact 24.1.2 constructs a function $\forall xy\,\Sigma z.\ pxyz$ using size recursion on $x + y$. The construction can be simplified with a *customized recursion operator* explicating the computational structure behind gcd predicates.

Fact 24.2.1 (GCD recursion)

$$\begin{aligned}
&\forall p^{\mathsf{N}\to\mathsf{N}\to\mathbb{T}}.\\
&(\forall y.\ p0y) \to\\
&(\forall xy.\ pxy \to pyx) \to\\
&(\forall xy.\ x \le y \to px(y-x) \to pxy) \to\\
&\forall xy.\ pxy
\end{aligned}$$

Proof By binary size induction on $x + y$ considering four cases: $x = 0$, $y = 0$, $x \le y$, and $y < x$. ∎

Note that with gcd recursion, the proof of Fact 24.1.2 becomes even simpler .

Exercise 24.2.2 Convince yourself that gcd recursion is stronger than the index induction principle for the inductive predicate G. Show that the index induction principle for G can be obtained with gcd recursion.

Exercise 24.2.3 (Deterministic gcd recursion) There are many recursion schemes that go with the recursive structure of gcd predicates. Prove the following deterministic recursion scheme.

$$\begin{aligned}
&\forall p^{\mathsf{N}\to\mathsf{N}\to\mathbb{T}}.\\
&(\forall y.\ p0y) \to\\
&(\forall x.\ p(\mathsf{S}x)0) \to\\
&(\forall xy.\ x \le y \to p(\mathsf{S}x)(y-x) \to p(\mathsf{S}x)(\mathsf{S}y)) \to\\
&(\forall xy.\ y < x \to p(x-y)(\mathsf{S}y) \to p(\mathsf{S}x)(\mathsf{S}y)) \to\\
&\forall xy.\ pxy
\end{aligned}$$

24.3 Arithmetic GCD Predicate

We will now define an arithmetic gcd predicate and show that it agrees with G. This gives us a function computing GCDs.

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

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

and specify GCDs with the following predicate:

$$\gamma x y z := \forall n.\ n \mid z \longleftrightarrow n \mid x \land n \mid y \qquad \textbf{GCD of } x \textbf{ and } y \textbf{ is } z$$

We call z the **GCD of x and y** if $\gamma x y z$. Provided x and y are not both 0, the number z is the greatest common divisor of x and y. We will show that γ is a functional gcd predicate. We start with the relevant facts about divisibility.

Fact 24.3.1

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 24.3.2 γ is a functional gcd predicate.

Proof The zero rule follows with Fact 24.3.1 (1). The symmetry rule is obvious from the definition. The subtraction rule follows with Fact 24.3.1 (2). The functionality of γ follows with Fact 24.3.1 (5). ■

Corollary 24.3.3 γ agrees with G.

Proof Follows with Fact 24.1.5. ■

Corollary 24.3.4 There is a function $f^{\mathsf{N} \to \mathsf{N} \to \mathsf{N}}$ computing GCDs (i.e., $f \equiv \gamma$).

Proof Follows with Fact 24.1.7. ■

24.4 Deterministic GCD Predicate

Can we show that the inductive gcd predicate G from § 24.1 is functional without using the arithmetic gcd predicate γ? The answer is yes, and the necessary proof involves an interesting idea. What we will do is define an inductive predicate G' whose rules follow the deterministic recursion scheme from Exercise 24.2.3. With routine proofs we then show that G' is a functional gcd predicate. Thus G' agrees with G, which gives us the functionality of G. The arithmetic gcd predicate γ will not be used.

Definition 24.4.1 (Deterministic gcd predicate)

$$\begin{aligned}
&G' : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathbb{P} \ ::= \\
&\mid G'_1 : \ \forall y.\ G'0yy \\
&\mid G'_2 : \ \forall x.\ G'(\mathsf{S}x)0(\mathsf{S}x) \\
&\mid G'_3 : \ \forall xyz.\ x \le y \to G'(\mathsf{S}x)(y-x)z \to G'(\mathsf{S}x)(\mathsf{S}y)z \\
&\mid G'_4 : \ \forall xyz.\ x > y \to G'(x-y)(\mathsf{S}y)z \to G'(\mathsf{S}x)(\mathsf{S}y)z
\end{aligned}$$

We need the following inversion rules for G'.

Fact 24.4.2 (Inversion rules)

1. $G'0yz \to z = y$
2. $G'(\mathsf{S}x)0z \to z = \mathsf{S}x$
3. $x \le y \to G'(\mathsf{S}x)(\mathsf{S}y)z \to G'(\mathsf{S}x)(y-x)z$
4. $x > y \to G'(\mathsf{S}x)(\mathsf{S}y)z \to G'(x-y)(\mathsf{S}y)z$

Proof The rules are intuitively obvious. One just looks at the unique rule that can establish the premise for G'. Formal proofs of the inversion rules can be obtained with an inversion operator as follows:

$$\begin{aligned}
\forall xyz\,\forall a^{\mathsf{G}'xyz}.\ &\textsc{match}\ x, y\ \textsc{return}\ \mathsf{G}'xyz \to \mathbb{P} \\
&[\,0,\ y \Rightarrow z = y \\
&\mid \mathsf{S}x,\ 0 \Rightarrow z = \mathsf{S}x \\
&\mid \mathsf{S}x,\ \mathsf{S}y \Rightarrow \textsc{if}\ \ulcorner x \le y \urcorner\ \textsc{then}\ \mathsf{G}'\,(\mathsf{S}x)(y-x)z\ \textsc{else}\ \mathsf{G}'\,(x-y)(\mathsf{S}y)z \\
&]\ a
\end{aligned}$$

■

Fact 24.4.3 (Functionality) G' is functional.

Proof We show $G'xyz \to G'xyz' \to z = z'$ by induction on $G'xyz$ and inversion on $G'xyz'$. All 4 cases are straightforward. ■

Fact 24.4.4 (Commutativity) $G'xyz \to G'yxz$.

Proof By induction on $G'xyz$. All cases but the third case are straightforward. We assume $x \le y$ and $G'(\mathsf{S}x)(y-x)z$, and prove $G'(\mathsf{S}y)(\mathsf{S}x)z$. The inductive hypothesis gives us $G'(y-x)(\mathsf{S}x)z$. If $x < y$, the claim follows with G'_4. If $x = y$, the claim is $G'(\mathsf{S}x)(\mathsf{S}x)z$ and the inductive hypothesis is $G'0(\mathsf{S}x)z$. By inversion we have $z = \mathsf{S}x$. Now the claim follows with G'_3 and G'_2. ■

Fact 24.4.5 G' is a functional gcd predicate.

Proof Given Facts 24.4.3 and 24.4.4, it remains to verify that G' satisfies the subtraction rule. Let $x \le y$ and $G'x(y-x)z$. We show $G'xyz$ by case analysis. If $x = 0$, the claim follows by inversion and G'_1. The case $x > 0$ and $y = 0$ is contradictory. The final case follows with G'_3. ■

Fact 24.4.6 G is a functional gcd predicate.

Proof Facts 24.4.5, 24.1.4 (1), and 24.1.3. ■

Exercise 24.4.7 Convince yourself that the index induction principle for G' can be obtained with deterministic gcd recursion (Exercise 24.2.3).

24.5 GCD Functions

A **gcd function** is a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfying the following equations:

1. $f0y = y$
2. $fxy = fyx$
3. $fxy = fx(y-x)$ if $x \le y$

Fact 24.5.1 Every gcd function respects every gcd predicate.

Proof By gcd induction (Fact 24.2.1). Straightforward. ■

Fact 24.5.2 A function is a gcd function if it respects a functional gcd predicate.

Proof Straightforward. No induction needed. ■

Fact 24.5.3 There is a gcd function.

Proof Follows with Facts 24.1.7, 24.4.5, and 24.5.2. ■

Fact 24.5.4 All gcd functions agree.

Proof By gcd induction (Fact 24.2.1). Straightforward. ■

Exercise 24.5.5 (Greatest common divisors with remainders)
Let $r^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ be a function satisfying the equation

$$rxy = \begin{cases} x & \text{if } x \le y \\ r(x - \mathsf{S}y)y & \text{if } x > y \end{cases}$$

a) Show that every gcd function f satisfies $fx(\mathsf{S}y) = f(\mathsf{S}y)(rxy)$.

b) Show that every function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfying the equations

$$\begin{aligned} fx0 &= x \\ fx(\mathsf{S}y) &= f(\mathsf{S}y)(rxy) \end{aligned}$$

is a gcd function.

c) Show that there is a function f satisfying the above equations. Do not use step-indexing.

24.6 Procedural Specification of GCD Functions

Consider the following equations for a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$:

$$\begin{aligned} f0y &= y \\ f(\mathsf{S}x)0 &= \mathsf{S}x \\ f(\mathsf{S}x)(\mathsf{S}y) &= \begin{cases} f(\mathsf{S}x)(y-x) & \text{if } x \le y \\ f(x-y)(\mathsf{S}y) & \text{if } x > y \end{cases} \end{aligned}$$

We will show that a function satisfies the equations if and only if it is a gcd function. Our main interest, however, will be in a step-indexed construction (§ 16.3) of a function satisfying the equations.

If we look at the equations, it is clear that they define a procedure, which computes by applying the equations from left to right. Given an application of the procedure, exactly one of the equations applies. We also observe that the procedure terminates for all arguments x and y since the sum $x + y$ of the two arguments is decreased upon recursion.

The recursion underlying the equations is not structural, which prevents us from taking the equations as defining equations for a function. In fact, it seems impossible to define a function f such that the specifying equations hold by computational equality. However, there are several methods for constructing a function satisfying the equations as propositional equations.

We define a function

$$\begin{aligned} &\Gamma : \ (\mathsf{N}\to\mathsf{N}\to\mathsf{N})\to\mathsf{N}\to\mathsf{N}\to\mathsf{N} \\ \Gamma f0y &:= y \\ \Gamma f(\mathsf{S}x)0 &:= \mathsf{S}x \\ \Gamma f(\mathsf{S}x)(\mathsf{S}y) &:= \begin{cases} f(\mathsf{S}x)(y-x) & \text{if } x \le y \\ f(x-y)(\mathsf{S}y) & \text{if } x > y \end{cases} \end{aligned}$$

capturing the procedural specification provided by the equations. We say that a function f **satisfies** Γ if $\forall xy.\ fxy = \Gamma fxy$. We will refer to Γ as a **procedural specification** or as a **step function**.

Fact 24.6.1 A function f satisfies Γ if and only if f satisfies the three specifying equations for f.

We leave the formal statement of the fact and its trivial proof as an exercise.

Fact 24.6.2 (Commutativity) If f satisfies Γ, then $\forall xy.\ fxy = fyx$.

Proof Let f satisfy Γ. Then $\forall xy.\ fxy = fyx$ follows by deterministic gcd induction (Exercise 24.2.3). Straightforward. ∎

Fact 24.6.3 A function satisfies Γ if and only if it is a gcd function.

Proof Both directions are straightforward. Direction $\rightarrow$ follows with commutativity (Fact 24.6.2). The other direction doesn't need a lemma. ∎

Fact 24.6.4 (Uniqueness) All functions satisfying Γ agree.

Proof Let f and g satisfy Γ. Then $\forall xy.\ fxy = gxy$ follows by deterministic gcd induction. Straightforward. ∎

Next we construct a function satisfying Γ. The trick is to define an auxiliary function with an extra argument called a **step index**. The step index serves as an upper bound for the remaining recursion steps. The recursion of the auxiliary function is on the step index which is initialized as $\mathsf{S}(x + y)$.

The necessary definitions are shown in Figure 24.1. We first show that the step index is handled correctly by g. Once more we use deterministic gcd induction.

Lemma 24.6.5 (Index independence) Let $m, n > x + y$. Then $gmxy = gnxy$.

Proof We prove $\forall xymn.\ gmxy = gnxy$ by deterministic gcd induction. Straightforward. ∎

Fact 24.6.6 (Existence) h satisfies Γ.

Proof We prove $g(\mathsf{S}(x + y))xy = \Gamma hxy$. We distinguish 4 cases. The cases $x = 0$ or $y = 0$ are straightforward. Otherwise we have $x = \mathsf{S}x'$, $y = \mathsf{S}y'$ and $x' \leq y'$ or $x' > y'$. We consider $x' \leq y'$, the other case is similar. We have to prove

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

which follows by Lemma 24.6.5. ∎

$$\begin{aligned}
&g : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\qquad\qquad g\,0\,\_\,\_ \;:=\; 0 \\
&\qquad\quad g\,(\mathsf{S}\_)0y \;:=\; y \\
&\quad g\,(\mathsf{S}\_)(\mathsf{S}x)0 \;:=\; \mathsf{S}x \\
&g\,(\mathsf{S}n)(\mathsf{S}x)(\mathsf{S}y) \;:=\; \begin{cases} gn(\mathsf{S}x)(y-x) & \text{if } x \le y \\ gn(x-y)(\mathsf{S}y) & \text{if } x > y \end{cases}
\end{aligned}$$

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

Figure 24.1: Step-indexed definition of a function satisfying Γ

Fact 24.6.7 h is a gcd function.

Proof Facts 24.6.6 and 24.6.3. ■

In summary, we have shown that the procedural specification Γ is unique and satisfiable, and that a function satisfies Γ if and only if it is a gcd function. The use of deterministic gcd induction was essential three times: for uniqueness, for commutativity, and for step index independence. Informally, we may say that deterministic gcd recursion models the termination of the recursion specified by Γ. Moreover, we may call the functions satisfying Γ **deterministic gcd functions** and interpret our results as saying that a function is a deterministic gcd function if and only if it is a gcd function. Note that deterministic gcd functions relate to gcd functions in a similar way the deterministic gcd predicate relates to gcd predicates. The notion of deterministic gcd functions matters since via step-indexing it provides for the construction of a gcd function. Similarly, the deterministic gcd predicate provides us with a functional gcd predicate.

25 Semi-Decidability and Markov's Principle

Computability theory 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. Many important problems are semi-decidable but not decidable. It turns out that semi-decidability has an elegant definition in type theory, and that Post's theorem is equivalent to Markov's principle.

We will see many uses of witness operators and pairing functions.

25.1 Preliminaries

Recall **boolean deciders** f for predicates p:

$$\mathsf{dec}\ p^{X \to \mathbb{P}}\ f^{X \to \mathsf{B}} \ := \ \forall x.\, px \longleftrightarrow fx = \mathbf{T}$$

We have two possibilities to express that a predicate p is decidable:[1]

- $\mathsf{ex}(\mathsf{dec}\, p)$ says that we *know* that there is a boolean decider for p.
- $\mathsf{sig}(\mathsf{dec}\, p)$ says that we *have* a concrete boolean decider for p.

Note that $\mathsf{sig}(\mathsf{dec}\, p)$ is stronger than $\mathsf{ex}(\mathsf{dec}\, p)$ since we have a function

$$\forall p.\ \mathsf{sig}(\mathsf{dec}\, p) \to \mathsf{ex}(\mathsf{dec}\, p)$$

but not necessarily a function for the converse direction. When we informally say that a predicate p is decidable we leave it to the context to determine whether the computational interpretation $\mathsf{sig}(\mathsf{dec}\, p)$ or the propositional interpretation $\mathsf{ex}(\mathsf{dec}\, p)$ is meant.

We will make frequent use of certifying deciders and their inter-translatability with boolean deciders (Fact 10.1.1).

Fact 25.1.1 $\forall p^{X \to \mathbb{P}}.\ \mathsf{sig}(\mathsf{dec}\, p) \Leftrightarrow \forall x.\, \mathcal{D}(px)$.

[1] Note that we have the computational equalities $\mathsf{ex}(\mathsf{dec}\, p) = \exists f.\, \mathsf{dec}\, p f$ and $\mathsf{sig}(\mathsf{dec}\, p) = \Sigma f.\, \mathsf{dec}\, p f$.

For several results in this chapter, we will use an *existential witness operator for numbers* (Chapter 19):

$$\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{sig}(\mathsf{dec}\,p) \to \mathsf{ex}\,p \to \mathsf{sig}\,p$$

We also need *arithmetic pairing functions* (Chapter 7)

$$\begin{aligned} \langle \_, \_ \rangle &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ \pi_1 &: \mathsf{N} \to \mathsf{N} \\ \pi_2 &: \mathsf{N} \to \mathsf{N} \end{aligned}$$

satisfying $\pi_1\langle x, y\rangle = x$ and $\pi_2\langle x, y\rangle = y$ for all x, y.

We call functions $\mathsf{N} \to \mathsf{B}$ **tests** and say that a test f is **satisfiable** if $\exists n.\ fn = \mathbf{T}$:

$$\mathsf{tsat}\, f^{\mathsf{N}\to\mathsf{B}} \;:=\; \exists n.\ fn = \mathbf{T}$$

Tests may be thought of as decidable predicates on numbers. Tests will play a major role in our development of semi-decidability. The witness operator ensures that test satisfiability is computational.

Fact 25.1.2 $\forall f^{\mathsf{N}\to\mathsf{B}}.\ (\exists n.\ fn = \mathbf{T}) \Leftrightarrow (\Sigma n.\ fn = \mathbf{T})$.

Recall the notion of a **stable proposition**:

$$\mathsf{stable}\, P^{\mathbb{P}} \;:=\; \neg\neg P \to P$$

Note that stable proposition satisfy a weak form of excluded middle providing for proof by contradiction. We will often tacitly exploit that stability is *extensional* (i.e. invariant under propositional equivalence).

Fact 25.1.3 (Extensionality) $(P \longleftrightarrow Q) \;\to\; \mathsf{stable}\, P \to \mathsf{stable}\, Q$.

Markov's principle says that satisfiability of tests is stable:

$$\mathsf{MP} \;:=\; \forall f^{\mathsf{N}\to\mathsf{B}}.\ \mathsf{stable}(\mathsf{tsat}\, f)$$

MP is a consequence of excluded middle that is weaker than excluded middle. It is know that computational type theory does not prove MP.

Fact 25.1.4 (MP characterisation)
MP holds if and only if satisfiability of decidable predicates on numbers is stable: $\mathsf{MP} \longleftrightarrow \forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{ex}(\mathsf{dec}\,p) \to \mathsf{stable}(\mathsf{ex}\,p)$.

Proof Decidable predicates on numbers are like tests. We leave a detailed proof as exercise. ■

Exercise 25.1.5 Show that excluded middle implies MP.

Exercise 25.1.6 Prove $\mathsf{MP} \longleftrightarrow \forall f^{\mathsf{N}\to\mathsf{B}}.\ \neg(\forall n.\ fn = \mathbf{F}) \to \mathsf{tsat}\, f$.

Exercise 25.1.7 Give a function $\forall f^{\mathsf{N}\to\mathsf{B}}.\ \mathsf{tsat}\, f \to \Sigma n.\ fn = \mathbf{T}$.

Exercise 25.1.8 Prove $\mathsf{MP} \Leftrightarrow \forall f^{\mathsf{N}\to\mathsf{B}}.\ \neg\neg\mathsf{tsat}\, f \to \Sigma n.\ fn = \mathbf{T}$.

25.2 Boolean Semi-Deciders

Boolean **semi-deciders** f for predicates p are defined as follows:

$$\mathsf{sdec}\ p^{X \to \mathbb{P}}\ f^{X \to \mathsf{N} \to \mathsf{B}} := \forall x.\ px \longleftrightarrow \mathsf{tsat}(fx)$$

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 admits a proof n of px if and only if px holds.

Fact 25.2.1 Decidable predicates are semi-decidable: $\mathsf{sig}(\mathsf{dec}\,p) \to \mathsf{sig}(\mathsf{sdec}\,p)$.

Proof Let f be a boolean decider p. Then $\lambda xn.fx$ is a semi-decider for p. ■

It turns out that we can strengthen a witness operator for decidable predicates on numbers to a witness operator for semi-decidable predicates on numbers using arithmetic pairing of numbers.

