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

Textbook under Construction
Version of September 1, 2023

Gert Smolka
Saarland University

Contents

Preface

This text teaches topics in computational logic computer scientists 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 models with machine-checked proofs. As common with programming languages, we teach foundations, 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 proof rules of the theory are intuitionistic while assuming the law of excluded middle is possible. As it will become apparent through case studies in this text, 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, an implementation of 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 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 abstracting from 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 type-theoretic design principles from engineering aspects coming with the Coq language. For instance, we will have equational inductive function definitions at the mathematical level and realize them with Coq's primitives at the coding level. This way we get mathematically satisfying function definitions and a fine explanation of Coq's pattern matching construct.

Acknowledgements

This text has been written for the course *Introduction to Computational Logic* I teach every summer semester at Saarland University (since 2003). In 2010, we switched to computational type theory and the proof assistant Coq. From 2010–2014 I taught the course together with Chad E. Brown and we produced lecture notes discussing Coq developments (in the style of Benjamin Pierce's Software Foundations at the time). It was great fun to explore with Chad intuitionistic reasoning and the propositions as types paradigm. It was then I learned about impredicative characterizations, Leibniz equality, and natural deduction. Expert advice on Coq often came from Christian Doczkal.

By Summer 2017 I was dissatisfied with the programming-centered approach we had followed so far and started writing lectures notes using mathematical language. There were also plenty of exciting things about type theory I still had to learn. My chief teaching assistants during this time were Yannick Forster (2017), Dominik Kirst (2018-2020), and Andrej Dudenhefner (2021), all of them contributing exercises and ideas to the text.

My thanks goes to the undergraduate and graduate students who took the course and who worked on related topics with me, and to the persons who helped me teach the course. You provided the challenge and motivation needed for the project. And the human touch making it fun and worthwhile.

Part I

Basics

1 Getting Started

We start with basic ideas from computational type theory and Coq. The main issues we discuss are inductive types, structural recursion, and equational reasoning with structural induction. We will see inductive types for booleans, natural numbers, and pairs. On inductive types we will define inductive functions using equations and structural case analysis. This will involve functions that are cascaded, recursive, 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 functional computation always terminates.

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 simplification, 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 computational type theory, set-theoretic explanations (e.g., of functions) are often not helpful, so free your mind for a foundational restart.

1.1 Booleans

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

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

introduces three typed constants called **constructors**:

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

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

Inductive types provide for the definition of **inductive functions**, where a **defining equation** is given for each value constructor. Our first example for an inductive function is a boolean negation function:

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

There is a **defining equation** for each of the two value constructors of B. We say that an inductive function is defined by **discrimination** on an **inductive argument** (an argument that has an inductive type). There must be exactly one defining equation for every value constructor of the type of the inductive argument the function **discriminates** on. In the literature, discrimination is known as **structural case analysis**.

The defining equations of an inductive function 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 defining equation of $!$. Note that $!!!\mathbf{T}$ is to be read as $!(!(!\mathbf{T}))$, and that the first rewrite step replaces the subterm $!\mathbf{T}$ with $\mathbf{F}$. 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 a **proof by computational equality**.

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 **discrimination** (i.e., case analysis) on the boolean variable x, which reduces the claim $!!x = x$ to the equations $!!\mathbf{T} = \mathbf{T}$ and $!!\mathbf{F} = \mathbf{F}$, which both hold by computational equality.

Next we define inductive functions for boolean conjunction and boolean disjunction:

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

Both functions discriminate on their first argument. Alternatively, one could define the functions by discrimination on the second argument, resulting in different computation rules. There is the general principle that computation rules must be **disjoint** (at most one computation rule applies to a given term).

The left hand sides of defining equations are called **patterns**. Often, patterns **bind variables** that can be used in the right hand side of the equation. The patterns of the defining equations for $\&$ and $|$ each bind the variable y.

Given the definitions of the basic boolean connectives, we can prove the usual boolean indenties with discrimination and computational equality. For instance, the distributivity law

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

follows by discrimination 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 valid equations.

1.2 Numbers

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

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

introduces three constructors

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

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. The constructor representation of numbers dates back to the Dedekind-Peano axioms.

We will use the familiar notations $0, 1, 2, \dots$ for the terms $0, \mathsf{S}0, \mathsf{S}(\mathsf{S}0), \dots$. Moreover, we will take the freedom to write terms like $\mathsf{S}(\mathsf{S}(\mathsf{S}x))$ without parentheses as $\mathsf{SSS}x$.

We define an inductive **addition** function discriminating 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** since it uses the function '+' being defined at the right hand side.

Computational type theory does not admit partial functions. To fulfill this design principle, recursion must always terminate. To ensure termination, recursion is restricted to inductive functions and must act on a single discriminating argument. One speaks of **structural recursion**. Recursive applications must be on variables introduced by the constructor of the pattern of the discriminating argument. In the above definitions of '+', only the variable x in the second defining equation qualifies for recursion. Intuitively, structural recursion terminates since every recursion step skips a constructor of the recursive argument. The condition for structural recursion can be checked automatically by a proof assistant.

We define **truncating subtraction** for numbers:

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

This time we have two discriminating arguments (we speak of a **cascaded discrimination**). The primary discrimination is on the first argument, followed by a secondary discrimination on the second argument in the successor case. The recursion is on the first argument. We require that a structural recursion is always on the first discriminating argument.

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

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

Exercise 1.2.1 Define functions as follows:

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

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

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

Exercise 1.2.2 (Symmetric boolean conjunction and disjunction) Using cascaded discrimination, we can define an inductive function for boolean conjunction with symmetric defining equations:

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

a) Prove that the symmetric function satisfies the defining equations for the standard boolean conjunction function ($\mathbf{T} \,\&\, y = y$ and $\mathbf{F} \,\&\, y = \mathbf{F}$).

b) Prove that the symmetric function agrees with the standard boolean conjunction function.

c) Define a symmetric boolean disjunction function and show that it agrees with the standard boolean disjunction function.

1.3 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:

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

For the arithmetic operations we assume the usual precedences, so multiplication '·' binds before addition '+' and subtraction '−', and all three of them are left associative. For instance:

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

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

Figure 1.1: Proof diagram for Equation 1.1

1.4 Structural Induction

We will now 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 two subgoals are obtained by discrimination on x 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 simplification and by rewriting with the inductive hypothesis. A **simplification step** simplifies a claim by applying defining equations from left to right. A **rewriting step** rewrites with an equation that is either assumed or has been established as a lemma. In the example above, rewriting takes place with the inductive hypothesis, an assumption introduced by the induction rule.

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

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

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

Figure 1.2 shows a proof diagram giving an inductive proof of Equation 1.4. Note that the proof of the base case involves a structural case analysis on x so that the defining equations for subtraction apply. Also note that the proof rewrites

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

Figure 1.2: Proof diagram for Equation 1.4

with Equation 1.1 and Equation 1.2, assuming that the equations have been proved before. The successor case closes with an application of the inductive hypothesis (i.e., the remaining claim agrees with the inductive hypothesis).

We remark that a structural case analysis in a proof (as in Figure 1.2) may also be called a *discrimination* or a **destructuring**.

The proof of Equation 1.3 is similar to the proof of Equation 1.4 (induction on x and rewriting with 1.1 and 1.2). We leave the proof as exercise.

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 the 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 digrams** since they come with separate columns for assumptions we *have*, claims we *want* to prove, and actions we perform to advance the proof.

Exercise 1.4.1 Give a proof diagram for Equation 1.2. Follow the layout of Figure 1.2.

[1]In this section, only inductive hypotheses appear as assumption. We will see more assumptions once we prove claims with implication in Chapter 3.

Exercise 1.4.2 Prove that addition is commutative (1.3). Use equations (1.1) and (1.2) as lemmas.

Exercise 1.4.3 Shorten the given proofs for Equations 1.1 and 1.4 by applying the inductive hypothesis from right to left thus avoiding the simplification step.

Exercise 1.4.4 Prove that addition is associative: $(x + y) + z = x + (y + z)$. Give a proof diagram.

Exercise 1.4.5 Prove the distributivity law $(x + y) \cdot z = x \cdot z + y \cdot z$. You will need associativity of addition.

Exercise 1.4.6 Prove that multiplication is commutative. You will need lemmas.

Exercise 1.4.7 (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) $x - (x + y) = 0$

c) $x - x = 0$

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

Hint: (d) follows with equations shown before.

1.5 Quantified Inductive Hypotheses

Sometimes it is necessary to do an inductive proof using a quantified inductive hypothesis. As an example we consider a variant of the subtraction function returning the distance between two numbers:

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

The defining equations discriminate on the first argument and in the successor case also on the second argument. The recursion occurs in the third equation and is structural in the first argument.

We now want to prove

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

		$\forall y.\ Dxy = (x - y) + (y - x)$	induction x
1		$\forall y.\ D0y = (0 - y) + (y - 0)$	disc. y
1.1		$D00 = (0 - 0) + (0 - 0)$	comp. eq.
1.2		$D0(\mathsf{S}y) = (0 - \mathsf{S}y) + (\mathsf{S}y - 0)$	comp. eq.
2	IH : $\forall y. \cdots$	$\forall y.\ D(\mathsf{S}x)y = (\mathsf{S}x - y) + (y - \mathsf{S}x)$	disc. y
2.1		$D(\mathsf{S}x)0 = (\mathsf{S}x - 0) + (0 - \mathsf{S}x)$	simpl.
		$\mathsf{S}x = \mathsf{S}x + 0$	apply (1.1)
2.2		$D(\mathsf{S}x)(\mathsf{S}y) = (\mathsf{S}x - \mathsf{S}y) + (\mathsf{S}y - \mathsf{S}x)$	simpl.
		$Dxy = (x - y) + (y - x)$	apply IH

Figure 1.3: Proof diagram for a proof with a quantified inductive hypothesis

We do the proof by induction on x followed by discrimination on y. The base cases with either $x = 0$ or $y = 0$ are easy. The interesting case is

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

After simplification (i.e., application of defining equations) we have

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

If this was the inductive hypothesis, closing the proof is trivial. However, the actual inductive hypothesis is

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

since it was instantiated by the discrimination on y. The problem can be solved by starting with a quantified claim

$$\forall y.\ Dxy = (x - y) + (y - x)$$

where induction on x gives us a quantified inductive hypothesis that is not affected by a discrimination on y. Figure 1.3 shows a complete proof diagram for the quantified claim.