Fact 25.2.2 (Witness operator) $\forall p^{\mathsf{N} \to \mathbb{P}}.\ \mathsf{sig}(\mathsf{sdec}\,p) \to \mathsf{ex}\,p \to \mathsf{sig}\,p$.

Proof Let f be a semi decider for a satisfiable predicate p. Then

$$\lambda n.\ f(\pi_1 n)(\pi_2 n) = \mathbf{T}$$

is a decidable and satisfiable predicate on numbers. Thus a witness operator for numbers gives us an n such that $f(\pi_1 n)(\pi_2 n) = \mathbf{T}$. We have $p(\pi_1 n)$. ■

Fact 25.2.3 (Semi-decidable equality)
Semi-decidable equality predicates are decidable:
$\forall X.\ \mathsf{sig}(\mathsf{sdec}(\mathsf{eq}\,X)) \to \mathsf{sig}(\mathsf{dec}(\mathsf{eq}))$.

Proof Let $f^{X \to X \to \mathsf{N} \to \mathsf{B}}$ satisfy $\forall xy^X.\ x = y \longleftrightarrow \exists n.\ fxyn = \mathbf{T}$. Assume x, y^X. With an existential witness operator for numbers we obtain k such that $fxxk = \mathbf{T}$. We now check $fxyk$. If $fxyk = \mathbf{T}$, we have $x = y$. If $fxyk = \mathbf{F}$, we have $x \neq y$. To see this, assume $fxyk = \mathbf{F}$ and $x = y$. Then $fxxk = \mathbf{F}$, which contradicts $fxxk = \mathbf{T}$. ■

Given the results of computability theory, no decider for tsat can be defined in Coq's type theory. We cannot expect a proof of this claim within Coq's type theory. On the other hand, there is a trivial semi-decider for tsat.

Fact 25.2.4 tsat is semi-decidable.

Proof $\lambda fn.\ fn$ is a semi-decider for tsat. ■

It turns out that under MP all semi-decidable predicates are stable. In fact, this property is also sufficient for MP since tsat is semi-decidable.

Fact 25.2.5 (MP characterisation)
MP holds if and only if semi-decidable predicates are pointwise stable:
$\mathsf{MP} \;\longleftrightarrow\; \forall X^{\mathbb{T}}\, \forall p^{X\to\mathbb{P}}.\ \mathsf{ex}(\mathsf{sdec}\, p) \to \forall x.\ \mathsf{stable}(px)$.

Proof Direction $\leftarrow$ holds since tsat is semi-decidable (Fact 25.2.4).
Direction $\rightarrow$. Let f be a semi-decider for p. We show that px is stable. We have $px \longleftrightarrow \exists n.\, fxn = \mathbf{T}$. Since $\exists n.\, fxn = \mathbf{T}$ is stable by MP, we have that px is stable. ■

Exercise 25.2.6 (Projection) Let $f^{X\to\mathsf{N}\to\mathsf{N}\to\mathsf{B}}$ be a semi-decider for $p^{X\to\mathsf{N}\to\mathbb{P}}$. Give a semi-decider for $\lambda x.\exists n.pxn$.

Exercise 25.2.7 (Skolem function) Let $R^{X\to\mathsf{N}\to\mathbb{P}}$ be a total relation (i.e., $\forall x\exists y.Rxy$) and let $f^{X\to\mathsf{N}\to\mathsf{N}\to\mathsf{B}}$ be a semi-decider for R. Give a function $g^{X\to\mathsf{N}}$ such that $\forall x.\ Rx(gx)$.

Exercise 25.2.8 Let $p^{X\to\mathbb{P}}$ be semi-decidable. Prove $\forall x.\ px \to \forall y.\ py + (y \neq x)$.
Hint: The proof is similar to the proof of Fact 25.2.3. Using the witness operator one obtains n such that $fxn = \mathbf{T}$ and then discriminates on fyn. In fact, Fact 25.2.3 is a consequence of the above result.

25.3 Certifying Semi-Deciders

Recall that boolean deciders are inter-translatable with certifying deciders, and that certifying deciders are technically convenient for many proofs. Following this design, we will now define semi-decisions and certifying semi-deciders. The idea for semi-decisions is implicit in boolean semi-deciders, which yield for x a test such that px holds if and only if the test is satisfiable. Following this idea, we define **semi-decision types** $\mathcal{S}(P)$ as follows:

$$\begin{aligned} \mathcal{S} &: \mathbb{P} \to \mathbb{T} \\ \mathcal{S}(P) &:= \Sigma f^{\mathsf{N}\to\mathsf{B}}.\ P \longleftrightarrow \mathsf{tsat}\, f \end{aligned}$$

We may say that a semi-decision for P is a test that is satisfiable if and only if P holds.

Fact 25.3.1 $\forall P^{\mathbb{P}}.\ \mathcal{D}(P) \to \mathcal{S}(P)$.

Proof If P holds, we choose the always succeeding test, otherwise the always failing test. ■

Fact 25.3.2 (Transport) $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ (P \longleftrightarrow Q) \to \mathcal{S}(P) \to \mathcal{S}(Q)$.

Fact 25.3.3 $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ \mathcal{S}(P) \to \mathcal{S}(Q) \to \mathcal{S}(P \wedge Q)$.

Proof Let f be the test for P and g be the test for Q. Then $\lambda n.\, f(\pi_1 n) \& g(\pi_2 n)$ is a test for $P \wedge Q$. Note the use of the pairing functions π_1 and π_2. ■

Fact 25.3.4 $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ \mathcal{S}(P) \to \mathcal{S}(Q) \to \mathcal{S}(P \vee Q)$.

Proof Let f be the test for P and g be the test for Q. Then $\lambda n. fn \mid gn$ is a test for $P \vee Q$. ■

Fact 25.3.5 $\forall P Q^{\mathbb{P}}.\ \ \mathcal{S}(P) \to \mathcal{S}(Q) \to (P \vee Q) \to (P + Q)$.

Proof Let f be the test for P and g be the test for Q. Then $\lambda n. fn \mid gn$ is a test for $P \vee Q$. Since we have $P \vee Q$, an existential witness operator for numbers gives us an n such that $fn \mid gn = \mathbf{T}$. Thus $(fn = \mathbf{T}) + (gn = \mathbf{T})$. If $fn = \mathbf{T}$, we have P. If $gn = \mathbf{T}$, we have Q. ■

Fact 25.3.6 $\mathcal{S}(\mathsf{tsat}\, f)$.

Proof Trivial. ■

Fact 25.3.7 (MP characterisation)
MP holds if and only if semi-decidable propositions are stable:
$\mathsf{MP} \ \longleftrightarrow\ \forall P^{\mathbb{P}}.\, \mathsf{S}(P) \to \mathsf{stable}(P)$.

Proof Direction $\to$. Let $P \longleftrightarrow \mathsf{tsat}\, f$. The claim $\mathsf{stable}(P)$ follows by extensionality and MP.

Direction $\leftarrow$. We show $\mathsf{stable}(\mathsf{tsat}\, f)$. By the assumption it suffices to show $\mathcal{S}(\mathsf{tsat}\, f)$. Trivial. ■

A **certifying semi-decider** for a predicate $p^{X \to \mathbb{P}}$ is a function $\forall x^X.\ \mathcal{S}(px)$. From a certifying semi-decider for p we can obtain a semi-decider for p by forgetting the proofs. Vice versa, we can construct from a semi-decider and its correctness proof a certifying semi-decider.

Fact 25.3.8
We can translate between semi-deciders and certifying semi-deciders:
$\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ \mathsf{sig}(\mathsf{sdec}\, p) \ \Leftrightarrow\ \forall x. \mathcal{S}(px)$.

Proof Direction $\Rightarrow$. We assume $\forall x.\ px \longleftrightarrow \exists n. fnx = \mathbf{T}$ and x^X and obtain $\mathcal{S}(px)$ with fx as test.

Direction $\Leftarrow$. We assume $g^{\forall x. \mathcal{S}(px)}$ and use $fx := \pi_1(gx)$ as semi-decider. It remains to show $px \longleftrightarrow \exists n. fxn = \mathbf{T}$, which is straightforward. ■

We offer another characterisation of semi-decisions.

Fact 25.3.9 $\forall P^{\mathbb{P}}.\ \mathcal{S}(P) \Leftrightarrow \Sigma f^{\mathsf{N}\to\mathcal{O}(P)}.\ P \to \exists n.\ fn \neq \emptyset$.

Proof Direction $\Rightarrow$. Let g be the test for P. Then

$$fn \ := \ \text{IF } gn \text{ THEN } {}^{\circ}\ulcorner P \urcorner \text{ ELSE } \emptyset$$

is a function as required.

Direction $\Leftarrow$. Let $P \to \exists n.\ fn \neq \emptyset$. Then

$$gn \ := \ \text{IF } \ulcorner fn = \emptyset \urcorner \text{ THEN } \mathbf{F} \text{ ELSE } \mathbf{T}$$

is a test for P. ■

It turns out that from a decider for tsat we can get a function translating semi-decisions into decisions, and vice versa.

Fact 25.3.10 $\mathsf{sig}(\mathsf{dec}(\mathsf{tsat})) \Leftrightarrow \forall P^{\mathbb{P}}.\ \mathcal{S}(P) \to \mathcal{D}(P)$.

Proof Direction $\Leftarrow$ follows since f is a test for $\mathcal{S}(\mathsf{tsat}\, f)$. For direction $\Rightarrow$ we assume $P \longleftrightarrow \mathsf{tsat}\, f$ and show $\mathcal{D}(P)$. By the primary assumption we have either $\mathsf{tsat}\, f$ or $\neg\mathsf{tsat}\, f$. Thus $\mathcal{D}(P)$. ■

Exercise 25.3.11 Prove $\forall P^{\mathbb{P}}.\ (P \vee \neg P) \to \mathcal{S}(P) \to \mathcal{S}(\neg P) \to \mathcal{D}(P)$.

Exercise 25.3.12 Prove $\mathsf{MP} \Leftrightarrow (\forall P^{\mathbb{P}}.\ \mathcal{D}(P) \Leftrightarrow \mathcal{S}(P) \times \mathcal{S}(\neg P))$.

25.4 Post Operators

We will consider **Post operators**,[2] which are functions of the type

$$\mathsf{Post} \ := \ \forall P^{\mathbb{P}}.\ \mathcal{S}(P) \to \mathcal{S}(\neg P) \to \mathcal{D}(P)$$

We will show that MP gives us a Post operator, and that the existence of a Post operator implies MP.

Fact 25.4.1 $\mathsf{MP} \to \mathsf{Post}$.

Proof Assume MP. Let f be a test for P and g be a test for $\neg P$. We show $\mathcal{D}(P)$. Let $hn := fn \mid gn$. It suffices to show $\Sigma n.\ hn = \mathbf{T}$. Since we have a witness operator and MP, we assume $H : \neg\mathsf{tsat}\, h$ and derive a contradiction. To do so, we show $\neg P$ and $\neg\neg P$. If we assume either P or $\neg P$, we have $\mathsf{tsat}\, h$ contradicting H. ■

[2]Post operators are named after Emil Post, who first showed that predicates are decidable if they are semi-decidable and co-semi-decidable.

Fact 25.4.2 Post → MP.

Proof We assume Post and $H : \neg\neg\mathsf{tsat}\, f$ and show $\mathsf{tsat}\, f$. It suffices to show $\mathcal{D}(\mathsf{tsat}\, f)$. Using Post it suffices to show $\mathcal{S}(\mathsf{tsat}\, f)$ and $\mathcal{S}(\neg\mathsf{tsat}\, f)$. $\mathcal{S}(\mathsf{tsat}\, f)$ holds with f as test, and $\mathcal{S}(\neg\mathsf{tsat}\, f)$ holds with $\lambda\_.\mathbf{F}$ as test. ■

Theorem 25.4.3 (MP Characterisation) MP ⇔ Post.

Proof Facts 25.4.1 and 25.4.2. ■

We define the **complement** of predicates $p^{X\to\mathbb{P}}$ as $\overline{p} := \lambda x.\neg px$.

Corollary 25.4.4 Given MP, a predicate is decidable if and only if it is semi-decidable and co-semi-decidable:
$\mathsf{MP} \to \forall p^{X\to\mathbb{P}}.\ \mathsf{sig}(\mathsf{dec}\, p) \Leftrightarrow \mathsf{sig}(\mathsf{sdec}\, p) \times \mathsf{sig}(\mathsf{sdec}\, \overline{p})$.

Proof Direction ⇒ doesn't need MP and follows with Fact 25.2.1 and $\mathsf{sig}(\mathsf{dec}\, p) \to \mathsf{sig}(\mathsf{dec}\, \overline{p})$. For direction ⇐ we use Fact 25.1.1 and obtain $\mathcal{D}(px)$ from $\mathcal{S}(px)$ and $\mathcal{S}(\neg px)$ using Facts 25.4.1 and 25.3.8. ■

25.5 Enumerators

We define **enumerators** f for predicates p as follows:

$$\mathsf{enum}\ p^{X\to\mathbb{P}}\ f^{\mathsf{N}\to\mathcal{O}(X)} \ :=\ \forall x.\ px \longleftrightarrow \exists n.\ fn = {}^\circ x$$

We will show that for predicates on data types (§ 28.4) one can freely translate between enumerators and semi-deciders.

We define **equality deciders** as follows:

$$\mathsf{eqdec}\ X^{\mathbb{T}}\ f^{X\to X\to\mathsf{B}} \ :=\ \forall xy^X.\ x = y \longleftrightarrow fxy = \mathbf{T}$$

Fact 25.5.1 $\forall p^{X\to\mathbb{P}}.\ \mathsf{sig}(\mathsf{eqdec}\, X) \to \mathsf{sig}(\mathsf{enum}\, p) \to \mathsf{sig}(\mathsf{sdec}\, p)$.

Proof Let d be an equality decider for X and f be an enumerator for p. Then

$$\lambda xn.\ \text{IF}\ \ulcorner fn = {}^\circ x \urcorner\ \text{THEN}\ \mathbf{T}\ \text{ELSE}\ \mathbf{F}$$

is a semi-decider for p. ■

For the other direction, we need an enumerator for the base type X. To ease the statement, we define a predicate as follows:

$$\mathsf{enum}\ X^{\mathbb{T}}\ f^{\mathsf{N}\to\mathcal{O}(X)} \ :=\ \forall x\, \exists n.\ fn = {}^\circ x$$

Fact 25.5.2 $\forall p^{X \to \mathbb{P}}.\ \mathsf{sig}(\mathsf{enum}\,X) \to \mathsf{sig}(\mathsf{sdec}\,p) \to \mathsf{sig}(\mathsf{enum}\,p)$.

Proof Let g be an enumerator for X and f be a semi-decider for p. We define an enumerator h for p interpreting its argument as a pair consisting of a number for g and an index for f:

$$hn := \begin{cases} {}^\circ x & \text{if } g(\pi_1 n) = {}^\circ x \wedge fx(\pi_2 n) = \mathbf{T} \\ \emptyset & \text{otherwise} \end{cases}$$

is an enumerator for p. ∎

Recall that a type is a data type if and only if it has an enumerator and an equality decider (Fact 28.4.5).

Corollary 25.5.3 One can translate between enumerators and semi-deciders for predicates on data types.

Fact 25.5.4 Decidable predicates on enumerable types are enumerable:
$\forall p^{X \to \mathbb{P}}.\ \mathsf{sig}(\mathsf{dec}\,p) \to \mathsf{sig}(\mathsf{enum}\,X) \to \mathsf{sig}(\mathsf{enum}\,p)$.

Fact 25.5.5 (MP characterisation)
MP holds if and only if satisfiability of enumerable predicates is stable:
$\mathsf{MP} \longleftrightarrow \forall X^{\mathbb{T}}\ \forall p^{X \to \mathbb{P}}.\ \mathsf{ex}(\mathsf{enum}\,p) \to \mathsf{stable}(\mathsf{ex}\,p)$.

Proof Direction $\to$. Let f be an enumerator for p. Then

$$\mathsf{ex}\,p \longleftrightarrow \exists nx.\ fn = {}^\circ x$$

Since $\lambda n.\exists x.fn = {}^\circ x$ is decidable, stability of $\mathsf{ex}\,p$ follows with Fact 25.1.4.

Direction $\leftarrow$. By Fact 25.1.4 it suffices to show that satisfiability of decidable predicates on numbers is stable. Follows since decidable predicates on numbers are enumerable (Fact 25.5.4). ∎

Fact 25.5.6 (MP characterisation)
MP holds if and only if enumerable predicates on discrete types are pointwise stable:
$\mathsf{MP} \longleftrightarrow \forall X^{\mathbb{T}}\ \forall p^{X \to \mathbb{P}}.\ \mathsf{ex}(\mathsf{eqdec}\,X) \to \mathsf{ex}(\mathsf{enum}\,p) \to \forall x.\ \mathsf{stable}(px)$.

Proof Direction $\to$. We assume MP, a discrete type X, a predicate $p^{X \to \mathbb{P}}$, and an enumerator f for p. It suffices to show that $\exists n.\ fn = {}^\circ x$ is stable. Since we have an equality decider for X, we have a decider for $\lambda n.fn = {}^\circ x$. Thus $\exists n.\ fn = {}^\circ x$ is stable by Fact 25.1.4 and MP.

Direction $\leftarrow$. We show $\mathsf{stable}(\mathsf{tsat}\,f)$ for $f^{\mathsf{N} \to \mathsf{B}}$. We define $p(k^{\mathsf{N}}) := \mathsf{tsat}\,f$. By the primary assumption it suffices to show $\mathsf{ex}(\mathsf{enum}\,p)$. By Fact 25.5.2 it suffices to give a semi-decider for p. Clearly, $\lambda kn.fn$ is a semi-decider for p. ∎

Exercise 25.5.7 Let f be an enumerator for $p^{X\to\mathbb{P}}$ and g be an enumerator for $q^{X\to\mathbb{P}}$.

1. Give an enumerator for $\lambda x.\, px \vee qx$.
2. Give an enumerators for $\lambda x.\, px \wedge qx$ assuming X is discrete.

Exercise 25.5.8 (Projections) Let $f^{\mathsf{N}\to\mathcal{O}(X\times Y)}$ be an enumerator for $p^{X\to Y\to\mathbb{P}}$. Give enumerators for the projections $\lambda x.\exists y.pxy$ and $\lambda y.\exists x.pxy$.

Exercise 25.5.9 (Skolem functions) Let $R^{X\to Y\to\mathbb{P}}$ be a total relation (i.e., $\forall x\exists y.Rxy$). Moreover, let $f^{\mathsf{N}\to\mathcal{O}(X\times Y)}$ be an enumerator for R and $d^{X\to X\to\mathsf{B}}$ be a equality decider for X. Give a function $g^{X\to Y}$ such that $\forall x.\, Rx(gx)$.

25.6 Reductions

Computability theory employs so-called many-one reductions to transport decidability and undecidability results between problems. We model problems as predicates and many-one reductions as functions. We define **reductions** from a predicate p to a predicate q as follows:

$$\mathsf{red}\;\; p^{X\to\mathbb{P}}\; q^{Y\to\mathbb{P}}\; f^{X\to Y} \;:=\; \forall x.\, p(x) \longleftrightarrow q(fx)$$

Fact 25.6.1
Decidability and undecidability transport through reductions as follows:

1. $\mathsf{sig}\,(\mathsf{red}\,pq) \to \mathsf{sig}\,(\mathsf{dec}\,q) \to \mathsf{sig}\,(\mathsf{dec}\,p)$
2. $\mathsf{sig}\,(\mathsf{red}\,pq) \to \mathsf{sig}\,(\mathsf{sdec}\,q) \to \mathsf{sig}\,(\mathsf{sdec}\,p)$
3. $\mathsf{red}\,pqf \to (\forall y.\mathcal{D}(qy)) \to (\forall x.\mathcal{D}(px))$
4. $\mathsf{red}\,pqf \to (\forall y.\, \mathcal{S}(qy)) \to (\forall x.\, \mathcal{S}(px))$
5. $\mathsf{ex}\,(\mathsf{red}\,pq) \to \mathsf{ex}\,(\mathsf{dec}\,q) \to \mathsf{ex}\,(\mathsf{dec}\,p)$
6. $\mathsf{ex}\,(\mathsf{red}\,pq) \to \neg\mathsf{ex}\,(\mathsf{dec}\,p) \to \neg\mathsf{ex}\,(\mathsf{dec}\,q)$
7. $\mathsf{ex}\,(\mathsf{red}\,pq) \to \mathsf{ex}\,(\mathsf{sdec}\,q) \to \mathsf{ex}\,(\mathsf{sdec}\,p)$
8. $\mathsf{ex}\,(\mathsf{red}\,pq) \to \neg\mathsf{ex}\,(\mathsf{sdec}\,p) \to \neg\mathsf{ex}\,(\mathsf{sdec}\,q)$

Proof 1. Let $\mathsf{red}\,pqf$ and $\mathsf{dec}\,qg$. Then $\lambda x.g(fx)$ is a boolean decider for p.

2. Let $\mathsf{red}\,pqf$ and $\mathsf{sdec}\,qg$. Then $\lambda x.g(fx)$ is a semi decider for p.
3. Follows from (1) with Fact 25.1.1.
4. Follows from (2) with Fact 25.3.8.
5. Straightforward with (1).
6. Straightforward with (5).
7. Straightforward with (2).

8. Straightforward with (6). ∎

Fact 25.6.2 Stability transports through reductions:
$\mathsf{ex}\,(\mathsf{red}\,pq) \to (\forall y.\ \mathsf{stable}(qy)) \to (\forall x.\ \mathsf{stable}(px))$.

Fact 25.6.3 A predicate is semi-decidable if and only if it reduces to tsat:
$\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ (\forall x.\ \mathcal{S}(px)) \Leftrightarrow \mathsf{sig}\,(\mathsf{red}\,p\,\mathsf{tsat})$.

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 25.6.4 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 25.6.5 Prove $\mathsf{red}\,p\,q\,f \to \mathsf{red}\,\overline{q}\,\overline{p}\,f$.

25.7 Summary of Markov Characterisations

We have established many different equivalent characterisations of Markov's principle making connections between tests, deciders, semi-deciders, enumerators, and semi-decisions. Most of the characterisations use the notion of stability. The following fact collects prominent characterisations of Markov's principle we have considered in this chapter.

Fact 25.7.1 (Markov equivalences) The following types are equivalent:

1. Satisfiability of tests is stable.
2. Satisfiability of decidable predicates on numbers is stable.
3. Satisfiability of semi-decidable predicates is stable.
4. Satisfiability of enumerable predicates is stable.
5. Semi-decidable predicates are pointwise stable.
6. Enumerable predicates on discrete types are pointwise stable.
7. Semi-decidable propositions are stable.
8. $\forall P^{\mathbb{P}}.\ \mathcal{S}(P) \to \mathcal{S}(\neg P) \to \mathcal{D}(P)$.

Exercise 25.7.2 Make sure you can prove equivalent the characterisations of Markov's principle stated in Fact 25.7.1. Start by writing down formally the characterisations stated informally.

Notes

The chapter originated with Forster et al. [8]. Andrej Dudenhefner and Yannick Forster contributed nice facts about semi-deciders. Forster et al. [7] certify a reduction from the halting problem for Turing machines (HTM) to the Post correspondence problem (PCP) (§ 22.6) in Coq. Thus PCP is undecidable if HTM is undecidable. We believe that Coq's type theory is consistent with assuming that HTM is undecidable. One can show in Coq's type theory that there is no Turing machine deciding HTM.

What we have developed here is a little bit of synthetic computability theory in Coq's type theory. We have assumed all definable functions as computable, which is in contrast to conventional computability theory, where computable functions are functions definable in some model of computation (i.e., Turing machines). We also mention that in conventional computability theory computable functions are restricted to specific data types like strings or numbers. A main advantage of synthetic computability theory is that all constructions can be carried out rigorously, which is practically impossible if Turing machines are used as model of computation.

26 Aczel Trees and Hierarchy Theorems

Aczel trees are wellfounded trees where each node comes with a type and a function fixing the subtree branching. Aczel trees were conceived by Peter Aczel [2] as a representation of set-like structures in type theory. Aczel trees are accommodated with inductive type definitions featuring a single value constructor and higher-order recursion.

We discuss the *dominance condition*, a restriction on inductive type definitions ensuring predicativity of nonpropositional universes. Using Aczel trees, we will show an important foundational result: No universe embeds into one of its types. From this hierarchy result we obtain that proof irrelevance is a consequence of excluded middle, and that omitting the elimination restriction in the presence of the impredicative universe of propositions results in inconsistency.

26.1 Inductive Types for Aczel Trees

We define an inductive type providing **Aczel trees**:

$$\mathcal{T} : \mathbb{T} \ ::= \ \mathsf{T}\,(X : \mathbb{T},\, X \to \mathcal{T})$$

There is an important constraint on the universe levels of the two occurrences of $\mathbb{T}$ we will discuss later. We see a tree $\mathsf{T}\,X f$ as a tree taking all trees fx as (immediate) **subtrees**, where the edges to the subtrees are labelled with the values of X. We clarify the idea behind Aczel trees with some examples. The term

$$\mathsf{T}\,\bot\,(\lambda a.\,\text{MATCH}\ a\ [\,])$$

describes an **atomic tree** not having subtrees. Given two trees t_1 and t_2, the term

$$\mathsf{T}\,\mathsf{B}\,(\lambda b.\,\text{MATCH}\ b\ [\,\mathbf{T} \Rightarrow t_1 \mid \mathbf{F} \Rightarrow t_2])$$

describes a tree having exactly t_1 and t_2 as subtrees where the boolean values are used as labels. The term

$$\mathsf{T}\,\mathsf{N}\,(\lambda\_.\,\mathsf{T}\bot\,(\lambda h.\,\text{MATCH}\ h\ [\,]))$$

describes an **infinitely branching tree** that has a subtree for every number. All subtrees of the infinitely branching tree are equal (to the atomic tree).

Consider the term

$$\mathsf{T}\,\mathcal{T}\;(\lambda s.s)$$

which seems to describe a **universal tree** having every tree as subtree. It turns out that the term for the universal tree does not type check since there is a universe level conflict. First we note that Coq's type theory admits the definition

$$\mathcal{T} : \mathbb{T}_i \;::=\; \mathsf{T}\,(X : \mathbb{T}_j,\; X \to \mathcal{T})$$

only if $i > j$. This reflects a restriction on inductive definitions we have not discussed before. We speak of the **dominance condition**. In its general form, the dominance condition says that the type of every value constructor (without the parameter prefix) must be a member of the universe specified for the type constructor. The dominance condition admits the above definition for $i > j$ since then $\mathbb{T}_j : \mathbb{T}_i$, $X : \mathbb{T}_i$, and $\mathcal{T} : \mathbb{T}_i$ and hence

$$(\forall X^{\mathbb{T}_j}.\, (X \to \mathcal{T}) \to \mathcal{T}) \;:\; \mathbb{T}_i$$

using the universe rules from § 4.2. For the reader's convenience we repeat the rules for universes

$$\begin{array}{l} \mathbb{T}_1 : \mathbb{T}_2 : \mathbb{T}_3 : \cdots \\ \mathbb{P} \subseteq \mathbb{T}_1 \subseteq \mathbb{T}_2 \subseteq \mathbb{T}_3 \subseteq \cdots \\ \mathbb{P} : \mathbb{T}_2 \end{array}$$

and function types

$$\frac{\vdash u : U \qquad x : u \vdash v : U}{\vdash \forall x^u.v : U} \qquad\qquad \frac{\vdash u : U \qquad x : u \vdash v : \mathbb{P}}{\vdash \forall x^u.v \;:\; \mathbb{P}}$$

here. The variable U ranges over the computational universes $\mathbb{T}_i$. The first rule says that every computational universe is closed under taking function types. The second rule says that the universe $\mathbb{P}$ enjoys a stronger closure property known as impredicativity.

Note that the term for the universal tree $\mathsf{T}\,\mathcal{T}\;(\lambda s.s)$ does not type check since we do not have $\mathcal{T} : \mathbb{T}_j$ for $i > j$.

Exercise 26.1.1 The dominance condition for inductive type definitions requires that the types of the value constructors are in the target universe of the type constructor, where the types of the value constructor are considered *without* the parameter prefix. That the parameter prefix is not taken into account ensures that the universes $\mathbb{T}_i$ are closed under the type constructors for pairs, options, and lists. Verify the following typings for lists:

$$\mathcal{L}(X : \mathbb{T}_i) : \mathbb{T}_i \;::=\; \mathsf{nil} \mid \mathsf{cons}\,(X, \mathcal{L}(X))$$

$$\begin{array}{rll}
\mathcal{L} : & \mathbb{T}_i \to \mathbb{T}_i & : \mathbb{T}_{i+1} \\
\mathsf{nil} : & \mathcal{L}(X) & : \mathbb{T}_i \quad (X : \mathbb{T}_i) \\
\mathsf{cons} : & X \to \mathcal{L}(X) \to \mathcal{L}(X) & : \mathbb{T}_i \quad (X : \mathbb{T}_i) \\
\mathsf{nil} : & \forall X^{\mathbb{T}_i}.\, \mathcal{L}(X) & : \mathbb{T}_{i+1} \\
\mathsf{cons} : & \forall X^{\mathbb{T}_i}.\, X \to \mathcal{L}(X) \to \mathcal{L}(X) & : \mathbb{T}_{i+1}
\end{array}$$

Write down an analogous table for pairs and options.

26.2 Propositional Aczel Trees

We now note that the definition

$$\mathcal{T}_\mathsf{p} : \mathbb{P} \;::=\; \mathsf{T_p}\,(X : \mathbb{P},\; X \to \mathcal{T}_\mathsf{p})$$

of the type of **propositional Aczel trees** satisfies the dominance condition since the type of the constructor $\mathsf{T_p}$ is in $\mathbb{P}$ by the impredicativity of the universe $\mathbb{P}$:

$$(\forall X^U.\, (X \to \mathcal{T}_\mathsf{p}) \to \mathcal{T}_\mathsf{p}) \;:\; \mathbb{P}$$

Moreover, the term for the universal tree

$$u_\mathsf{p} := \; \mathsf{T_p}\,\mathcal{T}_\mathsf{p}\,(\lambda s.s)$$

does type check for propositional Aczel trees. So there is a **universal propositional Aczel tree**.

The universal propositional Aczel tree u_p is paradoxical in that it conflicts with our intuition that all values of an inductive type are wellfounded. A value of an inductive type is *wellfounded* if descending to a subvalue through a recursion in the type definition always terminates. Given that reduction of recursive functions is assumed to be terminating, one would expect that values of inductive types are wellfounded. However, the universal propositional Aczel tree $\mathsf{T_p}\,\mathcal{T}_\mathsf{p}\,(\lambda s.s)$ is certainly not wellfounded. So we have to adopt the view that because of the impredicativity of the universe $\mathbb{P}$ certain recursive propositional types do admit non-wellfounded values. This does not cause harm since the elimination restriction reliably prevents recursion on non-wellfounded values.

We remark that there are recursive propositional types providing for functional recursion. A good example are the linear search types for the existential witness operator (§ 19.1). It seems that the values of computational propositions are always wellfounded.

26.3 Subtree Predicate and Wellfoundedness

We will consider **computational Aczel trees** at the lowest universe level

$$\mathcal{T} : \mathbb{T}_2 \ ::= \ \mathsf{T}\,(X : \mathbb{T}_1,\ X \to \mathcal{T})$$

and propositional Aczel trees

$$\mathcal{T}_{\mathsf{p}} : \mathbb{P} \ ::= \ \mathsf{T}_{\mathsf{p}}\,(X : \mathbb{P},\ X \to \mathcal{T}_{\mathsf{p}})$$

as defined before. We reserve the letters s and t for Aczel trees.

To better understand the situation, we define a **subtree predicate** for computational Aczel trees:

$$\begin{aligned} &\in :\ \mathcal{T} \to \mathcal{T} \to \mathbb{P} \\ &s \in \mathsf{T}Xf \ := \ \exists x.\ fx = s \end{aligned}$$

Remarkably, the elimination restriction prevents us from defining an analogous subtree predicate for propositional Aczel trees (since the return type is not a proposition but the universe $\mathbb{P}$).

For computational Aczel trees we can prove $\forall s.\ s \notin s$, which disproves the existence of a universal tree. We will prove $\forall s.\ s \notin s$ by induction on s.

Definition 26.3.1 (Eliminator for computational Aczel trees)

$$\begin{aligned} &\mathsf{E}_{\mathcal{T}} :\ \forall p^{\mathcal{T}\to\mathbb{T}}.\ (\forall Xf.\ (\forall x.\ p(fx)) \to p(\mathsf{T}Xf)) \to \forall s.\, ps \\ &\mathsf{E}_{\mathcal{T}}\, p\, F\, (\mathsf{T}Xf) \ := \ FXf(\lambda x.\, \mathsf{E}_{\mathcal{T}}\, p\, F\, (fx)) \end{aligned}$$

Fact 26.3.2 (Irreflexivity) $\forall s^{\mathcal{T}}.\, s \notin s$.

Proof By induction on s (using $\mathsf{E}_{\mathcal{T}}$) it suffice to show $\mathsf{T}Xf \notin \mathsf{T}Xf$ given the inductive hypothesis $\forall x.\ fx \notin fx$. It suffices to show for every x^X that $fx = \mathsf{T}Xf$ is contradictory. Since $fx = \mathsf{T}Xf$ implies $fx \in fx$, we have a contradiction with the inductive hypothesis. ∎

For propositional Aczel trees we can prove that a subtree predicate $R^{\mathcal{T}_{\mathsf{p}}\to\mathcal{T}_{\mathsf{p}}\to\mathbb{P}}$ such that

$$R\, s\, (\mathsf{T}_{\mathsf{p}}\, Xf) \longleftrightarrow \exists x.\ fx = s$$

does not exist. This explains why the existence of the universal propositional Aczel tree does not lead to a proof of falsity.

Definition 26.3.3 (Eliminator for propositional Aczel trees)

$$\begin{aligned} &\mathsf{E}_{\mathcal{T}_{\mathsf{p}}} :\ \forall p^{\mathcal{T}_{\mathsf{p}}\to\mathbb{P}}.\ (\forall Xf.\ (\forall x.\ p(fx)) \to p(\mathsf{T}_{\mathsf{p}}\, Xf)) \to \forall s.\, ps \\ &\mathsf{E}_{\mathcal{T}_{\mathsf{p}}}\, p\, F\, (\mathsf{T}_{\mathsf{p}}\, Xf) \ := \ FXf(\lambda x.\, \mathsf{E}_{\mathcal{T}_{\mathsf{p}}}\, p\, F\, (fx)) \end{aligned}$$

Fact 26.3.4 $\neg\exists R^{\mathcal{T}_p \to \mathcal{T}_p \to \mathbb{P}}.\ \forall sXf.\ Rs(\mathsf{T_p}\,Xf) \longleftrightarrow \exists x.\ fx = s.$

Proof Let $R^{\mathcal{T}_p \to \mathcal{T}_p \to \mathbb{P}}$ be such that $\forall sXf.\ Rs(\mathsf{T_p}\,Xf) \longleftrightarrow \exists x.\ fx = s$. We derive a contradiction. Since the universal propositional Aczel tree $u_p := \mathsf{T_p}\,\mathcal{T}_p\,(\lambda s.s)$ satisfies Ruu, it suffices to prove $\forall s.\neg Rss$. We can do this by induction on s (using $\mathsf{E}_{\mathcal{T}_p}$) following the proof for computational Aczel trees (Fact 26.3.2). ∎

We summarize the situation as follows. Given a type

$$\mathcal{T} : U ::= \mathsf{T}(X : V,\ X \to \mathcal{T})$$

of Aczel trees, if we can define a *subtree predicate* $\in\, : \mathcal{T} \to \mathcal{T} \to \mathbb{P}$ such that

$$s \in \mathsf{T}Xf \quad\longleftrightarrow\quad \exists x.\ fx = s$$

we cannot define a *universal tree* $u \in u$. This works out such that for propositional Aczel trees we cannot define a subtree predicate (because of the elimination restriction) and for computational Aczel trees we cannot define a universal tree (because of the dominance restriction).

Exercise 26.3.5 Suppose you are allowed exactly one violation of the elimination restriction. Give a proof of falsity.

26.4 Propositional Hierarchy Theorem

A fundamental result about Coq's type theory says that the universe $\mathbb{P}$ of propositions cannot be embedded into a proposition, even if equivalent propositions may be identified. This important result was first shown by Thierry Coquand [6] in 1989 for a subsystem of Coq's type theory. We will prove the result for Coq's type theory by showing that an embedding as specified provides for the definition of a subtree predicate for propositional Aczel trees.

Theorem 26.4.1 (Coquand) There is no proposition $A^{\mathbb{P}}$ such that there exist functions $E^{\mathbb{P}\to A}$ and $D^{A\to\mathbb{P}}$ such that $\forall P^{\mathbb{P}}.\ D(E(P)) \longleftrightarrow P$.

Proof Let $A^{\mathbb{P}}$, $E^{\mathbb{P}\to A}$, $D^{A\to\mathbb{P}}$ be given such that $\forall P^{\mathbb{P}}.\ D(E(P)) \longleftrightarrow P$. By Fact 26.3.4 is suffices to show that

$$Rst \ := \ D\,(\textsc{match}\ t\ [\,\mathsf{T_p}\,Xf \Rightarrow E(\exists x.\ fx = s)])$$

satisfies $\forall sXf.\ Rs(\mathsf{T_p}\,Xf) \longleftrightarrow \exists x.\ fx = s$, which is straightforward. Note that the match in the definition of R observes the elimination restriction since the proposition $\exists x. fx = s$ is encoded with E into a proof of the proposition A. ∎

Exercise 26.4.2 Show $\neg\exists A^{\mathbb{P}}\,\exists E^{\mathbb{P}\to A}\,\exists D^{A\to\mathbb{P}}\,\forall P^{\mathbb{P}}.\ D(E(P)) = P$.

Exercise 26.4.3 Show $\forall P^{\mathbb{P}}.\ P \neq \mathbb{P}$.

26.5 Excluded Middle Implies Proof Irrelevance

With Coquand's theorem we can show that the law of excluded middle implies proof irrelevance (see § 4.3 for definitions). The key idea is that given a proposition with two different proofs we can define an embedding as excluded by Coquand's theorem. For the proof to go through we need the full elimination lemma for disjunctions (see Exercise 26.5.2).

Theorem 26.5.1 Excluded middle implies proof irrelevance.

Proof Let $d^{\forall X:\mathbb{P}.\, X\vee\neg X}$ and let a and b be proofs of a proposition A. We show $a = b$. Using excluded middle, we assume $a \neq b$ and derive a contradiction with Coquand's theorem. To do so, we define an encoding $E^{\mathbb{P}\to A}$ and a decoding $D^{A\to\mathbb{P}}$ as follows:

$$\begin{aligned} E(X) &:= \text{IF } dX \text{ THEN } a \text{ ELSE } b \\ D(c) &:= (a = c) \end{aligned}$$

It remains to show $D(E(X)) \longleftrightarrow X$ for all propositions X. By computational equality it suffices to show

$$(a = \text{IF } dX \text{ THEN } a \text{ ELSE } b) \longleftrightarrow X$$

By case analysis on $dX : X \vee \neg X$ using the full elimination lemma for disjunctions (Exercise 26.5.2) we obtain two proof obligations

$$\begin{aligned} X &\to (a = a \longleftrightarrow X) \\ \neg X &\to (a = b \longleftrightarrow X) \end{aligned}$$

which both follow by propositional reasoning (recall the assumption $a \neq b$). ∎

Exercise 26.5.2 Prove the full elimination lemma for disjunctions

$$\forall XY^{\mathbb{P}}\, \forall p^{X\vee Y\to\mathbb{P}}.\ (\forall x^X.\, p(\mathsf{L}\,x)) \to (\forall y^Y.\, p(\mathsf{R}\,y)) \to \forall a.\, pa$$

which is needed for the proof of Theorem 26.5.1.