You may have questions about the precise rules for quantification and induction. Given that this is a teaser chapter, you will have to wait a little bit. It will take until Chapter 6 that quantification and induction are explained in depth.

Exercise 1.5.1 Prove $Dxy = Dyx$ by induction on x. No lemma is needed.

Exercise 1.5.2 (Maximum)
Define an inductive maximum function $M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ and prove the following:

a) $Mxy = Myx$ (commutativity)

b) $M(x+y)x = x+y$ (dominance)

Hint: Commutativity needs a quantified inductive hypothesis.
Extra: Do the exercise for a minimum function. Find a suitable reformulation for (b).

Exercise 1.5.3 (Symmetric addition) Using cascaded discrimination, we can define an inductive addition function with symmetric defining equations:

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

a) Prove that the symmetric addition function is commutative: $x + y = y + x$.

b) Prove that the symmetric addition function satisfies the defining equations for the standard addition function ($0 + y = y$ and $\mathsf{S}x + y = \mathsf{S}(x + y)$).

c) Prove that the symmetric addition function agrees with the standard addition function.

1.6 Procedural Specifications

The rules we have given for defining inductive functions are very restrictive as it comes to termination. There are many cases where a function can be specified with a system of equations that are exhaustive, disjoint, and terminating. We then speak of a **procedural specification** and its **specifying equations**. It turns out that in practice using strict structural recursion one can construct inductive functions satisfying procedural specifications relying on more permissive termination arguments.

Our first example for a procedural specification specifies a function $E : \mathsf{N} \to \mathsf{B}$ that checks whether a number is even:

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

The equations are exhaustive, disjoint, and terminating (two constructors are skipped). However, the equations cannot serve as defining equations for an inductive function since the recursion skips two constructors (rather that just one).

We can define an inductive function satisfying the specifying equations using the defining equations

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

(recall that '!' is boolean negation). The first and the second equation specifying E hold by computational equality. The third specifying equation holds by simplification and by rewriting with the lemma $!!b = b$.

Our second example specifies the **Fibonacci function** $F : \mathsf{N} \to \mathsf{N}$ with the equations

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

The equations do not qualify as defining equations for the same reasons we explained for E. It is however possible to define a Fibonacci function using strict structural recursion. One possibility is to obtain F with a helper function F' taking an extra boolean argument such that, informally, $F'nb$ yields $F(n+b)$:

$$\begin{aligned} F' :\ & \mathsf{N} \to \mathsf{B} \to \mathsf{N} \\ F'0\,\mathbf{F} \ &:=\ 0 \\ F'0\,\mathbf{T} \ &:=\ 1 \\ F'(\mathsf{S}n)\,\mathbf{F} \ &:=\ F'n\,\mathbf{T} \\ F'(\mathsf{S}n)\,\mathbf{T} \ &:=\ F'n\,\mathbf{F} + F'n\,\mathbf{T} \end{aligned}$$

Note that F' is defined by a cascaded discrimination on both arguments. We now define

$$\begin{aligned} F :\ & \mathsf{N} \to \mathsf{N} \\ F\,n \ &:=\ F'n\,\mathbf{F} \end{aligned}$$

That F satisfies the specifying equations for the Fibonacci function follows by computational equality.

Note that F is defined with a single defining equation without a discrimination. We speak of a **plain function** and a **plain function definition**. Since there is no discrimination, the defining equation of a plain function can be applied as soon as the function is applied to enough arguments. The defining equation of a plain function must not be recursive.

There are other possibilities for defining a Fibonacci function. Exercise 1.9.8 will obtain a Fibonacci function by iteration on pairs, and Exercise 1.11.5 will obtain a

Fibonacci function with a tail recursive helper function taking two extra arguments. Both alternatives employ linear recursion, while the definition shown above uses binary recursion, following the scheme of the third specifying equation.

We remark that Coq supports a more permissive scheme for inductive functions, providing for a straightforward definition of a Fibonacci function essentially following the specifying equations. In this text we will stick to the restrictive format explained so far. It will turn out that every function specified with a terminating system of equations can be defined in the restrictive format we are using here (see Chapter 31).

Exercise 1.6.1 Prove $E(n \cdot 2) = \mathbf{T}$.

Exercise 1.6.2 Verify that $Fn := F'n\,\mathbf{F}$ satisfies the specifying equations for the Fibonacci function.

Exercise 1.6.3 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 a helper function with an extra boolean argument.

1.7 Pairs and Polymorphic Functions

We have seen that booleans and numbers can be accommodated as inductive types. We will now see that 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. For instance, 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 be defined with a single equation:

$$\mathsf{swap} : \mathsf{Pair} \to \mathsf{Pair}$$
$$\mathsf{swap}\ (\mathsf{pair}\ x\ y)\ := \ \mathsf{pair}\ y\ x$$

Using discrimination 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 discrimination for pairs involves only a single case for the 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:

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

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 fix the types of the third and the fourth argument. Put differently, the types X and Y taken as first and second argument are the types for the components of the pair constructed. We say that the first and second argument of the value constructor pair are **parametric** and the third and fourth are **proper**.

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:

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

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

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

Note that the parametric arguments of pair are omitted in the **pattern** of the defining equation (i.e, the left hand side of the defining equation). The reason for the omission is that the parametric arguments of pair don't contribute relevant information in the pattern of a defining equation.

1.8 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 provide 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.

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:

$$\mathsf{swap} : \forall X\,Y.\, X \times Y \to Y \times X$$
$$\mathsf{swap}\;(x, y) \;:=\; (y, x)$$

Note that it takes considerable effort to recover the usual mathematical notation for pairs in the typed setting of computational type theory. There were three successive steps:

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

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

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

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

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

by destructuring of a (i.e., replacing a with (x, y)) and computational equality. Recall that a destructuring step is a discrimination step.

Exercise 1.8.1 Write the η-law and the definitions of the projections without using the notation (x, y) and without implicit arguments.

Exercise 1.8.2 Let a be a variable of type $X \times Y$. Write proof diagrams for the equations $\mathsf{swap}\,(\mathsf{swap}\,a) = a$ and $(\pi_1 a, \pi_2 a) = a$.

1.9 Iteration

If we look at the equations (all following by computational equality)

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

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.

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

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

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

Figure 1.4: Correctness of multiplication with iter

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

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

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

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

Figure 1.4 gives a proof diagram for the equation for multiplication.

Exercise 1.9.1 Check that $\text{iter}\ \mathsf{S}\ 2 = \lambda x.\, \mathsf{S}(\mathsf{S}x)$ holds by computational equality.

Exercise 1.9.2 Prove $n + x = \text{iter}\ \mathsf{S}\ n\ x$ and $x^n = \text{iter}\ (\cdot\, x)\ n\ 1$ by induction.

Exercise 1.9.3 Check that the plain function

$$\begin{aligned} &\text{add} : \ \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &\text{add}\, x\, y \ := \ \text{iter}\ \mathsf{S}\ x\ y \end{aligned}$$

satisfies the defining equations for inductive addition

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

by computational equality.

Exercise 1.9.4 (Shift) Prove $\text{iter}\ f\ (\mathsf{S}n)\ x = \text{iter}\ f\ n\ (fx)$.

Exercise 1.9.5 (Tail recursive iteration) Define a tail recursive version of iter and verify that it agrees with iter.

Exercise 1.9.6 (Even) The term $!^n\,\mathbf{T}$ tests whether a number n is even ('!' is boolean negation). Prove $\mathsf{iter}\,!\,(n\cdot 2)\ b = b$ and $\mathsf{iter}\,!\,(\mathsf{S}(n\cdot 2))\ b = !\,b$.

Exercise 1.9.7 (Factorials with iteration) 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 1.9.8 (Fibonacci with iteration) Fibonacci numbers (§1.6) can be computed by iteration on pairs. Find a function f such that $Fn := \pi_1(f^n(0,1))$ satisfies the specifying equations for the Fibonacci function:

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

Hint: If you formulate the step function with π_1 and π_2, the third specifying equation should follow by computational equality, otherwise discrimination on a subterm obtained with iter may be needed.

1.10 Ackermann Function

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

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

The equations cannot serve as a defining equations since the recursion is not structural. 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** helper function:[2]

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

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

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

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

Note that the three specifying equations hold by computational equality (i.e., both sides of the equations reduce to the same term). Thus verifying the equations with a proof assistant is trivial.

We remark that 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.

Exercise 1.10.1 (Truncating subtraction without cascaded discrimination)
Define a truncating subtraction function that discriminates on the first argument and delegates discrimination on the second argument to a helper function. Prove that your function agrees with the standard subtraction function `sub` from §1.2. Arrange your definitions such that your function satisfies the defining equations of `sub` by computational equality.

Exercise 1.10.2 (Ackermann with iteration)
There is an elegant iterative definition of the Ackermann function

$$An := B^n\,\mathsf{S}$$

using a higher-order helper function B defined with iteration. Define B and verify that A satisfies the specifying equations for the Ackermann function by computational equality. Consult Wikipedia to learn more about the Ackermann function.