26.6 Hierarchy Theorem for Computational Universes

We will now show that no computational universe embeds into one of its types. Note that by Coquand's theorem we already know that the universe $\mathbb{P}$ does not embed into one of its types.

We define a general **embedding predicate** $\mathcal{E}^{\mathbb{T}\to\mathbb{T}\to\mathbb{P}}$ for types:

$$\mathcal{E}XY \;:=\; \exists E^{X\to Y}\, \exists D^{Y\to X}\, \forall x.\ D(Ex) = x$$

Fact 26.6.1 Every type embeds into itself: $\forall X^{\mathbb{T}} : \mathcal{E}XX$.

Fact 26.6.2 $\forall XY^{\mathbb{T}} : \neg\mathcal{E}XY \to X \neq Y$.

Fact 26.6.3 $\mathbb{P}$ embeds into no proposition: $\forall P^{\mathbb{P}}.\ \neg\mathcal{E}\mathbb{P}P$.

Proof Follows with Coquand's theorem 26.4.1. ■

We now fix a computational universe U and work towards a proof of $\forall A^U.\ \neg\mathcal{E}UA$. We assume a type A^U and an embedding $\mathcal{E}UA$ with functions $E^{U\to A}$ and $D^{A\to U}$ satisfying $D(EX) = X$ for all types X^U. We will define a customized type $\mathcal{T} : U$ of Aczel trees for which we can define a subtree predicate and a universal tree. It then suffices to show irreflexivity of the subtree predicate to close the proof.

We define a type of customized Aczel trees:

$$\mathcal{T} : U \ ::= \ \mathsf{T}\,(a : A,\ Da \to \mathcal{T})$$

and a subtree predicate:

$$\begin{aligned} &\in\, :\ \mathcal{T} \to \mathcal{T} \to \mathbb{P} \\ &s \in \mathsf{T}\,af \ := \ \exists x.\ fx = s \end{aligned}$$

Fact 26.6.4 (Irreflexivity) $\forall s^{\mathcal{T}}.\, s \notin s$.

Proof Analogous to the proof of Fact 26.3.2. ■

Recall that we have to construct a contradiction. We embark on a little detour before we construct a universal tree. By Fact 26.6.1 and the assumption we have $\mathcal{E}\mathcal{T}(D(E\mathcal{T}))$. Thus there are functions $F^{\mathcal{T}\to D(E\mathcal{T})}$ and $G^{D(E\mathcal{T})\to\mathcal{T}}$ such that $\forall s^{\mathcal{T}}.\ G(Fs) = s$. We define

$$u := \ \mathsf{T}(E\mathcal{T})G$$

By Fact 26.6.1 it suffices to show $u \in u$. By definition of the membership predicate it suffices to show

$$\exists x.\ Gx = u$$

which holds with the witness $x := Fu$. We now have the hierarchy theorem for computational universes.

Theorem 26.6.5 (Hierachy) $\forall X^U.\ \neg\mathcal{E}UX$.

Exercise 26.6.6 Show $\forall X^U.\ X \neq U$ for all universes U.

Exercise 26.6.7 Let $i \neq j$. Show $\mathbb{T}_i \neq \mathbb{T}_j$.

Exercise 26.6.8 Assume the inductive type definition $A : \mathbb{T}_1 ::= C(\mathbb{T}_1)$ is admitted although it violates the dominance condition. Give a proof of falsity.

Acknowledgements

Thorsten Altenkirch suggested Aczel trees as a means for obtaining negative results in January 2016 at the POPL conference in St. Petersburg, Florida. Steven Schäfer came up with an elegant proof of Coquand's theorem using Aczel trees in June 2018 at the Types conference in Braga, Portugal.

27 Vectors

Vector types refine list types with an index recording the length of lists. Working with vector types is smooth in some cases and problematic in other cases. The problems stem from the fact that type checking relies on conversion rather than propositional equality.

27.1 Basic Definitions

The elements of a vector type $\mathcal{V}_n(X)$ may be though of as lists of length n over a base type X. The definition of the family of **vector types** $\mathcal{V}_n(X)$ accommodates X as a parameter and n as an index:

$$\begin{aligned}
&\mathcal{V}(X : \mathbb{T}) : \mathsf{N} \to \mathbb{T} \ ::= \\
&\mid \mathsf{Nil} : \ \mathcal{V}_0(X) \\
&\mid \mathsf{Cons} : \ \forall n.\ X \to \mathcal{V}_n(X) \to \mathcal{V}_{\mathsf{S}n}(X)
\end{aligned}$$

We write vector types $\mathcal{V}\,X\,n$ as $\mathcal{V}_n(X)$ to agree with the usual notation. The formal definition takes X before n since type constructors must take parameters before indices (a convention we adopt from Coq). For concrete vectors, we shall use the notation for lists. For instance, we write

$$[x, y, z] \quad \rightsquigarrow \quad \mathsf{Cons}\,X\,2\,x\,(\mathsf{Cons}\,X\,1\,y\,(\mathsf{Cons}\,X\,0\,z\,(\mathsf{Nil}\,X)))\ :\ \mathcal{V}_3(X)$$

We shall treat X as an implicit argument of Nil and Cons. Defining a **universal eliminator** for vectors

$$\begin{aligned}
&\forall X^{\mathbb{T}}\, \forall p^{\forall n.\ \mathcal{V}_n(X) \to \mathbb{T}}. \\
&p\,0\,\mathsf{Nil} \to \\
&(\forall nxv.\ pnv \to p(\mathsf{S}n)(\mathsf{Cons}\,xv)) \to \\
&\forall nv.\ pnv
\end{aligned}$$

is routine (recursion on the vector v). We will use the universal eliminator for inductive proofs.

We also define an **inversion operator** for vectors

$$\begin{aligned}
&\forall X\,\forall n \forall v^{\mathcal{V}_n(X)}.\\
&\textsc{match}\ n\ \textsc{return}\ \mathcal{V}_n(X) \to \mathbb{T}\\
&[\,0 \Rightarrow \lambda v.\, v = \mathsf{Nil}\\
&\mid \mathsf{S}n' \Rightarrow \lambda v.\Sigma x v'.\, v = \mathsf{Cons}\, n'\, x\, v'\\
&]\, v
\end{aligned}$$

by discrimination on v. We have explained the need for the reloading match before. The inversion operator will be essential for defining operations discriminating on nonempty vectors $\mathcal{V}_{\mathsf{S}n}(X)$.

27.2 Operations

We now define head and tail operation for vectors. In contrast to lists, the operations for vectors can exclude empty vectors through the index argument of their types. Moreover, head and tail can be obtained as instances of the inversion operator I:

$$\begin{aligned}
&\mathsf{hd} :\ \forall n.\ \mathcal{V}_{\mathsf{S}n}(X) \to X\\
&\mathsf{hd}\ n\, v\ := \ \pi_1(I(\mathsf{S}n)v)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{tl} :\ \forall n.\ \mathcal{V}_{\mathsf{S}n}(X) \to \mathcal{V}_n(X)\\
&\mathsf{tl}\ n\, v\ := \ \pi_1(\pi_2(I(\mathsf{S}n)v))
\end{aligned}$$

Note that hd and tl cannot be defined directly by discrimination on their vector argument since the index argument of the discriminating type is not an unconstrained variable. So to define hd and tl one must first come up with a generalized operation discriminating on an unconstrained vector type.

The problem reoccurs when we try to prove the η-law for vectors:

$$\forall n\, \forall v^{\mathcal{V}_{\mathsf{S}n}(X)}.\ v = \mathsf{Cons}\,(\mathsf{hd}\, v)(\mathsf{tl}\, v)$$

A direct discrimination of the vector argument is forbidden, and an application of the universal eliminator will not help. On the other hand, instantiating the inversion operator with v yields $\Sigma x v'.\ v = \mathsf{Cons}\, n'\, x\, v'$, which yields the claim by destructuring, rewriting, and conversion.

Now that we have hd and tl, showing injectivity of Cons

$$\mathsf{Cons}\ nxv = \mathsf{Cons}\ nx'v' \ \to\ x = x' \wedge v = v'$$

using feq is routine. With injectivity it is then routine to construct an equality decider for vectors:

$$\forall n\,\forall v_1 v_2^{\mathcal{V}_n(X)}.\ \mathcal{D}(v_1 = v_2)$$

The proof is as usual by recursion on v_1 (with v_2 quantified) followed by inversion of v_2 (using the inversion operator for vectors). Injectivity of Cons is needed for the negative cases obtained with the inductive hypothesis.

There is also an elegant definition of an operation that yields the last element of a vector:

$$\begin{aligned}
&\mathsf{last} : \forall n.\ \mathcal{V}_{\mathsf{S}n}(X) \to X \\
&\mathsf{last}\ 0 := \mathsf{hd}\,0 && : \mathcal{V}_{\mathsf{S}0}(X) \to X \\
&\mathsf{last}\ (\mathsf{S}n) := \lambda v.\,\mathsf{last}\,n\,(\mathsf{tl}\,(\mathsf{S}n)\,v) && : \mathcal{V}_{\mathsf{SS}n}(X) \to X
\end{aligned}$$

Note that last recurses on n rather than on the vector v, making a direct definition possible.

Another interesting operation on vectors is

$$\begin{aligned}
&\mathsf{sub} : \forall n.\ \mathcal{N}(n) \to \mathcal{V}_n(X) \to X \\
&\mathsf{sub}\,\_\,(\mathsf{Z}\,n) := \mathsf{hd}\,n && : \mathcal{V}_{\mathsf{S}n}(X) \to X \\
&\mathsf{sub}\,\_\,(\mathsf{U}\,n\,a) := \lambda v.\,\mathsf{sub}\,n\,a\,(\mathsf{tl}\,n\,a) && : \mathcal{V}_{\mathsf{S}n}(X) \to X
\end{aligned}$$

As with lists, sub nav yields the element the vector v carries at position a. If Z is interpreted as zero and U as successor operation, the positions are numbered from left to right starting with zero. The interesting fact here is that the numeral type $\mathcal{N}(n)$ contains exactly one numeral for every position of a vector $\mathcal{V}_n(X)$.

Exercise 27.2.1 Verify the above definitions and proofs using the proof assistant. In each case convince yourself that the inversion operator cannot be replaced by a direct discrimination.

Exercise 27.2.2 (Generalized head and tail) There is a direct realization of a generalized head operation incorporating ideas from the inversion operator:

$$\begin{aligned}
&\mathsf{hd} : \forall n\,\forall v^{\mathcal{V}_n(X)}.\ \text{MATCH}\ n\ [\,0 \Rightarrow \top \mid \mathsf{S}n' \Rightarrow X\,] \\
&\mathsf{hd}\,\_\,\mathsf{Nil} := \mathsf{I} && : \top \\
&\mathsf{hd}\,\_\,(\mathsf{Cons}\,n\,x\,v) := x && : X
\end{aligned}$$

Define a generalized tail operation using this idea.

Exercise 27.2.3 (Reversal) Define functions trunc, snoc, and rev as follows:

a) $\mathsf{trunc}\, nv$ yields the vector obtained by removing the last position of $v : \mathcal{V}_{\mathsf{S}n}(X)$.

b) $\mathsf{snoc}\, nvx$ yields the vector obtained by appending x at the end of $v : \mathcal{V}_n(X)$.

c) $\mathsf{rev}\, nv$ yields the vector obtained by reversing $v : \mathcal{V}_n(X)$.

Prove the following equations for your definitions (index arguments are omitted):

a) $\mathsf{last}\,(\mathsf{snoc}\, vx) = x$.

b) $v = \mathsf{snoc}\,(\mathsf{trunc}\, v)\,(\mathsf{last}\, v)$.

c) $\mathsf{rev}\,(\mathsf{snoc}\, vx) = \mathsf{Cons}\, x\,(\mathsf{rev}\, v)$.

d) $\mathsf{rev}\,(\mathsf{rev}\, v) = v$.

Hints: Equation (b) follows by induction on the index variable n and inversion of v, the others equations follow by induction on v.

Exercise 27.2.4 (Associativity) Define a concatenation operation

$$+\!\!+ : \forall Xmn.\ \mathcal{V}_m(X) \to \mathcal{V}_n(X) \to \mathcal{V}_{m+n}(X)$$

for vectors. Convince yourself that the statement for associativity of concatenation

$$(v_1 +\!\!+ v_2) +\!\!+ v_3 = v_1 +\!\!+ (v_2 +\!\!+ v_3)$$

does not type check (first add the implicit arguments and implicit types). Also check that the type checking problem would go away if the terms $(n_1 + n_2) + n_3$ and $n_1 + (n_2 + n_3)$ were convertible.

Exercise 27.2.5 Define a map function for vectors and prove its basic properties.

27.3 Converting between Vectors and Lists

We define a function that converts vectors into lists:

$$\begin{aligned} L :\ & \forall n.\ \mathcal{V}_n(X) \to \mathcal{L}(X) \\ L\,\_\,\mathsf{Nil} &:= [\,] \\ L\,\_\,(\mathsf{Cons}\, n\, x\, v) &:= x :: Lnv \end{aligned}$$

With a straightforward induction on vectors we show that L converts vectors of length n into lists of length n:

$$\forall nv.\ \mathsf{len}\,(Lnv) = n \tag{27.1}$$

We can also show that L is injective:

$$\forall n\, \forall v_1 v_2^{\mathcal{V}_n(X)}.\ Lnv_1 = Lnv_2 \ \to\ v_1 = v_2 \tag{27.2}$$

The proof is by induction on v_1 and inversion of v_2 and exploits the injectivity of the constructor cons for lists.

Next we show that L is surjective in the sense that every list of length n can be obtained from a vector of length n:

$$\forall A^{\mathcal{L}(X)}\, \Sigma v^{\mathcal{V}_{\mathsf{len}\,A}(X)}.\ L(\mathsf{len}\,A)v = A \tag{27.3}$$

We construct the function (27.3) by induction on A. Function (27.3) gives us a function

$$V:\ \mathcal{L}(X) \to \mathcal{V}_{\mathsf{len}\,A}(X)$$

such that

$$\forall A^{\mathcal{L}(X)}.\ L\,(\mathsf{len}\,A)\,(V(A)) = A$$

So far, so good. We now run into the problem that the statement

$$\forall n\, \forall v^{\mathcal{V}_n(X)}.\ V(Lnv) = v \tag{27.4}$$

does not type check since the types $\mathcal{V}_{\mathsf{len}(Lnv)}(X)$ and $\mathcal{V}_n(X)$ are not convertible. The problem may be resolved with a type cast transferring from $\mathcal{V}_{\mathsf{len}(Lnv)}(X)$ to $\mathcal{V}_n(X)$ based on the equation (27.1). The type checking problem goes away for concrete equations since conversion is strong enough if there are no variables. For instance, the concrete equation

$$L\,3\,(V[1,2,3]) = [1,2,3] \tag{27.5}$$

type checks (since $\mathsf{len}(L3(V[1,2,3])) \approx 3$) and holds by computational equality.

Exercise 27.3.1 Check the above claims with the proof assistant.

28 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.

28.1 Inverse Functions

We define predicates formulating basic properties of functions:

$$\begin{aligned} \mathsf{injective}\, f &:= \forall x x'.\, f x = f x' \to x = x' \\ \mathsf{surjective}\, f &:= \forall y \exists x.\, f x = y \\ \mathsf{bijective}\, f &:= \mathsf{injective}\, f \wedge \mathsf{surjective}\, f \\ \mathsf{inv}\, g f &:= \forall x.\, g(f x) = x \qquad\qquad g \text{ inverts } f \end{aligned}$$

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 28.1.1

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. $\mathsf{surjective}\, f \to \mathsf{inv}\, g f \to \mathsf{inv}\, g' f \to \forall y.\, g y = g' y$

Proof All claims follow by straightforward equational reasoning. Details are best understood with Coq. ∎

Note that Fact 28.1.1 (3) says that all inverse functions of a surjective function agree.

The following lemma facilitates the construction of inverse functions.

Lemma 28.1.2 $\forall f^{X\to Y}.\ (\forall y \Sigma x.\ fx = y) \to \Sigma g.\ \mathsf{inv}\, fg.$

Proof Let $G : \forall y \Sigma x.\ fx = y$ and define $gy := \pi_1(Gy)$. ■

Fact 28.1.3 (Transport) injective $f^{X\to Y} \to \mathcal{E}Y \to \mathcal{E}X.$

Proof Exercise. ■

Exercise 28.1.4 Give a function $\mathsf{N} \to \mathsf{N}$ that has disagreeing inverse functions.

28.2 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 28.2.1 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 28.2.2 Both functions of a bijection are bijective.

Proof Straightforward. ■

Theorem 28.2.3 (Pairing) $\mathcal{B}\,\mathsf{N}\,(\mathsf{N}\times\mathsf{N})$.

Proof See Chapter 7. ■

Exercise 28.2.4 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 $\mathcal{L}(\mathsf{N})$.

28.3 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 28.3.5). The second equation for injections for instance facilitates the construction of a least witness operator for data types (Fact 28.5.2).

Fact 28.3.1
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 28.3.2

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. $\mathcal{I}\perp X$
5. $\mathcal{I}X(\mathcal{O}(X))$

Proof Straightforward. Claim 5 follows with $fx := {}^\circ x$ and $ga := a$. ■

Fact 28.3.3 (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 28.3.4 (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:

$$qy := \text{MATCH } gy\ [\, {}^\circ x \Rightarrow px \mid \emptyset \Rightarrow \perp\,]$$

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 28.3.5 (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

$$\begin{aligned} &\forall x.\ g'(fx) = {}^\circ x \\ &\forall xy.\ g'y = {}^\circ x \to fx = y \end{aligned}$$

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 28.3.6 (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

$$\begin{aligned} &h : X \to \mathsf{B} \\ &hx := \text{MATCH } gx\ [\,{}^\circ\varphi \Rightarrow !\varphi x \mid \emptyset \to \mathbf{F}\,] \end{aligned}$$

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 28.3.7 Show $\mathcal{I}(X \to \mathsf{N})X \to \bot$.

Exercise 28.3.8 Show $\forall f^{X \to Y} \forall g.\ \mathsf{inv}\, g f \to \mathcal{E}Y \to \mathcal{I}XY$.

28.4 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 28.4.1

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: $\mathcal{I}XY \to \mathsf{dat}\,Y \to \mathsf{dat}\,X$.
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 28.2.3. ■

Fact 28.4.2 (Equality decider)
Data types have equality deciders.

Proof Follows with Fact 28.3.3. ■

Fact 28.4.3 (Existential witness operator)
Data types have existential witness operators.

Proof Follows with Fact 28.3.4. ■

Fact 28.4.4 (Inverse functions)
Bijective functions from data types to discrete types have inverse functions:
$\mathsf{bijective}\, f^{X \to Y} \to \mathsf{dat}\,X \to \mathcal{E}Y \to \Sigma g^{Y \to X}.\ \mathsf{inv}\,gf \wedge \mathsf{inv}\,fg$.

Proof By Fact 28.1.1 (2) it suffices to construct a function g such that $\mathsf{inv}\,fg$. By Lemma 28.1.2 it suffices to show $\forall y \Sigma x.\ fx = y$. Follows from the surjectivity of f with the existential witness operator for X (Fact 28.4.3) 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 28.4.5 (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 28.3.5 yields an injection $\mathcal{I}X\mathsf{N}$. ∎

Exercise 28.4.6 Show that $\mathsf{N}\to\mathsf{B}$ is not a data type.

Exercise 28.4.7 Show $\mathsf{dat}\,X \to \mathcal{W}(\mathcal{L}(X))$.

Exercise 28.4.8 Show that injections transport enumerators:
$\forall XY.\ \mathcal{I}XY \to \mathsf{enum}\,Y \to \mathsf{enum}\,X$.

28.5 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 28.5.1 (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 28.5.2 (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.

$$qn := \ \textsc{match}\ gn\ [\,{}^\circ x \Rightarrow px \mid \emptyset \Rightarrow \bot\,]$$

It is easy to see that q is decidable and Σ-satisfiable. Thus the least witness operator for numbers (Fact 15.2.2) 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 28.5.3 Define an existential least witness operator for data types: $\forall X^{\mathbb{T}}.\ \mathsf{dat}\,X \to (\forall x.\ \mathcal{D}(px)) \to (\exists x.\ px) \to (\Sigma x.\ px \wedge \forall y.\ py \to \#x \le \#y)$.

28.6 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 28.6.1 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 28.6.2 (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 17.3.6). 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 28.6.3 (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 28.4.2 and 28.4.3).

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 17.7.7). Since $\mathsf{len}\, A < \mathsf{len}\, (f@[0, \dots, n])$, the discrimination lemma 17.7.5 gives us an $x \notin A$. ■

Exercise 28.6.4 Show that B is not infinite.

Exercise 28.6.5 Show $\mathsf{dat}\, X \to \mathsf{infinite}\, X \to \Sigma x^X.\, \top$.

28.7 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.

$$g : \quad \emptyset, x_0, x_1, x_0, \emptyset, x_2, x_1, x_3, \dots$$

If g covers all members of X, we can compress g into a sequence $h : \mathsf{N} \to X$ without holes and repetitions:

$$h : \quad x_0, x_1, x_3, \dots$$

Seeing h as a function again, we have that h is a bijective function $\mathsf{N} \to X$.

Lemma 28.7.1 (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:

$$\begin{aligned} G_0 &:= [\,] \\ G_{\mathsf{S}n} &:= \text{MATCH } gn\ [\ ^\circ x \Rightarrow x :: G_n \mid \emptyset \Rightarrow Gn\ ] \end{aligned}$$

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 28.6.3 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 28.7.2 (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 $1 \to 2$. Compression lemma 28.7.1.
$2 \to 3$. Inverse function lemma 28.4.4.
$3 \to 4$. Fact 28.3.2 (3).
$4 \to 5$. Fact 28.3.1.
$5 \to 1$. Fact 28.6.3 (3). ■

29 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 and 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.

29.1 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:

$$\mathsf{listing}\ A^{\mathcal{L}(X)} \ := \ \mathsf{covering}\ A \land \mathsf{nrep}\ A$$

We need a couple of results for coverings and listings of discrete types.

Fact 29.1.1 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 17.7.4. ■

Fact 29.1.2 All listings of a discrete type have the same length.

Proof Follows with Corollary 17.7.6 (2). ■

Fact 29.1.3 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 17.7.6. ■

29.2 Finite Types

We define **finite types** as discrete types that come with a covering list:

$$\mathsf{fin}\ X^{\mathbb{T}} \ := \ \mathcal{E}(X) \times \Sigma A^{\mathcal{L}(X)}.\ \mathsf{covering}\, A$$

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**:

$$\mathsf{fin}_n\ X^{\mathbb{T}} \ := \ \mathcal{E}(X) \times \Sigma A^{\mathcal{L}(X)}.\ \mathsf{listing}\, A \wedge \mathsf{len}\, A = n$$

Fact 29.2.1 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 29.1.1 and 29.1.2. ■

Fact 29.2.2 If X and Y are finite types, then so are $X \times Y$, $X + Y$ and $\mathcal{O}(X)$.

Proof Discreteness follows with Facts 10.2.1 and 29.3.1. We leave the construction of the covering lists as an exercise. ■

Fact 29.2.3 Finite types are data types: $\mathsf{fin}\, X \to \mathcal{C}\, X$.

Proof By Fact 29.2.1 we assume a covering A for X. We use the upgrade lemma 28.3.5 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 § 17.9 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 17.9.1 ■

Fact 29.2.4 Finite types are not infinite: $\mathsf{fin}\, X \to \mathsf{infinite}\, X \to \bot$.

Proof Exercise. ■

Fact 29.2.5 $\mathsf{finite}\, X \ \to \ \mathcal{I}\,\mathsf{N}\, X \ \to \ \bot$.

Proof Follows with Facts 28.6.2, 28.6.1, and 29.2.4. ■

Fact 29.2.6 (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 29.1.3 (3).

Let f be injective. Then $f@A$ is nonrepeating by Exercise 17.7.7 (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 17.7.7 (b). ∎

Exercise 29.2.7 Prove $\mathsf{fin}_0\, \bot$, $\mathsf{fin}_1\, \top$, and $\mathsf{fin}_2\, \mathsf{B}$.

Exercise 29.2.8 Prove the following:

a) $\mathsf{dat}\, X \to (\exists A^{\mathcal{L}(X)} \forall x.\ x \in A) \to \mathsf{fin}\, X$.

b) $\mathsf{XM} \to \mathsf{dat}\, X \to \mathsf{infinte}\, X \lor (\exists A^{\mathcal{L}(X)} \forall x.\ x \in A)$.

Proving (b) with a sum type rather than a disjunction seems impossible.

Exercise 29.2.9 (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 29.2.10
Prove $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to m > 0 \to (\forall f^{X\to Y}.\ \mathsf{injective}\, f \longleftrightarrow \mathsf{surjective}\, f) \to m = n$.

29.3 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 29.3.1 (Discreteness) The finite ordinals are discrete: $\mathcal{E}(\mathsf{F}_n)$.

Proof Follows by induction on n since $\bot$ is discrete (Fact 10.2.1) and $\mathcal{O}$ preserves discreteness (Fact 10.3.2). ∎

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 29.3.2 L_n is a listing of F_n having length n.

Proof By induction on n. ∎

Fact 29.3.3 F_n is a finite type of size n: $\mathsf{fin}_n\, \mathsf{F}_n$.

Proof Facts 29.3.1 and 29.3.2. ∎

29.4 Bijections and Finite Types

Fact 29.4.1 (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 29.4.2 (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 $0, \dots, n-1$, and the elements of Y and the positions $0, \dots, n-1$. We realize the resulting bijection between X and Y using the list operations sub and pos with escape values (§ 17.9):

$$\begin{aligned} f x &:= \mathsf{sub}\, b\, B\ (\mathsf{pos}\, A\, x) \\ g y &:= \mathsf{sub}\, a\, A\ (\mathsf{pos}\, B\, y) \end{aligned}$$

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 29.4.3 $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to (\mathcal{B}XY \Leftrightarrow m = n)$.

Proof Direction $\Rightarrow$ follows with Facts 29.4.1 and 29.2.1 (2). Direction $\Leftarrow$ follows with Theorem 29.4.2. ∎

Exercise 29.4.4 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) $\mathcal{B}\, \mathsf{F}_n\, \mathsf{F}_{\mathsf{S}n} \to \bot$.

29.5 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 29.5.1 (Transport of covering lists)
$\mathcal{I} XY \to \mathsf{covering}_Y\, B \to \Sigma A.\; \mathsf{covering}_X\, A$.

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 $\mathcal{I} XY$. Let $A : \mathcal{L}(X)$ be the list obtained from $g@B$ by deleting the occurrences of $\emptyset$ and erasing the constructor $^\circ$ (Exercise 17.3.6). 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 29.5.2 (Transport of finiteness)
$\mathcal{I} XY \to \mathsf{fin}\, Y \to \mathsf{fin}\, X$.

Proof Fact 28.3.3 and Lemma 29.5.1. ∎

Fact 29.5.3 (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. $\Sigma Y.\, \mathcal{I} XY \;\times\; \mathsf{fin}\, Y$

Note that each of the types gives us a characterization of finite types.

Proof The equivalences follow with Facts 29.2.1 and 29.3.3, Theorem 29.4.2, and Facts 28.3.2 (3) and 29.5.2. ∎

Fact 29.5.4
$\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to m \le n \to \mathcal{I}\, X\, Y$.

Proof The proof is similar to the proof of Theorem 29.4.2. Again we use the upgrade lemma 28.3.5 so that only the first equation for $\mathcal{I} XY$ 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 $X \to \perp$ and the claim follows with computational elimination for $\perp$. Otherwise, we have an escape values $a \in A$ and $b \in B$. We realize the injection of X into Y as follows:

$$\begin{aligned} fx &:= \mathsf{sub}\, b\, B\, (\mathsf{pos}\, A\, x) \\ gy &:= {}^{\circ}\mathsf{sub}\, a\, A\, (\mathsf{pos}\, B\, y) \end{aligned}$$ ■

Fact 29.5.5
$\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to \mathcal{I}\, X\, Y \to m \leq n$.

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 17.7.6. ■

Theorem 29.5.6 (Finite injection) $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to (m \leq n \Leftrightarrow \mathcal{I}XY)$.

Proof Follows with Facts 29.5.4 and 29.5.5. ■

Corollary 29.5.7 $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to m > n \to \mathcal{I}XY \to \perp$.

Fact 29.5.8 (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 29.2.1, 29.5.2, and 29.5.5. Claim 2 follows with Fact 29.2.1, Claim 1, and Theorem 29.4.2. ■

Exercise 29.5.9 Prove the following:

a) $\mathcal{I}\, \mathsf{F}_m\, \mathsf{F}_n \;\Leftrightarrow\; m \leq n$.

b) $\mathcal{I}\, \mathsf{F}_{\mathsf{S}n}\, \mathsf{F}_n \;\to\; \perp$.

Exercise 29.5.10 Prove the following:
1. $\mathcal{I}\, \mathsf{N}\, \mathsf{F}_n \;\to\; \perp$.
2. $\mathsf{F}_n \neq \mathsf{N}$.

30 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 an introduction to propositional ND systems, and also as first example for the use of indexed inductive definitions.

30.1 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 30.1: 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 30.1.1 (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.

30.2 Intuitionistic ND System

The deduction rules of the intuitionistic ND system we will consider are given in Figure 30.1 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 30.1 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 30.1 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 30.1 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 30.1.

The rules in Figure 30.1 with a *logical constant* (i.e., $\bot$, $\to$, $\land$, $\lor$) 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 30.1 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 $\lor$. 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 $\land$, 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 30.2.1 A derivation for $s \vdash \neg\neg s$ may be depicted as a *derivation tree* as follows:

$$\cfrac{\cfrac{\cfrac{}{s, \neg s \vdash \neg s}\,\mathsf{A} \qquad \cfrac{}{s, \neg s \vdash s}\,\mathsf{A}}{s, \neg s \vdash \bot}\,\mathsf{E}_{\to}}{s \vdash \neg\neg s}\,\mathsf{I}_{\to}$$

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 $\neg\neg\bot \vdash \bot$.

30.3 Formalisation with Indexed Inductive Type Definition

It turns out that propositional deduction systems like the one in Figure 30.2 can be formalized elegantly and directly with inductive type definitions accommodating deduction rules as value constructors of derivation types $A \vdash s$.

$$\begin{array}{rl}
s \in A \;\to\; A \vdash s & \mathsf{A}_{\vdash} \\
A \vdash \bot \;\to\; A \vdash s & \mathsf{E}_{\bot} \\
A, s \vdash t \;\to\; A \vdash (s \to t) & \mathsf{I}_{\to} \\
A \vdash (s \to t) \;\to\; A \vdash s \;\to\; A \vdash t & \mathsf{E}_{\to} \\
A \vdash s \;\to\; A \vdash t \;\to\; A \vdash (s \wedge t) & \mathsf{I}_{\wedge} \\
A \vdash (s \wedge t) \;\to\; A, s, t \vdash u \;\to\; A \vdash u & \mathsf{E}_{\wedge} \\
A \vdash s \;\to\; A \vdash (s \vee t) & \mathsf{I}_{\vee}^{1} \\
A \vdash t \;\to\; A \vdash (s \vee t) & \mathsf{I}_{\vee}^{2} \\
A \vdash (s \vee t) \;\to\; A, s \vdash u \;\to\; A, t \vdash u \;\to\; A \vdash u & \mathsf{E}_{\vee}
\end{array}$$

Prefixes for A, s, t, u omitted, constructor names given at the right

Figure 30.2: Value constructors for derivation types $A \vdash s$

Let us explain this fundamental idea. We may see the deduction rules in Figure 30.1 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 30.2. Note that the types of the constructors follow exactly the patterns of the deduction rules in Figure 30.1.

When we look at the target types of the constructors in Figure 30.2, 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 ($\mathsf{I}_{\to}$, $\mathsf{I}_{\wedge}$, $\mathsf{I}_{\vee}^{1}$, $\mathsf{I}_{\vee}^{2}$). 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 30.2 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 30.3.1 (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 $\mathsf{E}_\rightarrow$ 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 $\mathsf{I}_\rightarrow$ and $\mathsf{A}_\vdash$.

(3) follows with $\mathsf{I}_\rightarrow$ from $s, \neg s \vdash \bot$, which follows with $\mathsf{E}_\rightarrow$ from $s, \neg s \vdash \neg s$ and $s, \neg s \vdash s$, which both follow with $\mathsf{A}_\vdash$. ■

Fact 30.3.2 (Cut) $A \vdash s \;\rightarrow\; A, s \vdash t \;\rightarrow\; A \vdash t$.

Proof We assume $A \vdash s$ and $A, s \vdash t$ and derive $A \vdash t$. By $\mathsf{I}_\rightarrow$ we have $A \vdash (s \rightarrow t)$. Thus $A \vdash t$ by $\mathsf{E}_\rightarrow$. ■

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 30.3.3 (Bottom) $A \vdash \neg\neg\bot \;\rightarrow\; A \vdash \bot$.

Proof Exercise. ■

30.4 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 30.3. 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.

$$
\begin{aligned}
\mathsf{E}_{\vdash} :\ & \forall p^{\mathcal{L}(\mathsf{For})\to\mathsf{For}\to\mathbb{T}}.\\
& (\forall As.\ s \in A \to pAs) \to\\
& (\forall As.\ pA\bot \to pAs) \to\\
& (\forall Ast.\ p(s :: A)t \to pA(s \to t)) \to\\
& (\forall Ast.\ pA(s \to t) \to pAs \to pAt) \to\\
& (\forall Ast.\ pAs \to pAt \to pA(s \wedge t)) \to\\
& (\forall Astu.\ pA(s \wedge t) \to p(s :: t :: A)u \to pAu) \to\\
& (\forall Ast.\ pAs \to pA(s \vee t)) \to\\
& (\forall Ast.\ pAt \to pA(s \vee t)) \to\\
& (\forall Astu.\ pA(s \vee t) \to p(s :: A)u \to p(t :: A)u \to pAu) \to\\
& \forall As.\ A \vdash s \to pAs
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{E}_{\vdash}\, pf_1 \dots f_9\, A\, \_\, (\mathsf{A}_{\vdash}\, sh) &:= f_1Ash\\
(\mathsf{E}_{\bot}\, sd) &:= f_2As(\mathsf{E}_{\vdash} \dots A\bot d)\\
(\mathsf{I}_{\to}\, std) &:= f_3Ast(\mathsf{E}_{\vdash} \dots (s :: A)td)\\
(\mathsf{E}_{\to}\, std_1d_2) &:= f_4Ast(\mathsf{E}_{\vdash} \dots A(s \to t)d_1)(\mathsf{E}_{\vdash} \dots Asd_2)\\
(\mathsf{I}_{\wedge}\, std_1d_2) &:= f_5Ast(\mathsf{E}_{\vdash} \dots Asd_1)(\mathsf{E}_{\vdash} \dots Atd_2)\\
(\mathsf{E}_{\wedge}\, stud_1d_2) &:= f_6Astu(\mathsf{E}_{\vdash} \dots A(s \wedge t)d_1)(\mathsf{E}_{\vdash} \dots (s :: t :: A)ud_2)\\
(\mathsf{I}^1_{\vee}\, std) &:= f_7Ast(\mathsf{E}_{\vdash} \dots Asd)\\
(\mathsf{I}^2_{\vee}\, std) &:= f_8Ast(\mathsf{E}_{\vdash} \dots Atd)\\
(\mathsf{E}_{\vee}\, stud_1d_2d_3) &:= f_9Astu(\mathsf{E}_{\vdash} \dots A(s \vee t)d_1)\\
&\qquad (\mathsf{E}_{\vdash} \dots (s :: A)ud_2)\\
&\qquad (\mathsf{E}_{\vdash} \dots (t :: A)ud_3)
\end{aligned}
$$

Figure 30.3: 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 30.3, 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 $\mathsf{E}_{\vdash}$.

30.5 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 30.5.1 (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 30.5.2 (Implication) $A \vdash (s \to t) \iff A, s \vdash t$.

Proof Direction $\Leftarrow$ holds by $\mathsf{I}_\rightarrow$. For direction $\Rightarrow$ we assume $A \vdash (s \rightarrow t)$ and obtain $A, s \vdash (s \rightarrow t)$ with weakening. Now $\mathsf{A}_\vdash$ and $\mathsf{E}_\rightarrow$ 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 $(s :: A) \cdot t := A \cdot (s \rightarrow t)$. For instance, we have $[s_1, s_2, s_3] \cdot t = s_3 \rightarrow s_2 \rightarrow s_1 \rightarrow t$.

Fact 30.5.3 (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 30.5.4 (Ground Prover) $\forall s.\ \mathsf{ground}\, s \rightarrow ([] \vdash s) + ([] \vdash \neg s)$.

Proof By induction on s using weakening. ∎

Exercise 30.5.5 Establish the following functions:

a) $A \vdash (s_1 \rightarrow s_2 \rightarrow t) \ \rightarrow\ A \vdash s_1 \ \rightarrow\ A \vdash s_2 \ \rightarrow\ A \vdash t$.

b) $\neg\neg s \in A \ \rightarrow\ A, s \vdash \bot \ \rightarrow\ A \vdash \bot$.

c) $A, s, \neg t \vdash \bot \ \rightarrow\ A \vdash \neg\neg(s \rightarrow t)$.

Exercise 30.5.6 Prove the following types.

a) $A \vdash ((\neg s \rightarrow \neg\neg\bot) \rightarrow \neg\neg s)$.

b) $A \vdash ((s \rightarrow \neg\neg t) \rightarrow \neg\neg(s \rightarrow t))$.

c) $A \vdash (\neg\neg(s \rightarrow t) \rightarrow \neg\neg s \rightarrow \neg\neg t)$.

d) $A \vdash (\neg\neg\neg s \rightarrow \neg s)$.

e) $A \vdash (\neg s \rightarrow \neg\neg\neg s)$.

Exercise 30.5.7 Prove $\forall s.\ \mathsf{ground}\, s \rightarrow\ \vdash (s \vee \neg s)$.

Exercise 30.5.8 Prove $\forall A s.\ \mathsf{ground}\, s \rightarrow A, s \vdash t \ \rightarrow\ A, \neg s \vdash t \ \rightarrow\ A \vdash t$.

Exercise 30.5.9 Prove the deduction laws for conjunctions and disjunctions:

a) $A \vdash (s \wedge t) \ \Leftrightarrow\ A \vdash s \ \times\ A \vdash t$

b) $A \vdash (s \vee t) \ \Leftrightarrow\ \forall u.\ \ A, s \vdash u \ \rightarrow\ A, t \vdash u \ \rightarrow\ A \vdash u$

30.6 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,

$$\nvdash \bot \qquad \nvdash x \qquad \nvdash (\neg\neg x \to x)$$

(we write $\nvdash s$ 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:

$$\mathsf{imp}\, ab \;:=\; \text{IF } a \le b \text{ THEN } 2 \text{ ELSE } b$$

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 $A \vDash s$ called **Heyting entailment** for which we can show the implication

$$A \vdash s \to A \vDash s$$

known as **soundness**. Using soundness, we can disprove $\vdash (\neg\neg x \to x)$ by disproving $\vDash (\neg\neg x \to x)$. It is common to refer to $A \vdash s$ as a **syntactic entailment relation** and to $A \vDash s$ 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 30.6.1 $s \in A \to A \vDash s$.

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 $(\mathcal{E}\alpha(u :: A) \leq \mathcal{E}\alpha s) = \mathbf{T}$ where $s \in A$. The inductive hypothesis gives us $(\mathcal{E}\alpha A \leq \mathcal{E}\alpha s) = \mathbf{T}$. 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 30.6.2 (Soundness) $A \vdash s \;\rightarrow\; A \vDash s$.