1.11 Unfolding Functions

Procedural specifications can be faithfully represented as non-recursive inductive functions taking a **continuation function** as first argument. We speak of **unfolding functions**. Figure 1.5 shows the unfolding functions for the procedural specifications of the Fibonacci and Ackermann functions we have discussed in §1.6 and §1.10.

An unfolding function is a higher-order function specifying a recursive function without recursion. It does so by abstracting out the recursion by means of a continuation function taken as argument.

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

Figure 1.5: Unfolding functions for the Fibonacci and Ackermann functions

Intuitively, it is clear that a function f satisfies the specifying equations for the Fibonacci function if and only if it satisfies the **unfolding equation**

$$f n = \mathsf{Fib}\, f\, n$$

for the unfolding function Fib. Formally, this follows from the fact that the specifying equations for the Fibonacci function are computationally equal to the respective instances of the unfolding equation:

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

The same is true for the Ackermann function.

Exercise 1.11.1 Verify with the proof assistant that the realizations of the Fibonacci function defined in §1.6 and Exercise 1.9.8 satisfy the unfolding equation for the specifying unfolding function.

Exercise 1.11.2 Verify with the proof assistant that the realizations of the Ackermann function defined in §1.10 satisfies the unfolding equation for the specifying unfolding function.

Exercise 1.11.3 Give unfolding functions for addition and truncating subtraction and show that the unfolding equations are satisfied by the inductive functions we defined for addition and subtraction.

Exercise 1.11.4 The unfolding function Fib is defined with a nested pattern $\mathsf{SS}n$ in the third defining equation. Show how the nested pattern can be removed by formulating the third equation with a helper function.

Exercise 1.11.5 (Iterative definition of a Fibonacci function) There is a different definition of a Fibonacci function using the helper 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 the unfolding equation for Fib.

c) Compare the iterative computation of Fibonacci numbers considered here with the computation using iter in Exercise 1.9.8.

1.12 Proof Rules as Propositions

So far we have described the proof rules for discrimination (constructor-based case analysis) and induction informally. It is however essential to have formal descriptions of proof rules nailing down all technical details. In fact, proof rules can be described formally and elegantly as propositions. We start with a proposition describing the proof rule for boolean discrimination:

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

The rule works for every predicate p on the booleans. Given proofs for $p(\mathbf{T})$ and $p(\mathbf{F})$, it yields a proof of $\forall x^{\mathsf{B}}.\ p(x)$. Predicates on booleans (statements about booleans) are modeled as functions $p : \mathsf{B} \to \mathbb{P}$, where $\mathbb{P}$ is the type of all propositions. This makes sense given that $p(\mathbf{T})$, $p(\mathbf{F})$, and $p(x)$ are supposed to be propositions.

We may see the proposition for the boolean discrimination rule as a function type describing functions that given a predicate $p^{\mathsf{B} \to \mathbb{P}}$ and proofs of $p(\mathbf{T})$ and $p(\mathbf{F})$ yield a proof of $\forall x^{\mathsf{B}}.\ p(x)$. The interpretation of propositions as types is in fact a main principle of type theory.

We are now prepared to state the proof rule for induction on numbers as a proposition:

$$\forall p^{\mathsf{N} \to \mathbb{P}}.\ \ p(0) \to (\forall n^{\mathsf{N}}.\ p(n) \to p(\mathsf{S}n)) \to \forall n^{\mathsf{N}}.\ p(n)$$

The rule says that we can obtain a proof of $\forall n^{\mathsf{N}}.\ p(n)$ by providing proofs for the *zero case* $p(0)$ and the *successor case* $\forall n^{\mathsf{N}}.\ p(n) \to p(\mathsf{S}n)$. A proof of the successor case may be seen as a function that given a number n and a proof of $p(n)$ yields a proof of $p(\mathsf{S}n)$. In other words, a proof of the successor case is a method that for every n upgrades a proof of $p(n)$ to a proof of $p(\mathsf{S}n)$. By iterating this method n-times on a proof of $p(0)$

$$p(0),\ p(1),\ p(2),\ldots,p(n)$$

we can obviously get a proof of $p(n)$ for every n. Note the appearance of the inductive hypothesis $p(n)$ as assumption in the successor case.

We remark that inductive proofs are obtained *backwards* in practice: One first announces that the claim is shown by induction and then works on the proof obligations for the zero and the successor case.

By now we have seen several function types $\forall x^X.\,t$ starting with a quantification specifying an argument x of type X. The variable x is used to describe a dependency of the target type t on the argument. In general, we speak of a type $\forall x^X.t$ as a *dependent function types*. The special case $\forall x^{\mathbb{T}}.\,t$ where X is the type $\mathbb{T}$ of all types is known as a *polymorphic function type*.

1.13 Concluding Remarks

The 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 and polymorphic functions are natural features of this equational language. The power of the language comes from the fact that functions and types can serve as arguments and results of functions.

We have seen how booleans, numbers, and pairs can be accommodated as inductive types using constructors, and how inductive functions discriminating on inductives types can be defined using equations. Functional recursion is restricted to structural recursion so that termination of computation is ensured.

As usual, we use the word function with two meanings. Usually, when we talk about a function, we refer to its concrete definition in type theory. This way, we can distinguish between inductive and plain functions, or recursive and non-recursive functions. Sometimes, however, we refer to a function as an abstract object that relates inputs to outputs but hides how this is done. The abstract view makes it possible to speak of a uniquely determined Fibonacci function or of a uniquely determined Ackermann function.

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

- Inductive type definitions, type and value constructors
- Inductive functions, plain functions

- Booleans, numbers, and pairs obtained with inductive types
- Defining equations, patterns, computation rules
- Disjoint, exhaustive, termining systems of equations
- Cascaded function types, partial applications
- Polymorphic function types, implicit arguments
- Structural recursion, structural case analysis, discrimination
- Structural induction, (quantified) inductive hypotheses
- Proof digrams, proof goals, subgoals, proof actions (tactics)
- Simplification steps, rewriting steps , computational equality
- Truncated subtraction, Fibonacci function, Ackermann function
- Iteration
- Procedural specifications, specifying equations
- Unfolding functions, unfolding equations, continuation functions

2 Basic Computational Type Theory

This chapter introduces 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 then 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. That computational type theory can express logical propositions and proofs will be shown in the next chapter. In Chapter 4 we will boost the expressivity of computational type theory by enhancing type checking so that it operates modulo computational equality of types. The resulting type theory covers equational and 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 a family of pair types:

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

Each of the definitions introduces a system of typed constants consisting of a **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 classifying 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 terms $\mathsf{pair}\,stuv$ where u has type s and v has type t.

A distinguishing feature of computational type theory are **dependent function types**

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

which we often write as $\forall x^s.\ t$. An example for a dependent function type is the type of the value constructor pair

$$\forall X^{\mathbb{T}}.\forall Y^{\mathbb{T}}.\ X \to Y \to \mathsf{Pair}\ X\ Y$$

which uses the primitive for dependent function types twice. This way 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. In other words, $s \to t$ is notation for $\forall x : s.t$, provided 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, the names for **bound variables** do not matter. For instance, the terms $\forall X.X \to X$ and $\forall Y.Y \to Y$ are identified.

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 $\mathbb{T} : \mathbb{T}$ (i.e., the type of $\mathbb{T}$ is $\mathbb{T}$). Later, we will remove the cycle and work with an infinite hierarchy of type universes $\mathbb{T}_1 \subset \mathbb{T}_2 \subset \cdots$.

A key feature of computational type theory is the fact that types and functions are values like all other values. We say that types and functions are first-class objects. Note that Pair and pair are functions taking types as arguments.

Type theory only admits **well-typed terms**. Examples for ill-typed terms are $\mathsf{S}\,\mathbb{T}$ and $\mathsf{pair}\,0\,\mathsf{N}$. In the basic type theory we are considering here, 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 more inductive arguments called **discriminating arguments**. We shall look at the examples appearing in Figure 2.2. Each of the three definitions first declares the name

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

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

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

Figure 2.1: Inductive function definitions

(a constant) and the type of the defined function. Then **defining equations** are given realizing a **disjoint** and **exhaustive** case analysis. 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 left hand sides of defining equations are called **patterns**. The variables occurring in a pattern are local to the equation and can be used in the right hand side of the equation. We say that a pattern **binds** the variables occurring in it. An important requirement for patterns is **linearity**, that is, none of the variables bound by a pattern can occur more that once in the pattern. Also for this reason the parametric arguments of the constructor pair in the pattern for swap are omitted. 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 conditions apply recursively.

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 works 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.

If an inductive function recurses, the recursion must be on the first discriminating argument and the variables introduced by the pattern for this argument. In the examples in Figure 2.2, only the variable x in the defining equations for add and sub qualify for recursion. We refer to this severely restricted form of recursion as **structural recursion**.

2.3 Reduction

The defining equations of an inductive function serve as **reduction rules** that rewrite applications of the defined function. For instance, the application sub (Ss) (St) can be **reduced** to sub s t using the third defining equation of sub. Things are arranged such that at most one defining equation applies to an application (disjointness), and such that every application where all discriminating arguments start with a constructor can be reduced (exhaustiveness). Thus a **closed term** (no free variables) can be reduced as long as it contains an application of a defined function. We refer to the process of applying reductions rules as **reduction**, and we see reduction as computation. We refer to reduction rules also as **computation rules**.

Things are arranged such that reduction **always terminates**. Without a restriction on recursion, non-terminating inductive functions are possible. The structural recursion requirement is a sufficient condition for termination that can be checked algorithmically.