Proof By induction on $A \vdash s$. The case for the assumption rule follows with Lemma 30.6.1. The case for the explosion rule fixes an assignment α, obtains $(\mathcal{E}\alpha A \leq 0) = \mathbf{T}$ by the inductive hypothesis, and shows $(\mathcal{E}\alpha A \leq \mathcal{E}\alpha s) = \mathbf{T}$ 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 30.6.3 $\vdash s \rightarrow \mathcal{E}\alpha s = 2$.

A formula s is **independent in** $\vdash$ if one can prove both $(\vdash s) \rightarrow \bot$ and $(\vdash \neg s) \rightarrow \bot$.

Corollary 30.6.4 (Independence) $\neg\neg x \rightarrow x$ and $x \vee \neg x$ are independent in $\vdash$.

Proof Follows with Corollary 30.6.3 and $\alpha n := 1$. ■

Corollary 30.6.5 (Consistency) $\nvdash \bot$.

Exercise 30.6.6 Show that x, $\neg x$, and $(x \rightarrow y) \rightarrow x) \rightarrow x$ are independent in $\vdash$.

Exercise 30.6.7 Show $\neg\forall s.\ ((\vdash (\neg\neg s \rightarrow s)) \rightarrow \bot)$.

Exercise 30.6.8 Show $A \vDash \bot \longleftrightarrow \forall \alpha.\ \mathcal{E}\alpha A = 0$ and $\vDash s \longleftrightarrow \forall \alpha.\ \mathcal{E}\alpha s = 2$.

30.7 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 30.7.1 (Double Negation) $A \dot\vdash (\neg\neg s \to s)$.

Proof Straightforward using the contradiction rule. ■

Fact 30.7.2 (Cut) $A \dot\vdash s \;\to\; A, s \dot\vdash t \;\to\; A \dot\vdash t$.

Proof Same as for the intuitionistic system. ■

Fact 30.7.3 (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 30.7.4 (Explosion) $A \dot\vdash \bot \;\to\; A \dot\vdash s$.

Proof By contradiction and weakening. ■

Fact 30.7.5 (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 30.7.6 (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 30.7.7 (Refutation) $A \dot\vdash s \;\Leftrightarrow\; A, \neg s \dot\vdash \bot$.

Proof Direction $\Rightarrow$ follows with weakening. Direction $\Leftarrow$ 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 30.7.8 Show $\dot\vdash s \vee \neg s$ and $\dot\vdash ((s \to t) \to s) \to s$.

Exercise 30.7.9 Show that classical ND is not sound for Heyting entailment: $\neg(\forall As.\ A \dot\vdash s \ \to\ A \vDash s)$.

Exercise 30.7.10 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 30.7.11 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.

30.8 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 30.8.1 $A \dot\vdash s \ \to\ A \vdash \neg\neg s$.

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 30.5.6 and the implication lemma 30.5.2. ■

Theorem 30.8.2 (Glivenko) $A \dot\vdash s \ \Leftrightarrow\ A \vdash \neg\neg s$.

Proof Direction $\Rightarrow$ follows with Lemma 30.8.1. Direction $\Leftarrow$ follows with translation (30.7.5) and double negation (30.7.1). ■

Corollary 30.8.3
Classical ND reduces to intuitionistic ND refutation: $A \dot\vdash s \Leftrightarrow A, \neg s \vdash \bot$.

Corollary 30.8.4 Intuitionistic and classical refutation agree: $A \vdash \bot \Leftrightarrow A \dot\vdash \bot$.

Proof Direction $\Rightarrow$ follows with translation 30.7.5. Direction $\leftarrow$ follows with Glivenko's theorem and the bottom law 30.3.3. ■

Corollary 30.8.5 Classical ND is consistent: $(\dot\vdash \bot) \to \bot$.

Proof Let $\dot\vdash \bot$. By consistency of intuitionistic ND (30.6.5) it suffices to prove $\vdash \bot$. Follows by Glivenko and the bottom law 30.3.3. ■

Exercise 30.8.6 Disprove $\dot\vdash x$ and $\dot\vdash \neg x$.

Exercise 30.8.7 Disprove $A \dot\vdash (s \vee t) \Leftrightarrow A \dot\vdash s \vee A \dot\vdash t$.

30.9 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 30.6, we define **boolean evaluation** and **boolean entailment** as shown in Figure 30.4. 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: $A \dot\vdash s \Leftrightarrow A \dot\vDash s$. 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 30.7.9).

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 30.9.1 $s \in A \to A \dot\vDash s$.

Proof By induction on A. ■

Fact 30.9.2 (Soundness) $A \dot\vdash s \to A \dot\vDash s$.

Proof By induction on $A \dot\vdash s$ using Lemma 30.9.1 for the assumption rule. ■

We can now give a consistency proof for classical ND that does not make use of intuitionistic ND.

Corollary 30.9.3 Classical ND is consistent: $(\dot\vdash \bot) \to \bot$.

Proof Follows with soundness and $\mathcal{E}\alpha\bot = 0$. ■

$$\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathsf{B}$$

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= \mathbf{F} \\
\mathcal{E}\alpha(s \to t) &:= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathbf{T} \\
\mathcal{E}\alpha(s \wedge t) &:= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathbf{F} \\
\mathcal{E}\alpha(s \vee t) &:= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathbf{T} \text{ ELSE } \mathcal{E}\alpha t
\end{aligned}$$

$$\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathcal{L}(\mathsf{For}) \to \mathsf{B}$$

$$\begin{aligned}
\mathcal{E}\alpha([]) &:= \mathbf{T} \\
\mathcal{E}\alpha(s :: A) &:= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha A \text{ ELSE } \mathbf{F}
\end{aligned}$$

$$A \mathrel{\dot{\vDash}} s \;:=\; \forall \alpha.\ \text{IF } \mathcal{E}\alpha A \text{ THEN } \mathcal{E}\alpha s = \mathbf{T} \text{ ELSE } \top$$

Figure 30.4: Boolean evaluation and entailment

Fact 30.9.4

1. $A \mathrel{\dot{\vDash}} s \;\longleftrightarrow\; \forall \alpha.\ \mathcal{E}\alpha A = \mathbf{T} \to \mathcal{E}\alpha s = \mathbf{T}$
2. $A \mathrel{\dot{\vDash}} \bot \;\longleftrightarrow\; \forall \alpha.\ \mathcal{E}\alpha A = \mathbf{F}$
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 30.9.5 Show that x and $\neg x$ are independent in $\dot{\vdash}$.

Exercise 30.9.6 Give boolean assignments showing that $\neg\neg\neg x$ is independent in $\dot{\vdash}$.

Exercise 30.9.7 Show the following properties of boolean entailment:

a) $A \mathrel{\dot{\vDash}} s \;\longleftrightarrow\; A, \neg s \mathrel{\dot{\vDash}} \bot$

b) $A \mathrel{\dot{\vDash}} s \;\longleftrightarrow\; \mathrel{\dot{\vDash}} A \cdot s$

c) $A \mathrel{\dot{\vDash}} (s \to t) \;\longleftrightarrow\; A, s \mathrel{\dot{\vDash}} t$

Exercise 30.9.8 Show $(\forall st.\ \dot{\vdash}(s \vee t) \to (\dot{\vdash} s) + (\dot{\vdash} t)) \to \bot$.

Exercise 30.9.9 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 $\dot{\vDash}(s \wedge t \to fst)$ and $\dot{\vDash}(fst \to s \wedge t)$. Do the same for disjunction.

30.10 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 30.9.4 (2)).

Fact 30.10.1
Given a certifying boolean solver, classical ND is complete and decidable:

1. $A \dot{\vDash} s \ \rightarrow\ A \dot{\vdash} s$ (completeness)
2. $\mathcal{D}(A \dot{\vdash} s)$ (decidability)

Proof By the refutation laws (Fact 30.7.7 and Exercise 30.9.7) it suffices to show the claims for $s = \bot$.

For (1) we assume $A \dot{\vDash} \bot$ 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 30.9.4 (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 30.9.4 (2). ■

In Chapter 31, we will construct a certifying boolean solver using the tableau method.

Exercise 30.10.2 (Refutation predicates)
A refutation predicate is a predicate $\rho : \mathcal{L}(\mathsf{For}) \rightarrow \mathbb{P}$ satisfying the following rules:

$\rho(s :: A +\!\!\!+ B) \rightarrow \rho(A +\!\!\!+ s :: B)$	rotation
$\rho(x :: \neg x :: A)$	clash
$\rho(\bot :: A)$	falsity
$\rho(\neg s :: A) \rightarrow \rho(t :: A) \rightarrow \rho((s \rightarrow t) :: A)$	implication
$\rho(s :: \neg t :: A) \rightarrow \rho(\neg(s \rightarrow t) :: A))$	implication$^-$
$\rho(s :: t :: A) \rightarrow \rho((s \wedge t) :: A)$	conjunction
$\rho(\neg s :: A) \rightarrow \rho(\neg t :: A) \rightarrow \rho(\neg(s \wedge t) :: A)$	conjunction$^-$
$\rho(s :: A) \rightarrow \rho(t :: A) \rightarrow \rho((s \vee t) :: A)$	disjunction
$\rho(\neg s :: \neg t :: A) \rightarrow \rho(\neg(s \vee t) :: A)$	disjunction$^-$

We will show in Chapter 31 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 $\lambda A.\ A \dot{\vdash} \bot$ is a refutation predicate.

b) Show that $\lambda A. \forall\alpha.\ \mathcal{E}\alpha A = \mathbf{F}$ is a refutation predicate.

Exercise 30.10.3 Assume a certifying boolean solver and show $\mathcal{D}(A \dot{\vDash} s)$.

30.11 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:

$$\begin{aligned}
\theta\cdot x &:= \theta x \\
\theta\cdot\bot &:= \bot \\
\theta\cdot(s \to t) &:= \theta\cdot s \to \theta\cdot t \\
\theta\cdot(s \wedge t) &:= \theta\cdot s \wedge \theta\cdot t \\
\theta\cdot(s \vee t) &:= \theta\cdot s \vee \theta\cdot t
\end{aligned}$$

$$\begin{aligned}
\theta\cdot[] &:= [] \\
\theta\cdot(s :: A) &:= \theta\cdot s :: \theta\cdot A
\end{aligned}$$

We will write θs and θA for $\theta\cdot s$ and $\theta\cdot A$.

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 30.11.1 $s \in A \to \theta s \in \theta A$.

Proof By induction on A. ■

Fact 30.11.2 $A \vdash s \ \to\ \theta A \vdash \theta s$ and $A \dot{\vdash} s \ \to\ \theta A \dot{\vdash} \theta s$.

Proof By induction on $A \vdash s$ and $A \dot{\vdash} s$ using Fact 30.11.1. ■

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:* $\exists s.\ \nVdash s$.
5. *Substitutivity:* $A \Vdash s \;\to\; \theta A \Vdash \theta s$.
6. *Explosion:* $A \Vdash \bot \;\to\; A \Vdash s$.
7. *Implication:* $A \Vdash (s \to t) \;\longleftrightarrow\; A, s \Vdash t$.
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 30.5: Requirements for entailment predicates

Lemma 30.11.3
$\mathcal{E}\alpha(\theta s) = \mathcal{E}(\lambda n.\mathcal{E}\alpha(\theta n))s$ holds both for Heyting and boolean evaluation.

Fact 30.11.4 $A \vDash s \;\to\; \theta A \vDash \theta s$ and $A \dot{\vDash} s \;\to\; \theta A \dot{\vDash} \theta s$.

Proof By induction on A using Lemma 30.11.3. ■

30.12 Entailment Predicates

An **entailment predicate** is a predicate

$$\Vdash : \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{P}$$

satisfying the properties listed in Figure 30.5. 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 30.12.1 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 30.12.2 (Modus Ponens) $A \Vdash (s \to t) \;\to\; A \Vdash s \;\to\; A \Vdash t$.

Proof By implication and cut. ■

Fact 30.12.3 $A \vdash s \to A \Vdash s$. That is, intuitionistic ND is a least entailment predicate.

Proof By induction on $A \vdash s$ using modus ponens. ■

Fact 30.12.4 $\Vdash s \to \Vdash \neg s \to \bot$.

Proof Let $\Vdash s$ and $\Vdash \neg s$. By Fact 30.12.2 we have $\Vdash \bot$. By consistency and explosion we obtain a contradiction. ■

Fact 30.12.5 (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 30.12.6 (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 30.12.2 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 30.12.7 $\Vdash s \to \dot{\vDash} s$.

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 30.12.4. ■

Theorem 30.12.8 (Sandwich)
Let $\Vdash$ be an entailment predicate. Then $A \vdash s \to A \Vdash s$ and $A \Vdash s \to A \dot{\vDash} s$.

Proof Claim 1 is Fact 30.12.3. Claim 2 follows with Lemma 30.12.7 and Facts 30.12.5 and 30.12.1. ■

Exercise 30.12.9 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 30.12.10 Tebbi's lemma provides for a particularly elegant proof of Lemma 30.12.7. Verify that Lemma 30.12.7 can also be obtained from the facts (1) $\vdash \hat{\alpha}s \vee \vdash \neg\hat{\alpha}s$ and (2) $\dot{\vDash} \hat{\alpha}s \to \mathcal{E}\alpha s = \mathbf{T}$ using Facts 30.12.3 and 30.12.4.

30.13 Notes

The study of natural deduction originated in the 1930’s with the work of Gerhard Gentzen [9, 10] and Stanisław Jaśkowski [13]. The standard text on natural deduction and proof theory is Troelstra and Schwichtenberg [19].

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.

31 Boolean Satisfiability

We study satisfiability of boolean formulas by constructing and verifying a DNF 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. 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 boolean expressions, nontrivial functions, and indexed inductive type families (the tableau system).

31.1 Boolean Operations

We will work with the boolean operations *conjunction*, *disjunction*, and *negation*, which we obtain as inductive functions $\mathsf{B} \to \mathsf{B} \to \mathsf{B}$ and $\mathsf{B} \to \mathsf{B}$:

$$\mathbf{T} \mathbin{\&} b := b \qquad \mathbf{T} \mid b := \mathbf{T} \qquad !\mathbf{T} := \mathbf{F}$$
$$\mathbf{F} \mathbin{\&} b := \mathbf{F} \qquad \mathbf{F} \mid b := b \qquad !\mathbf{F} := \mathbf{T}$$

With these definitions, boolean identities like

$$a \mathbin{\&} b = b \mathbin{\&} a \qquad a \mid b = b \mid a \qquad !!b = b$$

have straightforward proofs by boolean case analysis and computational equality. Recall that boolean conjunction and disjunction are commutative and associative.

An important notion for our development is disjunctive normal form (DNF). The idea behind DNF is that conjunctions are below disjunctions, and that negations are below conjunctions. Negations can be pushed downwards with the *negation laws*

$$!(a \mathbin{\&} b) = !a \mid !b \qquad !(a \mid b) = !a \mathbin{\&} !b \qquad !!a = a$$

and conjunctions can be pushed below disjunctions with the *distribution law*

$$a \mathbin{\&} (b \mid c) = (a \mathbin{\&} b) \mid (a \mathbin{\&} b)$$

Besides the defining equations, we will also make use of the *negation law*

$$b \wedge !b = \mathbf{F}$$

to eliminate conjunctions.

There are 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 not a good idea since the computation rules coming with the boolean operations are lost. The exception is the reflection rule for conjunctions, which offers the possibility to replace the argument terms of a conjunction with $\mathbf{T}$.

31.2 Boolean Formulas

Our main interest will be in boolean formulas, which are syntactic representations of boolean terms. We will consider the boolean **formulas**

$$s,t,u : \mathsf{For} \ ::= \ x \mid \bot \mid s \rightarrow t \mid s \wedge t \mid s \vee t \qquad (x : \mathsf{N})$$

realized with an inductive data type For representing each syntactic form with a value constructor. **Variables** x are represented as numbers. We will refer to formulas also as **boolean expressions**.

Our development would 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 $\neg$, which we accommodate at the meta level with the notations

$$\top := \bot \rightarrow \bot \qquad\qquad \neg s := s \rightarrow \bot$$

Given an **assignment** $\alpha : \mathsf{N} \rightarrow \mathsf{B}$, we can evaluate every formula to a boolean value. We formalize evaluation of formulas with the **evaluation function** shown in Figure 31.1. 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

$$\alpha \vDash s \ := \ \mathcal{E}\alpha s = \mathbf{T}$$

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= \mathbf{F} \\
\mathcal{E}\alpha(s \to t) &:= \,!\mathcal{E}\alpha s \mid \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \wedge t) &:= \mathcal{E}\alpha s \,\&\, \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \vee t) &:= \mathcal{E}\alpha s \mid \mathcal{E}\alpha t
\end{aligned}$$

Figure 31.1: Definition of the evaluation function $\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathsf{B}$

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 $\mathsf{sat}\ s$ if s has a solution. Finally, we say that two formulas are **equivalent** if they have the same solutions.

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 31.2.1 Prove that $s \to t$ and $\neg s \vee t$ are equivalent.

Exercise 31.2.2 Convince yourself that the predicate $\alpha \vDash s$ is decidable.

Exercise 31.2.3 Verify the following reflection laws for formulas:

$$\begin{aligned}
\alpha \vDash (s \wedge t) &\longleftrightarrow \alpha \vDash s \wedge \alpha \vDash t \\
\alpha \vDash (s \vee t) &\longleftrightarrow \alpha \vDash s \vee \alpha \vDash t \\
\alpha \vDash \neg s &\longleftrightarrow \neg(\alpha \vDash s)
\end{aligned}$$

Exercise 31.2.4 (Compiler to implicative fragment) Write and verify a compiler $\mathsf{For} \to \mathsf{For}$ translating formulas into equivalent formulas not containing conjunctions and disjunctions.

Exercise 31.2.5 (Equation compiler) Write and verify a compiler

$$\gamma : \mathcal{L}(\mathsf{For} \times \mathsf{For}) \to \mathsf{For}$$

translating lists of equations into equivalent formulas:

$$\forall \alpha.\ \ \alpha \vDash \gamma A \ \longleftrightarrow\ \forall (s,t) \in A.\ \ \mathcal{E}\alpha s = \mathcal{E}\alpha t$$

Exercise 31.2.6 (Valid formulas) We say that a formula is **valid** if it is satisfied by all assignments: $\mathsf{val}\, s := \forall \alpha.\ \alpha \vDash s$. Verify the following reductions.

a) s is valid iff $\neg s$ is unsatisfiable: $\forall s.\ \mathsf{val}\, s \longleftrightarrow \neg\mathsf{sat}(\neg s)$.

b) $\forall s.\ \mathsf{stable}(\mathsf{sat}\, s) \to (\mathsf{sat}\, s \longleftrightarrow \neg\mathsf{val}(\neg s))$.

Exercise 31.2.7 Write an evaluator $f : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathbb{P}$ such that $f\alpha s \longleftrightarrow \alpha \vDash s$ and $f\alpha(s \lor t) \approx f\alpha s \lor f\alpha t$ for all formulas s, t.
Hint: Recall the reflection laws from § 31.1.

31.3 Clausal DNFs

We are working towards a decider for satisfiability of boolean formulas. The decider will compute a *DNF* (disjunctive normal form) for the given formula and exploit that from the DNF it is clear whether the formula is decidable. Informally, a DNF is either the formula $\bot$ or a disjunction $s_1 \lor \cdots \lor s_n$ of *solved formulas* s_i, where a solved formula is a conjunction of variables and negated variables such that no variable appears both negated and unnegated. One can show that every formula is equivalent to a DNF. Since every solved formula is satisfiable, a DNF is satisfiable if and only if it is different from $\bot$.

There may be many different DNFs for satisfiable formulas. For instance, the DNFs $x \lor \neg x$ and $y \lor \neg y$ are equivalent since they are satisfied by every assignment.

Formulas by themselves are not a good data structure for computing DNFs of formulas. We will work with lists of signed formulas we call clauses:

$$S, T \ :\ \mathsf{SFor} ::= s^+ \mid s^- \qquad\qquad \textbf{signed formula}$$
$$C, D \ :\ \mathsf{Cla} := \mathcal{L}(\mathsf{SFor}) \qquad\qquad \textbf{clause}$$

Clauses represent conjunctions. We define evaluation of signed formulas and clauses as follows:

$$\mathcal{E}\alpha(s^+) := \mathcal{E}\alpha s \qquad\qquad \mathcal{E}\alpha\,[\,] := \mathbf{T}$$
$$\mathcal{E}\alpha(s^-) := \,!\mathcal{E}\alpha s \qquad\qquad \mathcal{E}\alpha(S :: C) := \mathcal{E}\alpha S \,\&\, \mathcal{E}\alpha C$$

Note that the empty clause represents the boolean $\mathbf{T}$. We also consider lists of clauses

$$\Delta \ :\ \mathcal{L}(\mathsf{Cla})$$

and interpret them disjunctively:

$$\mathcal{E}\alpha\,[\,] := \mathbf{F}$$
$$\mathcal{E}\alpha\,(C :: \Delta) := \mathcal{E}\alpha C \mid \mathcal{E}\alpha\Delta$$

Satisfaction of signed formulas, clauses, and lists of clauses is defined analogously to formulas, and so are the notations $\alpha \vDash S$, $\alpha \vDash C$, $\alpha \vDash \Delta$, and $\mathsf{sat}\ C$. 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 31.3.1 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 31.3.2 A clausal DNF is satisfiable if and only if it is nonempty.

Exercise 31.3.3 Prove $\mathcal{E}\alpha(C + D) = \mathcal{E}\alpha C \mathbin{\&} \mathcal{E}\alpha D$ and $\mathcal{E}\alpha(\Delta + \Delta') = \mathcal{E}\alpha\Delta \mid \mathcal{E}\alpha\Delta'$.

Exercise 31.3.4 Write a function that maps lists of clauses to equivalent formulas.

Exercise 31.3.5 Our formal proof of Fact 31.3.1 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 31.3.1. Can you do better?

31.4 DNF Solver

We would like to construct a function computing clausal DNFs for formulas. Formally, we specify the function with the informative type

$$\forall s\, \Sigma\Delta.\ \ \mathsf{DNF}\,\Delta \wedge s \equiv \Delta$$

where

$$\begin{aligned} s \equiv \Delta \;&:=\; \forall \alpha.\ \alpha \vDash s \longleftrightarrow \alpha \vDash \Delta \\ \mathsf{DNF}\,\Delta \;&:=\; \forall C \in \Delta.\ \mathsf{solved}\,C \end{aligned}$$

To define the function, we will generalize the type to

$$\forall CD.\ \mathsf{solved}\,C \to \Sigma\Delta.\ \ \mathsf{DNF}\,\Delta \wedge C + D \equiv \Delta$$

$$\begin{array}{rcl}
\mathsf{dnf}\ C\ [] & = & [C] \\
\mathsf{dnf}\ C\ (x^+ :: D) & = & \text{IF } \ulcorner x^- \in C \urcorner \text{ THEN } [] \text{ ELSE } \mathsf{dnf}\ (x^+ :: C)\ D \\
\mathsf{dnf}\ C\ (x^- :: D) & = & \text{IF } \ulcorner x^+ \in C \urcorner \text{ THEN } [] \text{ ELSE } \mathsf{dnf}\ (x^- :: C)\ D \\
\mathsf{dnf}\ C\ (\bot^+ :: D) & = & [] \\
\mathsf{dnf}\ C\ (\bot^- :: D) & = & \mathsf{dnf}\ C\ D \\
\mathsf{dnf}\ C\ ((s \to t)^+ :: D) & = & \mathsf{dnf}\ C\ (s^- :: D) +\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \to t)^- :: D) & = & \mathsf{dnf}\ C\ (s^+ :: t^- :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^+ :: D) & = & \mathsf{dnf}\ C\ (s^+ :: t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^- :: D) & = & \mathsf{dnf}\ C\ (s^- :: D) +\!\!+ \mathsf{dnf}\ C\ (t^- :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^+ :: D) & = & \mathsf{dnf}\ C\ (s^+ :: D) +\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^- :: D) & = & \mathsf{dnf}\ C\ (s^- :: t^- :: D)
\end{array}$$

Figure 31.2: Specification of a procedure $\mathsf{dnf} : \mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$

where $C \equiv \Delta := \forall \alpha.\ \alpha \vDash C \longleftrightarrow \alpha \vDash \Delta$. To compute a clausal DNF of a formula s, we will apply the function with $C = []$ and $D = [s^+]$.

We base the definition of the function on a purely computational procedure

$$\mathsf{dnf} :\ \mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$$

specified with equations in Figure 31.2. We refer to the first argument C of the procedure as **accumulator**, and to the second argument as **agenda**. The agenda holds the signed formulas still to be processed, and the accumulator collects signed variables taken from the agenda. The procedure processes the formulas on the agenda one by one decreasing the size of the agenda with every recursion step. We define the **size** of clauses and formulas as follows:

$$\begin{array}{rclcrcl}
\sigma[] & := & 0 & \qquad & \sigma x & := & 1 \\
\sigma(s^+ :: C) & := & \sigma s + \sigma C & & \sigma\bot & := & 1 \\
\sigma(s^- :: C) & := & \sigma s + \sigma C & & \sigma(s \circ t) & := & 1 + \sigma s + \sigma t
\end{array}$$

Note that the equations specifying the procedure in Figure 31.2 are clear from the correctness properties stated for the procedure, the design that the first formula on the agenda controls the recursion, and the boolean identities given in Section 31.1.

Lemma 31.4.1 $\forall C D.\ \mathsf{solved}\, C \to \Sigma \Delta.\ \ \mathsf{DNF}\, \Delta \wedge C +\!\!+ D \equiv \Delta.$

Proof By size induction on σD with C quantified in the inductive hypothesis augmenting the design of the procedure dnf with the necessary proofs. Each of the 13 cases is straightforward. ∎

Theorem 31.4.2 (DNF solver) $\forall C \Sigma \Delta.\ \mathsf{DNF}\,\Delta \land C \equiv \Delta$.

Proof Immediate from Lemma 31.4.1. ∎

Corollary 31.4.3 $\forall s \Sigma \Delta.\ \mathsf{DNF}\,\Delta \land s \equiv \Delta$.

Corollary 31.4.4 There is a solver $\forall C.\ (\Sigma\alpha.\ \alpha \vDash C) + \neg\mathsf{sat}\ C$.

Corollary 31.4.5 There is a solver $\forall s.\ (\Sigma\alpha.\ \alpha \vDash s) + \neg\mathsf{sat}\ s$.

Corollary 31.4.6 Satisfiability of clauses and formulas is decidable.

Exercise 31.4.7 Convince yourself that the predicate $S \in C$ is decidable.

Exercise 31.4.8 Rewrite the equations specifying the DNF procedure so that you obtain a boolean decider $\mathcal{D} : \mathsf{Cla} \to \mathsf{Cla} \to \mathsf{B}$ for satisfiability of clauses. Give an informative type subsuming the procedure and specifying the correctness properties for a boolean decider for satisfiability of clauses.

Exercise 31.4.9 Recall the definition of valid formulas from Exercise 31.2.6. Prove the following:

a) Validity of formulas is decidable.

b) A formula is satisfiable if and only if its negation is not valid.

c) $\forall s.\ \mathsf{val}\ s + (\Sigma\alpha.\ \mathcal{E}\alpha s = \mathbf{F})$.

Exercise 31.4.10 If you are already familiar with well-founded recursion in computational type theory (Chapter 20), define a function $\mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$ satisfying the equations specifying the procedure dnf in Figure 31.2.

31.5 DNF Recursion

From the equations for the DNF procedure (Figure 31.2) and the construction of the basic DNF solver (Lemma 31.4.1) one can abstract out the recursion scheme shown in Figure 31.3. We refer to this recursion scheme as **DNF recursion**. DNF recursion has one clause for every equation of the DNF procedure in Figure 31.2 where the recursive calls appear as inductive hypotheses. DNF recursion simplifies the proof of Lemma 31.4.1. However, DNF recursion can also be used for other constructions (our main example is a completeness lemma (31.6.5) for a tableau system) given that it is formulated with an abstract type function p. Note that DNF recursion encapsulates the use of size recursion on the agenda, the set-up and justification of the case analysis, and the propagation of the precondition $\mathsf{solved}\,C$. We remark that all clauses can be equipped with the precondition, but for our applications the precondition is only needed in the clause for the empty agenda.

$$
\begin{array}{l}
\forall p^{\mathsf{Cla}\to\mathsf{Cla}\to\mathbb{T}} \\
(\forall C.\ \mathsf{solved}\,C \to pC[\,]) \to \\
(\forall CD.\ x^- \in C \to pC(x^+ :: D)) \to \\
(\forall CD.\ x^- \notin C \to p(x^+ :: C)D \to pC(x^+ :: D)) \to \\
(\forall CD.\ x^+ \in C \to pC(x^- :: D)) \to \\
(\forall CD.\ x^+ \notin C \to p(x^- :: C)D \to pC(x^- :: D)) \to \\
(\forall CD.\ pC(\bot^+ :: D)) \to \\
(\forall CD.\ pCD \to pC(\bot^- :: D)) \to \\
(\forall CD.\ pC(s^- :: D) \to pC(t^+ :: D) \to pC((s \to t)^+ :: D)) \to \\
(\forall CD.\ pC(s^+ :: t^- :: D) \to pC((s \to t)^- :: D)) \to \\
(\forall CD.\ pC(s^+ :: t^+ :: D) \to pC((s \wedge t)^+ :: D)) \to \\
(\forall CD.\ pC(s^- :: D) \to pC(t^- :: D) \to pC((s \wedge t)^- :: D)) \to \\
(\forall CD.\ pC(s^+ :: D) \to pC(t^+ :: D) \to pC((s \vee t)^+ :: D)) \to \\
(\forall CD.\ pC(s^- :: t^- :: D) \to pC((s \vee t)^- :: D)) \to \\
\forall CD.\ \mathsf{solved}\,C \to pCD
\end{array}
$$

Figure 31.3: DNF recursion scheme

Lemma 31.5.1 (DNF recursion)
The DNF recursion scheme shown in Figure 31.3 is inhabited.

Proof By size recursion on the σD with C quantified using the decidability of membership in clauses. Straightforward. ■

DNF recursion provides the abstraction level one would use in an informal correctness proof of the DNF procedure. In particular, DNF recursion separates the termination argument from the partial correctness argument. We remark that DNF recursion generalizes the functional induction scheme one would derive for a DNF procedure.

Exercise 31.5.2 Use DNF recursion to construct a certifying boolean solver for clauses: $\forall C.\ (\Sigma\alpha.\ \alpha \vDash C) + (\neg\mathsf{sat}(C))$.

$$\frac{\mathsf{tab}(S :: C +\!\!\!+ D)}{\mathsf{tab}(C +\!\!\!+ S :: D)} \qquad \frac{}{\mathsf{tab}(x^+ :: x^- :: C)} \qquad \frac{}{\mathsf{tab}(\bot^+ :: C)}$$

$$\frac{\mathsf{tab}(s^- :: C) \qquad \mathsf{tab}(t^+ :: C)}{\mathsf{tab}((s \to t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^+ :: t^- :: C)}{\mathsf{tab}((s \to t)^- :: C)}$$

$$\frac{\mathsf{tab}(s^+ :: t^+ :: C)}{\mathsf{tab}((s \wedge t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^- :: C) \qquad \mathsf{tab}(t^- :: C)}{\mathsf{tab}((s \wedge t)^- :: C)}$$

$$\frac{\mathsf{tab}(s^+ :: C) \qquad \mathsf{tab}(t^+ :: C)}{\mathsf{tab}((s \vee t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^- :: t^- :: C)}{\mathsf{tab}((s \vee t)^- :: C)}$$

Figure 31.4: Inductive type family $\mathsf{tab} : \mathsf{Cla} \to \mathbb{T}$

31.6 Tableau Refutations

Figure 31.4 defines an indexed inductive type family $\mathsf{tab} : \mathsf{Cla} \to \mathbb{T}$ for which we will prove

$$\mathsf{tab}(C) \;\Leftrightarrow\; \neg\mathsf{sat}(C)$$

We call the inhabitants of a type $\mathsf{tab}(C)$ **tableau refutations** for C. The above equivalence says that for every clause unsatisfiability proofs are inter-translatable with tableau refutations. Tableau refutations may be seen as explicit syntactic unsatisfiability proofs for clauses. Since we have $\neg\mathsf{sat}\, s \Leftrightarrow \neg\mathsf{sat}\,[s^+]$, tableau refutations may also serve as refutations for formulas.

We speak of **tableau refutations** since the type family tab formalizes a proof system that belongs to the family of tableau systems. We call the value constructors for the type constructor tab **tableau rules** and refer to type constructor tab as **tableau system**.

We may see the tableau rules in Figure 31.4 as a simplification of the equations specifying the DNF procedure in Figure 31.2. Because termination is no longer an issue, the accumulator argument is not needed anymore. Instead we have a tableau rule (the first rule) that rearranges the agenda.

We refer to the first rule of the tableau system 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 31.6.1 (Soundness)
Tableau refutable clauses are unsatisfiable: $\mathsf{tab}(C) \to \neg\mathsf{sat}(C)$.

Proof Follows by induction on tab. ■

For the completeness lemma we need a few lemmas providing derived rules for the tableau system.

Fact 31.6.2 (Clash)
All clauses containing a conflicting pair of signed variables are tableau refutable: $x^+ \in C \to x^- \in C \to \mathsf{tab}(C)$.

Proof Without loss of generality we have $C = C_1 + x^+ :: C_2 + x^- :: C_3$. The primitive clash rule gives us $\mathsf{tab}(x^+ :: x^- :: C_1 + C_2 + C_3)$. Using the move rule twice we obtain $\mathsf{tab}(C)$. ■

Fact 31.6.3 (Weakening)
Adding formulas preserves tableau refutability:
$\forall CS.\ \mathsf{tab}(C) \to \mathsf{tab}(S :: C)$.

Proof By induction on tab. ■

The move rule is strong enough to reorder clauses freely.

Fact 31.6.4 (Move Rules) The following rules hold for tab:

$$\frac{\mathsf{tab}(\mathsf{rev}\,D + C + E)}{\mathsf{tab}(C + D + E)} \qquad \frac{\mathsf{tab}(D + C + E)}{\mathsf{tab}(C + D + E)} \qquad \frac{\mathsf{tab}(C + S :: D)}{\mathsf{tab}(S :: C + D)}$$

We refer to the last rule as **inverse move rule**.

Proof The first rule follows by induction on D. 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 31.6.5 (Completeness)
$\forall DC.\ \mathsf{solved}\,C \to \neg\mathsf{sat}\,(D + C) \to \mathsf{tab}(D + C)$.

Proof By DNF recursion. The case for the empty agenda is contradictory since solved clauses are satisfiable. The cases with conflicting signed variables follow with the clash lemma. The cases with nonconflicting signed variables follow with the inverse move rule. The case for $\bot^-$ follows with the weakening lemma. ■

Theorem 31.6.6
A clause is tableau refutable if and only if it is unsatisfiable:
$\mathsf{tab}(C) \Leftrightarrow \neg\mathsf{sat}(C)$.

Proof Follows with Fact 31.6.1 and Lemma 31.6.5. ■

Corollary 31.6.7 $\forall C.\ \mathsf{tab}(C) + (\mathsf{tab}(C) \to \bot)$.

We remark that the DNF solver and the tableau system adapt to any choice of boolean connectives. We just add or delete cases 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 31.6.8 Prove $\mathsf{tab}(C + S :: D + T :: E) \longleftrightarrow \mathsf{tab}(C + T :: D + S :: E)$.

Exercise 31.6.9 Give an inductive type family deriving exactly the satisfiable clauses. Start with an inductive family deriving exactly the solved clauses.

31.7 Abstract Refutation Systems

An **unsigned clause** is a list of formulas. We will now consider a tableau system for unsigned clauses that comes close to the refutation system associated with natural deduction. For the tableau system we will show decidability and agreement with unsatisfiability. Based on the results for the tableau system one can prove decidability and completeness of classical natural deduction (Chapter 30).

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.

$$\frac{\rho\,(s :: A +\!\!\!+ B)}{\rho\,(A +\!\!\!+ s :: B)} \qquad \frac{}{\rho\,(x :: \neg x :: A)} \qquad \frac{}{\rho\,(\bot :: A)}$$

$$\frac{\rho\,(\neg s :: A) \qquad \rho\,(t :: A)}{\rho\,((s \to t) :: A)} \qquad \frac{\rho\,(s :: \neg t :: A)}{\rho\,(\neg(s \to t) :: A)}$$

$$\frac{\rho\,(s :: t :: A)}{\rho\,((s \wedge t) :: A)} \qquad \frac{\rho\,(\neg s :: A) \qquad \rho\,(\neg t :: A)}{\rho\,(\neg(s \wedge t) :: A)}$$

$$\frac{\rho\,(s :: A) \qquad \rho\,(t :: A)}{\rho\,((s \vee t) :: A)} \qquad \frac{\rho\,(\neg s :: \neg t :: A)}{\rho\,(\neg(s \vee t) :: A)}$$

Figure 31.5: Rules for abstract refutation systems $\rho : \mathcal{L}(\mathsf{For}) \to \mathbb{P}$

Fact 31.7.1 $\mathcal{E}\alpha\hat{C} = \mathcal{E}\alpha C$, $\mathcal{E}\alpha A^+ = \mathcal{E}\alpha A$, and $\widehat{A^+} = A$.

Fact 31.7.2 (Decidability) Satisfiability of unsigned clauses is decidable.

Proof Follows with Corollary 31.4.6 and $\mathcal{E}\alpha A^+ = \mathcal{E}\alpha A$. ■

We call a type family ρ on unsigned clauses an **abstract refutation system** if it satisfies the rules in Figure 31.5. 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 31.7.3 Let ρ be a refutation system. Then $\mathsf{tab}\ C \to \rho\hat{C}$.

Proof Straightforward by induction on $\mathsf{tab}\ C$. ■

Fact 31.7.4 (Completeness)
Every refutation system derives all unsatisfiable unsigned clauses.

Proof Follows with Theorem 31.6.6 and Lemma 31.7.3. ■

We call an abstract refutation system **sound** if it derives only unsatisfiable clauses (that is, $\forall A.\ \rho A \to \neg\mathsf{sat}\,A$).

Fact 31.7.5 A sound refutation system is decidable and derives exactly the unsatisfiable unsigned clauses.

Proof Facts 31.7.4 and 31.7.2. ■

Theorem 31.7.6 The minimal refutation system inductively defined with the rules for abstract refutation systems derives exactly the unsatisfiable unsigned clauses.

Proof Follows with Fact 31.7.4 and a soundness lemma similar to Fact 31.6.1. ■

Exercise 31.7.7 (Certifying Solver) Construct a function $\forall A.\ (\Sigma\alpha.\ \alpha \models A) + \mathsf{tab}\,A$.

Exercise 31.7.8 Show that boolean entailment

$$A \dot{\models} s \ := \ \forall\alpha.\ \alpha \models A \ \to \ \alpha \models s$$

is decidable.

Exercise 31.7.9 Let $A \dot{\vdash} s$ be the inductive type family 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 system.