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 and refer to irreducible terms as **normal terms**. Terms that are closed and normal are also called **canonical terms**.

We can now formulate **four key properties of computational type theory**:

- **Termination** Reduction always terminates.
- **Unique normal forms** Terms reduce to at most one normal form.
- **Type preservation** Reduction preserves types: If a term of type t is reduced, the obtained term is again of type t.
- **Canonicity** Closed normal terms of an inductive type start with a value constructor of the type.

Canonicity gives an important integrity guarantee for inductive types saying that the elements of an inductive type do not change when inductive functions returning values of the type are added. Canonicity ensures that the canonical terms of an inductive type are exactly the terms that one can build with the value constructors of the type.

The definition format for inductive functions is carefully designed such that the key properties are preserved when a definition is added. Exhaustiveness of the defining equations is needed for canonicity, disjointness of the defining equations is needed for uniqueness, and the structural recursion requirement ensures termination. Moreover, the type checking conditions for equations are needed for type preservation.

Exercise 2.3.1 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.

2.4 Plain Definitions

Besides inductive function definitions, there are **plain definitions** with a single defining equation

$$cx_1 \dots x_n \;:=\; s$$

where the pattern $cx_1 \dots x_n$ must not contain a constructor. Only the variables $x_1, \dots, x_n$ may appear in s. We speak of a **plain constant definition** if $n = 0$ and a **plain function definition** if $n > 0$. Similar to inductive function definitions, plain definitions must declare the type of the defined constant c.

The reduction rule for plain definitions is known as **delta reduction** (δ-reduction). It takes the form

$$cx_1 \dots x_n \;\succ\; s$$

where s is the term appearing as the right hand side of the definition of c.

Plain definitions must not be recursive. This ensures that the key properties of computational type theory are preserved when plain definitions are added.

Exercise 2.4.1 Recall the definition of iter (§ 1.9). Explain the difference between the following plain definitions:

$$A \;:=\; \mathsf{iter}\,\mathsf{S}$$
$$Bxy \;:=\; \mathsf{iter}\,\mathsf{S}\,x\,y$$

Note that the terms Axy and Bxy both reduce to the normal term $\mathsf{iter}\,\mathsf{S}\,x\,y$. Moreover, note that the terms A and Ax are reducible, while the terms B and Bx are not reducible.

Exercise 2.4.2 Type checking is crucial for termination of δ-reduction. Consider the ill-typed plain function definition $cx := xx$ and the ill-typed term cc, which by δ-reduction reduces to itself: $cc \succ cc$. Convince yourself that there cannot be type for c such that the self application cc type-checks.

2.5 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 give 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 $\forall X^{\mathbb{T}}.X \to X$, which 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 term 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 (assuming that it is determined by the context). We also omit parentheses and lambdas relying on two basic notational rules:

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

To specify the type of 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 2.5.1 Type checking is crucial for termination of β-reduction. Convince yourself that β-reduction of the 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.

2.6 Typing Rules

Type checking is an algorithm that determines whether a term or a defining equation or an entire definition 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 you may 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 term of an application.

Note that the rules admit any type $u : \mathbb{T}$ as argument type of a dependent function type $\forall x : u.v$. So far we have only seen examples of dependent function types where u is $\mathbb{T}$. Dependent function types $\forall x : u.v$ where u is not the universe $\mathbb{T}$ will turn out to be important.

Recall that simple function types $u \to v$ are dependent function types $\forall x : u.v$ where the argument variable x does not occur in the result type v. If we specialize the typing rules to simple function types, we obtain rules that will look familiar to functional programmers:

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

2.7 Let Expressions

We will also use **let expressions**

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

providing for local definitions. The reduction rule for let expressions

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

is called **zeta rule** (ζ-rule).

Let expressions can usually be expressed as β-redexes. There will be a feature of computational type theory (the conversion rule in §4.1) that distinguishes let expressions from β-redexes in that let expressions introduce local reduction rules.

Exercise 2.7.1 Express LET $x^t = s$ IN u with a β-redex. Reduction of the β-redex should give the same term as reduction of the let expression.

2.8 Matches

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

$$\text{MATCH } s \; [\, 0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v \,]$$

and come with two reduction rules:

$$\begin{aligned} \text{MATCH } 0 \; [\, 0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v \,] &\succ u \\ \text{MATCH } \mathsf{S}s \; [\, 0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v \,] &\succ (\lambda x.v)s \end{aligned}$$

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

Matches can be expressed as applications of certain inductive functions, and this translation will be our preferred view on matches. In other words, we will see matches as a derived notation. For matches on numbers, we may define the function

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

and replace matches with applications of this function:

$$\text{MATCH } s \; [\, 0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v \,] \quad \rightsquigarrow \quad \mathsf{M}\, \_\, s\, u\, (\lambda x.v)$$

We say that M is the simply typed **match function** for N.

Since matches are notation for applications 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 derived typing rule for matches:

> A term 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.

We may write boolean matches with the familiar **if-then-else notation**:

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

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 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 2.8.1 (Boolean negation) Consider the inductive type definition

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

for booleans and the plain definition

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

of 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 2.8.2 (Swap function for pairs)

a) Define a function swap swapping the components of a pair using a plain definition, lambda abstractions, and a match.

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

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

2.9 Recursive Abstractions

Recursive abstractions take the form

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

and represent recursive functions as unfolding functions. There are two local variables f and x, where f acts as continuation function and x as argument. The type of u must be t, and the type of the recursive abstraction itself is $s \to t$.

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

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

The reduction rule for recursive abstractions looks as follows:

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

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 an inductive function definition, 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 2.9.1 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.

2.10 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, patterns of clauses in 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**

$$(\lambda x.sx) \approx_\eta s \qquad \text{if } x \text{ does not occur free in } s$$

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 constructor S 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. A **simplification** is a conversion where the final term is obtained from the initial term by reduction. Examples for conversions that are not simplifications are applications of the η-equivalence law, or **expansions**, which are reductions in reverse order (e.g., proceeding from x to $0 + x$). Figure 5.2 in Chapter 5 contains several proof diagrams with expansion steps.

A complex operation the reduction rules build on is **substitution** s^x_t. 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**.

Exercise 2.10.1 (Currying) Assume types X, Y, Z and define functions

$$C : (X \times Y \to Z) \to (X \to Y \to Z)$$
$$U : (X \to Y \to Z) \to (X \times Y \to Z)$$

such that the equations $C(Uf) = f$ and $U(Cg)(x, y) = g(x, y)$ hold by computational equality. Find out where η-equivalence is used.

2.11 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, where we formalize 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 constant, or a constant applied to canonical terms, or an abstraction (obtained with λ or FIX), or a function type (obtained with $\forall$), or a universe (so far we have $\mathbb{T}$). A **constant** is either a constructor or a defined constant.

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

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

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

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

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.

2.12 Choices Made by Coq

Coq provides neither inductive function definitions nor plain function definitions. Inductive functions can be described with plain constant definitions, recursive abstractions, and matches. Plain functions can be described with plain constant definitions and lambda abstractions. There is syntactic sugar facilitating the translation of function definitions (inductive or plain) into Coq's kernel language. We have seen a translation of an inductive function D (§ 2.9) into Coq's kernel language.

Not having function definitions makes reduction more fine-grained and introduces intermediate normal forms users don't want to see. To mitigate the problem, 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 work with inductive function definitions and not use recursive abstractions at all. The accompanying demo files show how our 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) when inductive types where added to a language designed without having inductive types in mind (around 1985). Agda is a modern implementation of computational type theory that comes with inductive function definitions and does not offer matches and recursive abstractions.

2.13 Executive Summary

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

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 with a conversion rule such that type checking operates modulo computational equality of types. The conversion rule is needed so that proof rules such as rewriting and induction on numbers can be obtained within computational type theory.

The computational type theory we are considering is based on typed function application and definitions (plain or inductive). Lambda abstractions, let expressions, and matches are often convenient but are not essential in that they can always be translated away. Recursive abstractions are mentioned here because they are needed in Coq to compensate the lack of inductive functions.

Last but not least we mention that every function expressible 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. The evaluation algorithm 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.

2.14 Notes

Our presentation of computational type theory is informal. We took some motivation from the previous chapter but it may take time until you fully understand what is said in the current chapter. Previous familiarity with functional programming will help. 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 should prove useful.

Formalizing the system presented in this chapter and proving the claimed properties is a substantial 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.

A comprehensive discussion of the historical development of computational type theories can be found in Constable's survey paper [8]. We recommend the book on homotopy type theory [26] for a complementary presentation of computational type theory. The reader may also be interested in learning more about *lambda calculus* [4, 15], a minimal computational system arranged around lambda abstractions and beta reduction.

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 the representing 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 basic 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 $s \land t$ and disjunctions $s \lor t$ will be obtained with inductive type constructors such that a proof of $s \land t$ consists of a proof of s and a proof of t, and a proof of $s \lor t$ is either a proof of s or a proof of t. The proposition falsity having no proof will be expressed as an empty inductive type $\bot$. With falsity we will 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 extended the type theory with 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 $\mathbb{P}$ contains all function types $\forall x^u.v$ where v 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 both a proposition and a type. 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 $\lambda X^{\mathbb{P}}\, x^X.x$, and

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

is a proposition that has the proof

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

Interestingly,

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

is a proposition that has no proof (see Exercise 3.2.1).

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 ending with $\mathbb{P}$.

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

Exercise 3.1.2 Give proofs of the following propositions:

a) $\forall XY^{\mathbb{P}}. (X \to Y) \to ((X \to Y) \to X) \to Y$

b) $\forall XY^{\mathbb{P}}. (X \to X \to Y) \to ((X \to Y) \to X) \to Y$

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.[2] We define an inductive function

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

discriminating on its second argument of type $\bot$. Since $\bot$ has no value constructor, we need no defining equation for $\mathsf{E}_\bot$. The function $\mathsf{E}_\bot$ realizes an important logical principle known as **explosion rule** or **ex falso quodlibet**: Given a hypothetical proof of falsity, we can get a proof of everything. More generally, given a hypothetical proof of falsity, $\mathsf{E}_\bot$ gives us an element of every type. Following common language we explain later, we call $\mathsf{E}_\bot$ the **eliminator** for $\bot$.

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

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

[2]Suppose there is a closed term of type $\bot$. Because of termination and type preservation there is a closed and normal term of type $\bot$. By canonicity this term must be obtained with a constructor for $\bot$. Contradiction.

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

Variable X ranges over propositions.

Figure 3.1: Proofs of propositions involving negations

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 based on unproven 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 eliminator $\mathsf{E}_\bot$ makes use of the special properties of falsity. Note the use of the let expression in the proof in the last line. 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.

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 speak of **consistency** if a type theory can express empty types. Consistency is needed for a type theory so that it can express negative propositions.

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

Most people are 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. We make this design explicit with two inductive type definitions:

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

The definitions introduce the following constructors:

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

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.

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

$$\mathsf{M}_\vee : \forall XYZ^{\mathbb{P}}.\; X \vee Y \to (X \to Z) \to (Y \to Z) \to Z$$
$$\mathsf{M}_\vee\, XYZ\,(\mathsf{L}\,x)\,e_1 e_2 \;:=\; e_1 x$$
$$\mathsf{M}_\vee\, XYZ\,(\mathsf{R}\,y)\,e_1 e_2 \;:=\; e_2 y$$

$$\text{MATCH } s\;[\,\mathsf{C}\,xy \Rightarrow t\,] \;\rightsquigarrow\; \mathsf{M}_\wedge\,\_\,\_\,\_\,s\,(\lambda xy.t)$$
$$\text{MATCH } s\;[\,\mathsf{L}\,x \Rightarrow t_1 \mid \mathsf{R}\,y \Rightarrow t_2\,] \;\rightsquigarrow\; \mathsf{M}_\vee\,\_\,\_\,\_\,s\,(\lambda x.t_1)\,(\lambda y.t_2)$$

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}
$X \wedge Y \to X$	$\lambda a.\,\text{MATCH } a\;[\,\mathsf{C}xy \Rightarrow x\,]$
$X \wedge Y \to Y$	$\lambda a.\,\text{MATCH } a\;[\,\mathsf{C}xy \Rightarrow y\,]$
$X \wedge Y \to Y \wedge X$	$\lambda a.\,\text{MATCH } a\;[\,\mathsf{C}xy \Rightarrow \mathsf{C}yx\,]$
$X \vee Y \to Y \vee X$	$\lambda a.\,\text{MATCH } a\;[\,\mathsf{L}x \Rightarrow \mathsf{R}_{YX}\,x \mid \mathsf{R}y \Rightarrow \mathsf{L}_{YX}\,y\,]$

The variables X, Y, Z range over propositions.

Figure 3.3: Proofs for propositions involving conjunctions and disjunctions

We note that $\mathsf{E}_\perp$ (§ 3.2) is the match function for the inductive type $\perp$. We define the notation

$$\text{MATCH } s\;[\,] \;\rightsquigarrow\; \mathsf{E}_\perp\,\_\,s$$

Figure 3.3 shows proofs of propositions involving conjunctions and disjunctions. The propositions formulate familiar logical laws. Note that we supply as subscripts the implicit arguments of the proof constructors C, L, and R when we think it is helpful.

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

$$\text{MATCH } a\;[\,\mathsf{C}(\mathsf{C}xy)z \Rightarrow \mathsf{C}x(\mathsf{C}yz)\,]$$

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

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

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

Figure 3.4: Proofs with nested patterns

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

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

nesting a second plain match.

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 provided the application does not appear as part of a pattern.

3.4 Propositional Equivalence

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

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

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. The equivalences formulate well-known properties of conjunction and disjunction.

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

Figure 3.5: Equivalence laws for conjunctions and disjunctions

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 $\neg(X \wedge Y) \longleftrightarrow \neg X \vee \neg Y$ since it requires the law of excluded middle (§ 3.8). 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

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

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

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

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

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

3.6 Impredicative Characterizations

Quantification over propositions has amazing expressivity. Given two propositional variables X and Y, we can prove the equivalences

$$\begin{aligned} \bot \quad &\longleftrightarrow \quad \forall Z^{\mathbb{P}}.\ Z \\ X \wedge Y \quad &\longleftrightarrow \quad \forall Z^{\mathbb{P}}.\ (X \rightarrow Y \rightarrow Z) \rightarrow Z \\ X \vee Y \quad &\longleftrightarrow \quad \forall Z^{\mathbb{P}}.\ (X \rightarrow Z) \rightarrow (Y \rightarrow Z) \rightarrow Z \end{aligned}$$

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, and disjunction. Figure 3.6 gives proof terms for the equivalences. One speaks of an **impredicative proposition** if the proposition contains a quantification over all propositions.

Note that the impredicative characterizations are related to the types of the match functions for $\bot$, $X \wedge Y$, and $X \vee Y$.

Exercise 3.6.1 Find an impredicative characterization for $\top$.

Exercise 3.6.2 (Exclusive disjunction)
Consider exclusive disjunction $X \oplus Y \longleftrightarrow (X \wedge \neg Y) \vee (\neg X \wedge Y)$.

a) Define exclusive disjunction with an inductive type definition. Use two proof constructors and prove the specifying equivalence.

b) Find and verify an impredicative characterization of exclusive disjunction.

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

$$X \wedge Y \;\longleftrightarrow\; \forall Z^{\mathbb{P}}.\, (X \to Y \to Z) \to Z$$
$$\mathsf{C}\ (\lambda a Z f.\, \textsc{match}\ a\ [\, \mathsf{C} x y \Rightarrow f x y\,])\ (\lambda f.\, f(X \wedge Y)\mathsf{C}_{XY})$$

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

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

Figure 3.6: Impredicative characterizations with proof terms

3.7 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 X
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

$$\lambda a^{X \longleftrightarrow \neg X}.\ \textsc{match}\ a\ [\ \mathsf{C} f g \Rightarrow \textsc{let}\ x = g(\lambda x. f x x)\ \textsc{in}\ f x x\ ]$$

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 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:

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

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

Figure 3.7 shows a proof diagram for a double negation law for universal quantification. Since universal quantifications are function types like implications, no new proof actions are needed.

Figure 3.8 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 the necessary matches could be inserted by the intro action.

Figure 3.9 gives a proof diagram for a distributivity law involving 6 subgoals. Note the symmetry in the proof digram and the proof term constructed.

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

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

Figure 3.7: Proof diagram for a double negation law for universal quantification

	$\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}} \forall q^{X \to \mathbb{P}}.$ $(\forall x.\, px \longleftrightarrow qx) \to (\forall x.\, qx) \to \forall x.\, px$	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 3.8: Proof diagram using a destructuring action

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

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

Figure 3.9: Proof diagram for a distributivity law

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

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

Figure 3.10: Proof diagram for a double negation law for implication

Figure 3.10 gives a proof diagram for a double negation law for implication. Note the use of the **exfalso action** applying the explosion rule as realized by $\mathsf{E}_\bot$.

Exercise 3.7.1 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) $\neg\neg(X \to Y) \;\longleftrightarrow\; \neg(X \wedge \neg Y)$

Exercise 3.7.2 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 3.7.3 Find out why one direction of the equivalence
$\forall X^{\mathbb{T}} \forall Z^{\mathbb{P}}.\ (\forall x^X. Z) \longleftrightarrow Z$ cannot be proved.

Exercise 3.7.4 Prove $\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}} \forall Z^{\mathbb{P}}.\ (\forall x.px) \to Z \to \forall x.\ px \wedge Z.$

3.8 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 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, except in one case. Exercise 12.5.5 will tell you more.

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 terms of propositional types. Function types provide for implications and universal quantifications. Falsity, conjunctions and disjunctions can be added using inductive type definitions. Universal quantification as obtained with the propositions as types approach is general in that it can quantify over all types including function types and universes. Since proofs are accommodated as first-class objects, one can even quantify over proofs. In the following chapters we will see that the type-theoretic approach to logic scales 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 reuse reduces proof checking to type checking and much simplifies the implementation of proof assistants.

In the propositions as types approach, proofs are obtained as terms built with lambda abstractions, applications, and matches. The resulting proof language is amazingly elegant and compact. The primitives of the language generalize familiar proof patterns: making assumptions, applying implicational assumptions, and destructuring of assumptions.

This chapter is the place where the reader will 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 use proof diagrams recording incremental constructions of proof terms. When we construct proof terms in interaction with a proof assistant, we issue proof actions that incrementally build the proof term and display the information recorded in the proof diagram.

In the system presented so far, proofs are verified with the typing rules and no use is made of the reduction rules. This will change in the next chapter where we extend the typing discipline with a conversion rule identifying computationally equal types.

We remark that all constructions shown in this chapter carry through if the universe $\mathbb{P}$ is replaced with $\mathbb{T}$. In fact, the basic intuitions for the propositions as types approach don't require the presence of a universe of propositions. The reasons for having $\mathbb{P}$ in addition to $\mathbb{T}$ will become clearer in the next chapter.

The details of the typing rules matter. What prevents a proof of falsity are the typing rules and the rules restricting the form of inductive definitions. In this text, we explain the details of the typing rules mostly informally, exploiting that compliance with the typing rules is verified automatically 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 deepen your understanding of the typing rules using the proof assistant.