32 Abstract Reduction Systems

32.1 Paths Types

We assume a relation $R : X \to X \to \mathbb{T}$. We see R as a graph whose vertices are the elements of X and whose edges are the pairs (x, y) such that Rxy. Informally, a **path in R** is a walk

$$x_0 \overset{R}{\to} x_1 \overset{R}{\to} \cdots \overset{R}{\to} x_n$$

through the graph described by R following the edges. We capture this design formally with an indexed inductive type

$$\begin{aligned}
&\mathsf{path}\,(x : X) : X \to \mathbb{T} \;::= \\
&\mid \mathsf{P}_1 : \;\mathsf{path}\,xx \\
&\mid \mathsf{P}_2 : \;\forall x'y.\, Rxx' \to \mathsf{path}\,x'y \to \mathsf{path}\,xy
\end{aligned}$$

The constructors are chosen such that that the elements of a **path type** $\mathsf{path}\,xy$ formalize the **paths from x to y**. The first argument of the type constructor path is a nonuniform parameter and the second argument of path is an **index**. The second argument cannot be made a parameter because it is **instantiated** to x by the value constructor P_1. Here are the full types of the constructors:

$$\begin{aligned}
\mathsf{path} &: \forall X^{\mathbb{T}}.\,(X \to X \to \mathbb{T}) \to X \to X \to \mathbb{T} \\
\mathsf{P}_1 &: \forall X^{\mathbb{T}}\,\forall R^{X\to X\to\mathbb{P}}\,\forall x^X.\;\mathsf{path}_{XR}\,xx \\
\mathsf{P}_2 &: \forall X^{\mathbb{T}}\,\forall R^{X\to X\to\mathbb{P}}\,\forall xx'y^X.\;Rxx' \to \mathsf{path}_{XR}\,x'y \to \mathsf{path}_{XR}\,xy
\end{aligned}$$

Note that the type constructor path takes three parameters followed by a single index as arguments. There is the general rule that parameters must go before indices.

We shall use notation with implicit arguments in the following. It is helpful to see the value constructors in simplified form as inference rules:

$$\mathsf{P}_1\;\frac{}{\mathsf{path}_R\,xx} \qquad\qquad \mathsf{P}_2\;\frac{Rxx' \qquad \mathsf{path}_R\,x'y}{\mathsf{path}_R\,xy}$$

The second constructor is reminiscent of a cons for lists. The premise Rxx' ensures that adjunctions are licensed by R. And, in contrast to plain lists, the endpoints of a path are recorded in the type of the path.

Fact 32.1.1 (Step function) $\forall xy.\ Rxy \to \mathsf{path}_R xy$.

Proof The function claimed can be obtained with the value constructors P_1 and P_2:

$$\dfrac{Rxy \qquad \dfrac{}{\mathsf{path}_R\, yy}\,\mathsf{P}_1}{\mathsf{path}_R\, xy}\,\mathsf{P}_2$$

■

We now define an inductive function len that yields the length of a path (i.e., the number of edges the path runs trough).

$$\begin{aligned} \mathsf{len} &: \forall xy.\ \mathsf{path}\, xy \to \mathsf{N} \\ \mathsf{len}\, x\, \_\, (\mathsf{P}_1 \_) &:= 0 \\ \mathsf{len}\, x\, \_\, (\mathsf{P}_2 \_\, x'\, y\, r\, a) &:= \mathsf{S}(\mathsf{len}\, x'\, y\, a) \end{aligned}$$

Note the underlines in the patterns. The underlines after P_1 and P_2 are needed since the first arguments of the constructors are parameters (instantiated to x by the pattern). The underlines before the applications of P_1 and P_2 are needed since the respective argument is an **index argument**. The index argument appears as variable y in the type declared for len. We refer to y (in the type of len) as **index variable**. What identifies y as index variable is the fact that it appears as index argument in the type of the discriminating argument. The index argument must be written as underline in the patterns since the succeeding pattern for the discriminating argument determines the index argument. There is the general constraint that the index arguments in the type of the discriminating argument must be variables not occurring otherwise in the type of the discriminating argument (the so-called **index condition**). Moreover, the declared type must be such that all index arguments are taken immediately before the discriminating argument.

Type checking elaborates the defining equations into quantified propositional equations where the pattern variables are typed and the underlines are filled in. For the defining equations of len, elaboration yields the following equations:

$$\begin{aligned} \forall x^{\mathsf{N}}.\ \mathsf{len}\, x\, x\, (\mathsf{P}_1\, x) &= 0 \\ \forall x x' y^{\mathsf{N}}\, \forall r^{Rxx'}\, \forall a^{\mathsf{path}\, x' y}.\ \mathsf{len}\, x\, y\, (\mathsf{P}_2\, x\, x'\, y\, r\, a) &= \mathsf{S}(\mathsf{len}\, x'\, y\, a) \end{aligned}$$

We remark that the underlines for the parameters are determined by the declared type of the discriminating argument, and that the underlines for the index arguments are determined by the elaborated type for the discriminating argument.

We now define an append function for paths

$$\mathsf{app} : \forall zxy.\ \mathsf{path}\, xy \to \mathsf{path}\, yz \to \mathsf{path}\, xz$$

discriminating on the first path. The declared type and the choice of the discriminating argument (not explicit yet) identify y as an index variable and fix an index argument for app. Note that the index condition is satisfied. The argument z is taken first so that the index argument y can be taken immediately before the discriminating argument. We can now write the defining equations:

$$\begin{aligned} \mathsf{app}\, zx\, \_\, (\mathsf{P}_1\, \_) &:= \lambda b.b & &: \mathsf{path}\, xz \to \mathsf{path}\, xz \\ \mathsf{app}\, zx\, \_\, (\mathsf{P}_2\, \_\, x'y\, ra) &:= \lambda b.\, \mathsf{P}_2\, xx'z\, r(\mathsf{app}\, zx'y\, ab) & &: \mathsf{path}\, yz \to \mathsf{path}\, xz \end{aligned}$$

As always, the patterns are determined by the declared type and the choice of the discriminating argument. We have the types $r : Rxx'$ and $a : \mathsf{path}\, x'y$ for the respective pattern variables of the second equation. Note that the index argument is instantiated to x in the first equation and to y in the second equation.

We would now like to verify the equation

$$\forall xyz\, \forall a^{\mathsf{path}\, xy}\, \forall b^{\mathsf{path}\, yz}.\ \mathsf{len}\,(\mathsf{app}\, ab) = \mathsf{len}\, a + \mathsf{len}\, b$$

which is familiar from lists. As for lists, the proof is by induction on a. Doing the proof by hand, ignoring the type checking, is straightforward. After conversion, the case for P_2 gives us the proof obligation

$$\mathsf{S}(\mathsf{len}\,(\mathsf{app}\, ab)) = \mathsf{S}(\mathsf{len}\, a + \mathsf{len}\, b)$$

which follows by the inductive hypothesis, Formally, the induction can be validated with the universal eliminator for path:

$$\begin{aligned} E :\ & \forall p^{\forall xy.\ \mathsf{path}\, xy \to \mathbb{T}}. \\ & (\forall x.\ pxx(\mathsf{P}_1\, x)) \to \\ & (\forall xyz\, \forall r^{Rxy}\, \forall a^{\mathsf{path}\, yz}.\ pxz(\mathsf{P}_2\, xyzra)) \to \\ & \forall xya.\ pxya \end{aligned}$$

$$\begin{aligned} E\, pe_1e_2\, x\, \_\, , (\mathsf{P}_1\, \_) &:= e_1x \\ E\, pe_1e_2\, x\, \_\, (\mathsf{P}_2\, \_\, x'y\, r\, a) &:= e_2\, xx'y\, r(E\, pe_1e_2\, x'y\, a) \end{aligned}$$

Not that the type function p takes the nonuniform parameter, the index, and the discriminating argument as arguments. The general rule to remember here is that all nonuniform parameters and all indices appear as arguments of the return type function of the universal eliminator. As always with universal eliminators, the defining equations follow from the type of the eliminator, and the types of the continuation functions e_1 and e_2 follow from the types of the value constructors and the type of the return type function.

No doubt, type checking the above examples by hand is a tedious exercise, also for the author. In practice, one leaves the type checking to the proof assistant and designs the proofs assuming that the type checking works out. With trained intuitions, this works out well.

Exercise 32.1.2 Give the propositional equations obtained by elaborating the defining equations for len, app, and E. Hint: The propositional equations for len are explained above. Use the proof assistant to construct and verify the equations.

Exercise 32.1.3 Define the step function asserted by Fact 32.1.1 with a term.

Exercise 32.1.4 (Index eliminator) Define an index eliminator for path:

$$\begin{aligned}
&\forall p^{X \to X \to \mathbb{T}}.\\
&(\forall x.\ pxx) \to\\
&(\forall x x' y.\ Rxx' \to px'y \to pxy) \to\\
&(\forall xy.\ \mathsf{path}\,xy \to pxy)
\end{aligned}$$

Note that the type of the index eliminator is obtained from the type of the universal eliminator by deleting the dependencies on the paths.

Exercise 32.1.5 Use the index eliminator to prove that the relation path is transitive: $\forall xyz.\ \mathsf{path}\,xy \to \mathsf{path}\,yz \to \mathsf{path}\,xz$.

Exercise 32.1.6 (Arithmetic graph) Let $Rxy := (\mathsf{S}x = y)$. We can see R as the graph on numbers having the edges $(x, \mathsf{S}x)$. Prove $\mathsf{path}_R\,xy \Leftrightarrow x \le y$.
Hints. Direction $\Rightarrow$ follows with index induction (i.e., using the index eliminator from Exercise 32.1.4). Direction $\Leftarrow$ follows with $\forall k.\ \mathsf{path}_R\,x(k + x)$, which follows by induction on k with x quantified.

32.2 Reflexive Transitive Closure

We can see the type constructor path as a function that maps relations $X \to X \to \mathbb{T}$ to relations $X \to X \to \mathbb{T}$. We will write R^* for path_R in the following and speak of the **reflexive transitive closure** of R. We will explain later why this speak is meaningful.

We first note that R^* is reflexive. This fact is stated by the type of the value constructor P_1.

We also note that R^* is transitive. This fact is stated by the type of the inductive function app.

Moreover, we note that R^* contains R (i.e., $\forall xy.\ Rxy \to R^*xy$). This fact is stated by Fact 32.1.1.

Fact 32.2.1 (Star recursion)
Every reflexive and transitive relation containing R contains R^*:
$\forall p^{X \to X \to \mathbb{T}}.\ \mathsf{refl}\, p \to \mathsf{trans}\, p \to R \subseteq p \to R^* \subseteq p$.

Proof Let p be a relation as required. We show $\forall xy.\, R^*xy \to pxy$ using the index eliminator for path (Exercise 32.1.4). Thus we have to show that p is reflexive, which holds by assumption, and that $\forall xx'y.\, Rxx' \to px'y \to pxy$. So we assume Rxx' and $px'y$ and show pxy. Since p contains R we have pxx' and thus we have the claim since p is transitive. ■

Star recursion as stated by Fact 32.2.1 is a powerful tool. The function realized by star recursion is yet another eliminator for path. We can use star recursion to show that R^* and $(R^*)^*$ agree.

Fact 32.2.2 R^* and $(R^*)^*$ agree.

Proof We have $R^* \subseteq (R^*)^*$ by Fact 32.1.1. For the other direction $(R^*)^* \subseteq R^*$ we use star recursion (Fact 32.2.1). Thus we have to show that R^* is reflexive, transitive, and contains R^*. We have argued reflexivity and transitivity before, and the containment is trivial. ■

Fact 32.2.3 R^* is a least reflexive and transitive relation containing R.

Proof This fact is a reformulation of what we have just shown. On the one hand, it says that R^* is a reflexive and transitive relation containing R. On the other hand, it says that every such relation contains R^*. This is asserted by star recursion. ■

If we assume function extensionality and propositional extensionality, Fact 32.2.2 says $R^* = (R^*)^*$. With extensionality R^* can be understood as a closure operator which for R yields the unique least relation that is reflexive, transitive, and contains R. In an extensional setting, R^* is commonly called the reflexive transitive closure of R.

We have modeled relations as general type functions $X \to X \to \mathbb{T}$ rather than as predicates $X \to X \to \mathbb{P}$. Modeling path types R^*xy as computational types gives us paths as computational values and provides for computational recursion on paths as it is needed for the length function len. If we switch to propositional relations $X \to X \to \mathbb{P}$, everything we did carries over except for the length function.

Exercise 32.2.4 (Functional characterization)
Prove $R^*xy \iff \forall p^{X \to X \to \mathbb{T}}.\ \mathsf{refl}\, p \to \mathsf{trans}\, p \to R \subseteq p \to pxy$.

Appendix: Inductive Definitions

We collect technical information about inductive definitions here. Inductive definitions come in two forms, inductive type definitions and inductive function definitions. Inductive type definitions introduce typed constants called constructors, and inductive function definitions introduce typed constants called eliminators. Inductive function definitions come with defining equations serving as computation rules. Inductive definitions are designed such that they preserve consistency.

Inductive Type Definitions

An inductive type definition introduces a system of typed constants consisting of a **type constructor** and $n \geq 0$ **value constructors**. The type constructor must target a universe, and the value constructors must target the type constructor. The first $n \geq 0$ arguments of the type constructor may be declared as **parameters**.

Parameter condition: Each value constructor must take the parameters as leading arguments and must target the type constructor applied to the parameters.

Strict positivity condition: If a value constructor uses the type constructor in an argument type, a path to the type constructor must not go through the left-hand side of a function type.

Dominance condition: If the type constructor targets a universe $\mathbb{T}_i$, the types of the nonparametric arguments of the value constructors must be in $\mathbb{T}_i$.

An **inductive type** is a type obtained with a type constructor. The nonparametric arguments of a type constructor are called **indices**.

Inductive Function Definitions

An inductive function definition introduces a functional constant called an **eliminator** together with a system of **defining equations** serving as computation rules. An eliminator must be defined with a functional type, a number of *required arguments*, and a distinguished required argument called the **principal argument**. The type of an eliminator must have the form

$$\forall x_1 \dots x_k \, \forall y_1 \dots y_m. \; c s_1 \dots s_n y_1 \dots y_m \;\to\; t$$

where the following conditions are satisfied:

- $cs_1 \dots s_m x_1 \dots x_n$ types the principal argument.
- c is a type constructor with $m \geq 0$ parameters and $n \geq 0$ indices.
- The **index variables** $y_1, \dots, y_m$ are distinct and do not occur in $s_1, \dots, s_n$.
- **Elimination restriction**: If c is sealing, t must be propositional. A type constructor is **sealing** if it is propositional and has either more than one proof constructor, or a single proof constructor with a nonparametric argument that is not propositional.

We refer to the conditions for principal arguments whose type is indexed as **index condition**.

For every value constructor of c a defining equation must be provided, where the pattern and the target type of the defining equations are determined by the type of the eliminator, the position of principal argument, and the number of arguments succeeding the principle argument. Each pattern contains exactly two constants, the eliminator and a value constructor in the position of the principal argument. Patterns must be linear (no variable appears twice) and must give the index arguments of the eliminator and the parametric arguments of the value constructor with underlines.

Every defining equation must satisfy the **guard condition**, which constrains the recursion of the eliminator to be structural on the principal argument. The guard condition must be realized as a decidable condition. There are different possibilities for the guard condition. In this text we have been using the strictest form of the guard condition.

Remarks

1. The format for eliminators is such that **universal eliminators** can be defined that can express all other eliminators.
2. The special case of zero value constructors is redundant. A proposition $\perp$ with an eliminator $\perp \to \forall X^{\mathbb{T}}. X$ can be defined with a single proof constructor $\perp \to \perp$.
3. Assuming type definitions at the computational level, accommodating type definitions at the propositional level adds the sealing condition. Note that the dominance condition is vacuously satisfied for propositional type definitions.
4. Defining equations with a secondary case analysis (e.g., subtraction) are a syntactic convenience. They can be expressed with auxiliary functions defined as eliminators.
5. Our presentation of inductive definitions is compatible with Coq but takes away some of the flexibility provided by Coq. Our format requires that in Coq terms a fix is directly followed by a match on the principal argument. This excludes a

direct definition of Euclidean division. It also excludes the (redundant) eager recursion pattern frequently used for well-founded recursion in the Coq literature.

Appendix: Basic Definitions

We summarize basic definitions concerning functions and predicates. We make explicit the generality coming with dependent typing. As it comes to arity, we state the definitions for the minimal number of arguments and leave the generalization to more arguments to the reader (as there is no formal possibility to express this generalization).

For functions $f : \forall x^X.\, px$ we define:

$\mathsf{injective}\,(f)$	$:=$	$\forall x x'.\, fx = fx' \to x = x'$	**injectivity**
$\mathsf{surjective}\,(f)$	$:=$	$\forall y\, \exists x.\, fx = y$	**surjectivity**
$\mathsf{bijective}\,(f)$	$:=$	$\mathsf{injective}\,(f) \land \mathsf{surjective}\,(f)$	**bijectivity**
$f \equiv f'$	$:=$	$\forall x.\, fx = fx'$	**agreement**

The definitions extend to functions with $n \geq 2$ arguments as one would expect. Note that injectivity, surjectivity, and bijectivity are invariant under agreement.

For binary predicates $P : \forall x^X.\, px \to \mathbb{P}$ we define:

$\mathsf{functional}\,(P)$	$:=$	$\forall x y y'.\, Pxy \to Pxy' \to y = y'$	**functionality**
$\mathsf{total}\,(P)$	$:=$	$\forall x\, \exists y.\, Pxy$	**totality**

The definitions extend to predicates with $n \geq 2$ arguments as one would expect.

For unary predicates $P, Q : X \to \mathbb{P}$ we define:

$P \subseteq Q$	$:=$	$\forall x.\, Px \to Qx$	**respect**
$P \equiv Q$	$:=$	$\forall x.\, Px \longleftrightarrow Qx$	**agreement**

The definitions extend to predicates with $n \geq 2$ arguments as one would expect.

For functions $f : \forall x^X.\, px$ and predicates $P : \forall x^X.\, px \to \mathbb{P}$:

$f \subseteq P$	$:=$	$\forall x.\, Px(fx)$	**respect**

The definitions extend to functions with $n \geq 2$ arguments and predicates with $n + 1$ arguments as one would expect.

The following facts have straightforward proofs:

1. $P \subseteq Q \to \mathsf{functional}\,(Q) \to \mathsf{functional}\,(P)$
2. $P \subseteq Q \to \mathsf{functional}\,(Q) \to \mathsf{total}\,(P) \to P \equiv Q$
3. $P \subseteq Q \to \mathsf{total}\,(P) \to \mathsf{total}\,(Q)$
4. $f \subseteq P \to \mathsf{functional}\,(P) \to (\forall xy.\, Pxy \longleftrightarrow fx = y)$

Appendix: Favorite Proofs

Here is a list of remarkable proofs the author was exited about when he first understood them.

1. Well-founded size recursion (Corollary 20.2.4).

Bibliography

[1] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[2] Peter Aczel. The Type Theoretic Interpretation of Constructive Set Theory. *Studies in Logic and the Foundations of Mathematics*, 96:55–66, January 1978.

[3] 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.

[4] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[5] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[6] Thierry Coquand. Metamathematical investigations of a calculus of constructions, 1989.

[7] Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-related computational reductions in Coq. In *Interactive Theorem Proving (ITP 2018), Oxford*, LNCS 10895, pages 253–269. Springer, 2018.

[8] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *CPP 2019, Lisbon, Portugal*, 2019.

[9] 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.

[10] 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.

[11] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

[12] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *LICS 1994*, pages 208–212, 1994.

[13] 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.

[14] Nicolai Kraus, Martín Hötzel Escardó, Thierry Coquand, and Thorsten Altenkirch. Generalizations of Hedberg's theorem. In *Proceedings of TLCA 2013*, volume 7941 of *LNCS*, pages 173–188. Springer, 2013.

[15] Edmund Landau. *Grundlagen der Analysis: With Complete German-English Vocabulary*, volume 141. American Mathematical Soc., 1965.

[16] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[17] Bengt Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619, Sep 1988.

[18] Raymond M. Smullyan and Melvin Fitting. *Set Theory and the Continuum Hypothesis*. Dover, 2010.

[19] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory*. Cambridge University Press, 2nd edition, 2000.

[20] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[21] Louis Warren, Hannes Diener, and Maarten McKubre-Jordens. The drinker paradox and its dual. *CoRR*, abs/1805.06216, 2018.