4 Conversion Rule, Universe Hierarchy, and Elimination Restriction

This chapter introduces a new typing rule called conversion rule, generalizing typing so that it operates modulo computational equality of types. The conversion rule is needed for types that quantify over predicates, which will appear with equality, existential quantification, and induction. The conversion rule also provides for defined predicates, which are essential for more elaborate proofs.

We also discuss two necessary restrictions of the typing discipline, which apply to more agressive uses of types. Both restrictions are needed to maintain consistency of the type theory (i.e., the presence of empty types). First, we have to withdraw the naive 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 $\mathbb{T}_1 \subset \mathbb{T}_2 \subset \mathbb{T}_3 \subset \cdots$. Second, we introduce the so-called elimination restriction, restricting discrimination on non-computational types to propositional result types.

4.1 Conversion Rule

Recall the typing rules for applications and lambda abstractions from §2.6.

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

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

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

The rule says that a term s has type u if s has type u' and u and u' are computationally equal (§2.10). We use the notation $u \approx u'$ to say that two terms u and u' are computationally equal. Note that the conversion rule has a premise $\vdash u : \mathbb{T}$, which ensures that the term u describes a type.

Adding the conversion rule preserves the key properties of computational type theory (§2.3). As before, there is an algorithm that given a term decides whether

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

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

Figure 4.1: Proof diagram for Leibniz symmetry

the term 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., $\mathbb{P}\subset\mathbb{T}$).

We explain the conversion rule with two applications.

Negation and propositional equivalence as defined functions

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

$$\neg:\ \mathbb{P}\to\mathbb{P} \qquad\qquad \longleftrightarrow:\ \mathbb{P}\to\mathbb{P}\to\mathbb{P}$$

$$\neg X\ :=\ X\to\bot \qquad\qquad X\longleftrightarrow Y\ :=\ (X\to Y)\wedge(Y\to X)$$

The plain definitions provide us with the constants $\neg$ and $\longleftrightarrow$ for functions constructing negations and propositional equivalences. Delta reduction replaces applications $\neg s$ with propositions $s\to\bot$, and applications $s\longleftrightarrow t$ with propositions $(s\to t)\wedge(t\to s)$. The conversion rules ensures that proofs of $s\to\bot$ are proofs of $\neg s$ (and vice versa), and that proofs of $(s\to t)\wedge(t\to s)$ are proofs of $s\longleftrightarrow t$ (and vice versa).

Leibniz symmetry

The conversion rule yields proofs for propositions that are not provable without the conversion rule. 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

$$py\to px\ \approx\ (\lambda z.\ pz\to px)\,y$$
$$(\lambda z.\ pz\to px)\,x\ \approx\ px\to px$$

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 better 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

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

Note that third rule establishes subtyping of function types so that, for instance, we obtain the inclusion $(u \to \mathbb{P}) \subset (u \to \mathbb{T})$ 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 §32.3). What works, however, is an infinite cumulative hierarchy of universes

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

realized with the following typing rules:

$$\frac{}{\vdash \mathbb{P} : \mathbb{T}_2} \qquad \frac{}{\vdash \mathbb{T}_i : \mathbb{T}_{i+1}} \qquad \frac{}{\vdash \mathbb{P} \subset \mathbb{T}_i} \qquad \frac{i < j}{\vdash \mathbb{T}_i \subset \mathbb{T}_j}$$

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

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

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 universes $\mathbb{T}_i$. This is justified by the observation 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 values 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 type $cs_1 \dots s_n$ with $n \geq 0$ is **computational** if its type constructor c is computational. A type constructor c is **computational** if in case it targets $\mathbb{P}$ it has at most one proof constructor d and all nonparametric arguments of d have propositional types. Examples for computational propositions we have already seen are $\bot$, $\top$, and conjunctions $s \wedge t$. Examples for noncomputational propositions we have already seen are disjunctions $s \vee t$. Note that by our definition every nonpropositional inductive type is computational.

The **elimination restriction** applies to inductive function definitions and requires that the result type of the defined function must be propositional if there is a discriminating argument whose type is a noncomputational 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 with $\mathbb{T}$ 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 speak of a **computational falsity elimination** when we use $\mathsf{E}_\bot$ with a nonpropositional type. It turns out that computational falsity elimination is essential for defining certain functions. Examples appear in §10.6, §10.7, and §17.1.

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 all proofs of a proposition are equal 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 §32.5).

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

$$\text{MATCH } (\mathsf{L\,I}) \; [\, \mathsf{L}\,\_ \Rightarrow \bot \mid \mathsf{R}\,\_ \Rightarrow \top \,]$$

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

$$\text{MATCH } (\mathsf{R\,I}) \; [\, \mathsf{L}\,\_ \Rightarrow \bot \mid \mathsf{R}\,\_ \Rightarrow \top \,]$$

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 $\mathbb{P}$, 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 32).

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 $\perp$. The other essential computational propositions are types involving higher-order recursion (Chapters 18 and 31), and inductive equality types providing for casts (Chapter 29).

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 $s = t$ 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 of trivial equations $s = s$, and one constant providing for rewriting. To prove with the constants, it suffices to know their types, their actual definitions are not needed. We will speak of declared constants.

The conversion rule of the type theory gives the constants for trivial equations and for rewriting the necessary proof power. In particular, the conversion rule has the effect that propositional equality subsumes computational equality. Moreover, the conversion rule and quantification over predicates ensure that all equational rewriting situations can be captured with a single rewriting constant.

There is much elegance and surprise in this chapter. Much of the technical essence of computational type theory is exercised with propositional equality. Take your time to understand this beautiful construction in depth.

5.1 Abstract Propositional Equality

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:

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

For now we keep the constants abstract. It turns out that we can do equational reasoning without knowing the definitions of the constants. All we need are the constants and their types.

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

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

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. This is a remarkable fact.

The constant R provides for equational rewriting: Given a proof of an equation $s = t$, we can place a claim pt with the claim ps using R. Moreover, we can get from an assumption ps an additional assumption pt by asserting pt and proving pt with R and 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 proof term for the equation $!\mathbf{T} = \mathbf{F}$. Explain why the term is also a proof term for the equation $\mathbf{F} = !!\mathbf{F}$.

Exercise 5.1.2 Give a proof term for the **converse rewriting law**
$\forall X^{\mathbb{T}} \forall xy \forall p^{X \to \mathbb{P}}.\ \mathsf{eq}\, Xxy \to py \to px$.

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.

Exercise 5.1.4 Give a term applying R to 7 arguments (including implicit arguments). In fact, for every number n there is a term that applies R to exactly n arguments.

5.2 Basic Equational Facts

The constants Q and R give us straightforward proofs for many equational facts. To say it once more, Q together with the conversion rule provides proofs by computational equality, and R together with the conversion rule provides equational rewriting. Figure 5.1 shows a collection of basic equational facts, and Figure 5.2

$\top \neq \bot$	propositional disjointness
$\mathbf{T} \neq \mathbf{F}$	constructor disjointness for B
$\forall x^{\mathsf{N}}.\ 0 \neq \mathsf{S}x$	constructor disjointness for N
$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ \mathsf{S}x = \mathsf{S}y \to x = y$	injectivity of successor
$\forall X^{\mathbb{T}} Y^{\mathbb{T}} f^{X \to Y} x\, y.\ x = y \to fx = fy$	applicative closure (feq)
$\forall X^{\mathbb{T}} x^X y^X.\ x = y \to y = x$	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

gives proof diagrams and the resulting proof terms for most of them. The remaining proofs are left as exercise. It is important that you understand each of the proofs in detail.

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.

If we look at the facts in Figure 5.2, we see that three of them

$\mathbf{T} \neq \mathbf{F}$	constructor disjointness for B
$\forall x^{\mathsf{N}}.\ 0 \neq \mathsf{S}x$	constructor disjointness for N
$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ \mathsf{S}x = \mathsf{S}y \to x = y$	injectivity of successor

concern inductive types while the others are not specifically concerned with inductive types. We speak of **constructor laws** for inductive types. Note that the proofs of the constructor laws all involve a match on the underlying inductive type, and recall that matches are obtained as inductive functions. So to prove a constructor law, one needs to discriminate on the underlying inductive type at some point.

Interestingly, the proof of the transitivity law

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

can be simplified so that the conversion rule is not used. The simplified proof term

$$\lambda X x y z e_1 e_2.\ \mathsf{R}_{(\mathsf{eq}\,x)}\, e_2\, e_1$$

exploits the fact that the equation $x = z$ is the application $(\mathsf{eq}\,x)z$ up to notation.

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

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

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

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

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

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

$X : \mathbb{T},\ x : X,\ y : X,\ z : X,$	$x = y \to y = z \to x = z$	intro
$e : x = y$	$y = z \to x = z$	conversion
	$(\lambda y.\ y = z \to x = z)\,y$	apply $\mathsf{R}\,\_\,e$
	$(\lambda y.\ y = z \to x = z)\,x$	conversion
	$x = z \to x = z$	$\lambda e.e$

Proof term: $\lambda Xxyze.\, \mathsf{R}_{(\lambda y.\ y=z\to x=z)}\, e\, (\lambda e.e)$

Figure 5.2: Proofs of basic equational facts

Exercise 5.2.1 Study the two proof terms given for transitivity in detail using Coq. Give the proof diagram for the simplified proof term. Convince yourself that there is no proof term for symmetry that can be type-checked without the conversion rule.

Exercise 5.2.2 Give proof diagrams and proof terms for the following propositions:

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

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

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

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

Exercise 5.2.3 (Constructor law for pairs)
Prove that the pair constructor is injective: $\mathsf{pair}\, x\, y = \mathsf{pair}\, x'\, y' \to x = x' \wedge y = y'$.

Exercise 5.2.4 (Leibniz characterization of equality)
Verify the following characterization of equality:

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

The equivalence is known as *Leibniz characterization* or as *impredicative characterization* of equality. Also verify the *symmetric Leibniz characterization*

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

which may be phrased as saying that two objects are equal if and only if they satisfy the same properties.

Exercise 5.2.5 (Disequality) From the Leibniz characterization of equality it follows that $x \neq y$ if there is a predicate that holds for x but does not hold for y. Prove the proposition $\forall X^{\mathbb{T}}\, \forall x y^X\, \forall p^{X \to \mathbb{P}}.\ px \to \neg py \to x \neq y$ expressing this insight.

5.3 Definition of Leibniz Equality

Here are plain function definitions defining the constants for abstract propositional equality:

$$\mathsf{eq} : \forall X^{\mathbb{T}}.\ X \to X \to \mathbb{P}$$
$$\mathsf{eq}\, X x y \;:=\; \forall p^{X \to \mathbb{P}}.\, px \to py$$

$$\mathsf{Q} : \forall X^{\mathbb{T}} \forall x.\ \mathsf{eq}\, X\, x\, x$$
$$\mathsf{Q}\, X x \;:=\; \lambda pa.a$$

$$\mathsf{R} : \forall X^{\mathbb{T}} \forall x y \forall p^{X \to \mathbb{P}}.\ \mathsf{eq}\, X x y \to px \to py$$
$$\mathsf{R}\, X x y p f \;:=\; fp$$

$$\begin{aligned}
\bot &: \mathbb{P} \\
\mathsf{E}_\bot &: \forall Z^{\mathbb{P}}.\ \bot \to Z \\[4pt]
\land &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\
\mathsf{C} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ X \to Y \to X \land Y \\
\mathsf{E}_\land &: \forall X^{\mathbb{P}} Y^{\mathbb{P}} Z^{\mathbb{P}}.\ X \land Y \to (X \to Y \to Z) \to Z \\[4pt]
\lor &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\
\mathsf{L} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ X \to X \lor Y \\
\mathsf{R} &: \forall X^{\mathbb{P}} Y^{\mathbb{P}}.\ Y \to X \lor Y \\
\mathsf{E}_\lor &: \forall X^{\mathbb{P}} Y^{\mathbb{P}} Z^{\mathbb{P}}.\ X \lor Y \to (X \to Z) \to (Y \to Z) \to Z
\end{aligned}$$

Figure 5.3: Abstract constants for falsity, conjunctions, and disjunctions

The definitions are amazingly simple. Note that the conversion rule is needed to make use of the defining equation of eq. The definition of eq follows the Leibniz characterization of equality established in Exercise 5.2.4.

The above definition of propositional equality is known as **Leibniz equality** and appears already in Whitehead and Russell's Principia Mathematica (1910-1913). Computational type theory also provides for the definition of a richer form of propositional equality using an indexed inductive type family. We will study the definition of inductive equality in Chapter 29. Until then the concrete definition of propositional equality does not matter since all we will be using are the three abstract constants provided by both definitions.

5.4 Abstract Presentation of Propositional Connectives

Like propositional equality, falsity, conjunction, and disjunction can be accommodated with systems of abstract constants, as shown in Figure 5.3. This demonstrates a general abstractness property of logical reasoning. Among the constants in Figure 5.3, we distinguish between **constructors** and **eliminators**. The inductive definitions of falsity, conjunction, and disjunction in Chapter 3 provide the constructors directly as constructors. The eliminators may then be obtained as inductive functions. We have seen the eliminators before in Chapter 3 as explosion rule and match functions (Figure 3.2). If we look at the abstract constants for equality, we can identify eq and Q as constructors and R as eliminator.

There is great beauty to the abstract presentation of the propositional connectives with typed constants. Each constant serves a particular purpose:

- The **formation constants** ($\bot$, $\wedge$, $\vee$) provide the abstract syntax for the respective connectives.
- The **introduction constants** (C, L, R) provide the basic proof rules for the connectives.
- The **elimination constants** ($\mathsf{E}_\bot$, $\mathsf{E}_\wedge$, $\mathsf{E}_\vee$) provide proof rules that for the proof of an arbitrary proposition Z make use of the proof of the respective connective.

We emphasize that the definitions of the constants do not matter for the use of the constants as proof rules. In other words, the definitions of the constants do not contribute to the essence of the propositional connectives, which is fully covered by the types of the constants. The constants can be defined either inductively or impredicatively. The impredicative definitions are purely functional and do not involve inductive definitions. Note that the impredicative characterizations can be read of the types of the elimination constants.

As we have seen, propositional equality can be obtained with typed constants following the formation-introduction-elimination scheme:

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

Note that the impredicative characterization

$$\mathsf{eq}\, X x y \quad\longleftrightarrow\quad \forall p^{X \to \mathbb{P}}.\ p x \to p y$$

and hence the impredicative definition can be read of the type of the elimination constant R.

We will see later that existential quantification can also be incorporated with a systems of typed constants following the formation-introduction-elimination scheme, and that the constants can be defined either inductively or impredicatively.

Exercise 5.4.1 (Impredicative definitions) Define the constructors and eliminators for falsity, conjunction, and disjunction assuming that the logical constants are defined using their impredicative characterizations. Do not use the inductive definitions. Note that we have typed Z in the eliminator for falsity in Figure 5.3 with $\mathbb{P}$ rather than $\mathbb{T}$ to enable an impredicative definition.

Exercise 5.4.2 Prove commutativity of conjunction and disjunction just using the abstract constructors and eliminators.

Exercise 5.4.3 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.

5.5 Declared Constants and Theorems

Assuming constants without justification is something one does not do in type theory. For instance, if we assume a constant of type $\bot$, we can prove everything using the explosion rule, which completely ruins the carefully constructed logical system. A safe way for introducing a constant consists in obtaining it with one of the definitional facilities of computational type theory: inductive type definitions, inductive function definitions, and plain definitions. The definitional facilities are controlled by carefully designed restrictions ensuring that nothing bad can happen (e.g., a proof of falsity).

It will often be useful to hide the definition of a constant and just keep its type. We will speak of **declared constants**. The idea is that we declare a system of typed constants for which we then provide definitions that will be kept confidential. The definitions may be seen as justifications of the constants. Declared constants provide us with a notion of abstraction that is well known from mathematics (e.g., abstract groups) and programming (interfaces and implementations). We remark that the (hidden) definitions of declared constants do not contribute reduction rules to computational equality.

Computational type theory accommodates theorems and the like as declared constants.[1] This makes explicit that when we use a lemma we don't need its proof but just its representation as a typed constant.

Recall that we distinguish between inductive and plain definitions, and for plain definitions further between plain functions and plain constants. We now also distinguish between *plain reducible constants* (no definition hiding) and *plain declared constants* (definition hiding).

We mention that Coq's standard way of introducing a declared constant is stating it as a theorem and constructing the term defining it with a proof script ended by the command *Qed*. If one instead uses the command *Defined*, one obtains a plain reducible constant.

Example: Applicative closure

We use the opportunity to discuss a surprisingly useful lemma for propositional equality known as applicative closure:

$$\mathsf{f\_eq} : \forall XZ^{\mathbb{T}}\,\forall f^{X\to Z}\,\forall xy^{X}.\; x = y \to fx = fy$$

[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.

Using the predecessor function

$$\begin{aligned} \mathsf{P} : \ & \mathsf{N} \to \mathsf{N} \\ \mathsf{P}\,0 \ &:= \ 0 \\ \mathsf{P}\,(\mathsf{S}n) \ &:= \ n \end{aligned}$$

the lemma can be used to give an elegant proof for the injectivity of S:

$$\mathsf{f\_eq}\ \mathsf{N}\ \mathsf{N}\ \mathsf{P}\,(\mathsf{S}x)\,(\mathsf{S}y)\ :\ \mathsf{S}x = \mathsf{S}y \to x = y$$

What makes the proof so concise is the conversion rule, which converts the equation $\mathsf{P}(\mathsf{S}x) = \mathsf{P}(\mathsf{S}y)$ into the target equation $x = y$.

Exercise 5.5.1 Define f_eq as a declared constant (on paper and in Coq).

Exercise 5.5.2 Prove $0 = \mathsf{S}x \to \mathbf{T} = \mathbf{F}$ using f_eq and an inductive function $\mathsf{Z} : \mathsf{N} \to \mathsf{B}$ testing for zero.

Exercise 5.5.3 There is a second form of applicative closure

$$\forall XZ^{\mathbb{T}}\ \forall fg^{X \to Z}\ \forall x^X.\ f = g \to fx = gx$$

that may be used to instantiate equations between functions. Prove this proposition.

6 Inductive Eliminators

For inductive types we can define functions called eliminators that through their types provide proof rules for case analysis and induction, and that through their defining equations provide schemes for defining functions discriminating and recursing 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. The universal eliminator for an inductive type is obtained as a parameterized inductive function that can be instantiated to all inductive functions definable for the inductive type. As it comes to typing, this necessary generality is obtained with dependent return types using return type functions and the conversion rule providing for instantiation. Return type functions generalize the return type predicate we have seen 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:

$$\begin{aligned}
&\mathsf{E_B} : \forall p^{\mathsf{B}\to\mathbb{T}}.\ p\,\mathbf{T} \to p\,\mathbf{F} \to \forall x.px \\
&\mathsf{E_B}\, p\, e_1 e_2\, \mathbf{T} := e_1 \qquad : p\,\mathbf{T} \\
&\mathsf{E_B}\, p\, e_1 e_2\, \mathbf{F} := e_2 \qquad : p\,\mathbf{F}
\end{aligned}$$

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.

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

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

Figure 6.1: Proof diagram for a boolean elimination

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 $\mathsf{B} \to \mathbb{P}$.

Now look at the defining equations of $\mathsf{E_B}$. They are well-typed since the patterns $\mathsf{E_B}\, pab\, \mathbf{T}$ and $\mathsf{E_B}\, pab\, \mathbf{F}$ on the left 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: Boolean Case Analyis and 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**

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

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

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

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.

We remark that a simply typed boolean eliminator

$$\begin{aligned} \mathsf{E_B} &: \ \forall Z^{\mathbb{T}}.\ Z \to Z \to \mathsf{B} \to Z \\ \mathsf{E_B}\, Z\, e_1 e_2\, \mathbf{T} &:= \ e_1 \\ \mathsf{E_B}\, Z\, e_1 e_2\, \mathbf{F} &:= \ e_2 \end{aligned}$$

does not suffice for proving by boolean case analysis.

Second Example: Kaminski's Equation

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

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

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

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

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

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

This was the first time we saw a proof discriminating on a term rather than a variable. Speaking informally, the proof Kaminski's equation proceeds by cascaded discrimination on x, $f\mathbf{T}$, and $f\mathbf{F}$, where the equations recording the discriminations on the terms $f\mathbf{T}$, and $f\mathbf{F}$ are made available as assumptions. While this proof pattern is not primitive in type theory, it can be expressed as shown above. A proof assistant may support this and other proof patterns with specialized tactics.[2]

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

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

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

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

Exercise 6.1.3 (Boolean pigeonhole principle)

a) Prove the boolean pigeonhole principle: $\forall xyz^{B}.\ x = y \vee x = z \vee y = z.$

b) Prove Kaminski's equation based on the instance of the boolean pigeonhole principle for $f(fx)$, fx, and x.

[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.

[2] Coq supports the pattern with the `destruct` tactic and the `eqn` modifier.

Exercise 6.1.4 (Boolean enumeration) Prove $\forall x^{\mathsf{B}}.\ x = \mathbf{T} \vee x = \mathbf{F}$ and use it to prove Kaminski's equation by enumerating x, fx, and $f(fx)$ and solving the resulting 2^3 cases with Coq's congruence tactic.

Exercise 6.1.5 (Eliminator for $\top$)
Recall that $\top$ is an inductive type with exactly one element.

a) Define an eliminator for $\top$ following the design you have seen for B.

b) Use the eliminator to show that all elements of $\top$ are equal.

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 $\mathsf{M_N}$ as an inductive function is straightforward:

$$\begin{aligned} &\mathsf{M_N} :\ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to (\forall n.\, p(\mathsf{S}n)) \to \forall n.\, pn \\ &\mathsf{M_N}\, p\, e_1 e_2\, 0 \ := \ e_1 \qquad : \ p0 \\ &\mathsf{M_N}\, p\, e_1 e_2\, (\mathsf{S}n) \ := \ e_2 n \qquad : \ p(\mathsf{S}n) \end{aligned}$$

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** E_N with the given type, and does the type of E_N really suffice to do proofs by structural induction? The answer to both questions is yes.

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

$$\begin{aligned} E_N :\ & \forall p^{N \to \mathbb{T}}.\ p0 \to (\forall n.\ pn \to p(Sn)) \to \forall n.\ pn \\ E_N\, p\, e_1 e_2\, 0 \ &:=\ e_1 && :\ p0 \\ E_N\, p\, e_1 e_2 (Sn) \ &:=\ e_2\, n\, (E_N\, p\, e_1 e_2 n) && :\ p(Sn) \end{aligned}$$

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

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

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

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

$$E_N\ (\lambda x.\ x + 0 = x)\ \ulcorner 0 + 0 = 0 \urcorner\ \ulcorner \forall x.\ x + 0 = x \to Sx + 0 = Sx \urcorner$$

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 E_N.

We have explained the eliminator for numbers starting from its type using intuitions from proving. A more direct explanation would say that the inductive definition of E_N is the most general inductive function definition for N. This explanation can build on the untyped defining equations

$$\begin{aligned} E\, 0 \ &:=\ a \\ E\,(Sn) \ &:=\ fn(En) \end{aligned}$$

assuming a context providing a and f (the value for zero and the upgrade function).

Exercise 6.2.1 Prove the following propositions in Coq using E_N and M_N.

a) $Sn \neq n$.

b) $n + Sk \neq n$.

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

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

		$x + 0 = x$	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:

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

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

Exercise 6.2.2 Write a term expressing the addition function using an application of $\mathsf{E_N}$. Prove that your addition function agrees with the standard addition function using $\mathsf{E_N}$.

Exercise 6.2.3 Define a match function $\mathsf{M_N}$ with the eliminator $\mathsf{E_N}$ and prove the equations $\mathsf{M_N} a f 0 = a$ and $\mathsf{M_N} a f (\mathsf{S}x) = f x$.

Exercise 6.2.4 Express the Ackermann function using two applications of $\mathsf{E_N}$. Follow the scheme from §1.10. Verify that the specifying equations hold by computational equality.

6.3 Equality of Numbers is Logically Decidable

We now show that equality of numbers is logically decidable.

Fact 6.3.1 $\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \lor x \neq y.$

To prove this fact we need induction on x and case analysis on y. Moreover, it is essential that y is quantified in the inductive hypothesis. We start with the partial proof term

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

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

Figure 6.3: Proof diagram with a quantified inductive hypothesis

The base case follows with case analysis on y:

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

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

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

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

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

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 6.3.2 Define a function $M : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ for truncating subtraction using E_{N} (both for the recursion on the first argument and the discrimination on the second argument). Prove $Mxy = x - y$ using E_{N}.

Exercise 6.3.3 (Boolean equality decider for numbers)
Write a function $\mathsf{eqb} : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$ such that $\forall xy.\ x = y \longleftrightarrow \mathsf{eq}_{\mathsf{N}}\, xy = \mathbf{T}$. Prove the equivalence using E_{N}. Next express eqb using E_{N} for both the recursion and the discrimination.

6.4 Eliminator for Pairs

Recall the inductive type definition for pairs from §1.7:

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

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:

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

Exercise 6.4.1 Prove the following facts for pairs $a : X \times Y$ using the eliminator $\mathsf{E}_\times$:

a) $(\pi_1 a, \pi_2 a) = a$ — η-law

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

Exercise 6.4.2 Use $\mathsf{E}_\times$ to write functions that agree with π_1, π_2, and swap (see §1.7).

Exercise 6.4.3 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.

6.5 Disequality of Types

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

Since B and N both have type $\mathbb{T}_1$, we can write the propositions $\mathsf{N} = \mathsf{B}$ and $\mathsf{N} \neq \mathsf{B}$. So the question is whether we can prove $\mathsf{N} \neq \mathsf{B}$. We can do this with a property distinguishing the two types (Exercise 5.2.5). We choose the predicate

$$p(X^{\mathbb{T}}) \ :=\ \forall x^X y^X z^X.\ x = y \lor x = z \lor y = z$$

saying that a type has at most two elements. I now suffices to prove $p\mathsf{B}$ and $\neg p\mathsf{N}$. With boolean case analysis on the variables x, y, z we can show that p holds for B. Moreover, we can disprove $p\mathsf{N}$ by choosing $x = 0$, $y = 1$, and $z = 2$ and proving

$$(0 = 1 \vee 0 = 2 \vee 1 = 2) \to \bot$$

by disjunctive case analysis and disjointness and injectivity of 0 and S.

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

On paper, it doesn't make sense to work out the proof in more detail since this involves a lot of writing and routine verification. With Coq, however, doing the complete proof is quite rewarding since the writing and the tedious details are taken care of by the 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 6.5.2 Prove the following inequations between types.

a) $\mathsf{B} \neq \mathsf{B} \times \mathsf{B}$

b) $\bot \neq \top$

c) $\bot \neq \mathsf{B}$

d) $\mathsf{B} \neq \top$

e) $\mathbb{P} \neq \top$

f) $\mathsf{B} \neq \mathbb{T}$

Hint: You will need the eliminator for $\top$ (Exercise 6.1.5).

Exercise 6.5.3 Note that one cannot prove $\mathsf{B} \neq \mathsf{B} \times \top$ since one cannot give a predicate that distinguishes the two types. Neither can one prove $\mathsf{B} = \mathsf{B} \times \top$.

6.6 Uniqueness of Procedural Specifications

Recall from §1.11 that we can specify functions with non-recursive unfolding functions. For instance, we can specify the Fibonacci function and the Ackermann function with unfolding functions. Procedural specifications are interesting whenever the recursion pattern of an equational specification does not meet the format required for inductive function definitions. Given an unfolding function, we can ask whether there is a function satisfying the unfolding function (existence), and whether two function satisfying the unfolding function always agree (uniqueness). In Chapter 1 we show that the procedural specifications for the Fibonacci and the Ackermann function are satisfiable. We can now show that these specifications are also unique.

To get the feel for uniqueness proofs, we start with an unfolding function for the addition function:

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

Clearly, the addition function $+$ from §1.2 satisfies the procedural specification: $\forall xy.\ x + y = \mathsf{Add}(+)xy$ follows by discrimination on x and computational equality. Moreover, we can show that all functions satisfying Add agree:

$$\forall fg.\ (\forall xy.\ fxy = \mathsf{Add}\, fxy) \to (\forall xy.\ gxy = \mathsf{Add}\, gxy) \to \forall xy.\ fxy = gxy$$

The proof is by induction on x and equational reasoning (i.e., rewriting and computational equality).

We remark that for every inductive function definition the corresponding procedural specification is unique. In each case induction and discrimination following the recursion and discrimination of the inductive function definition suffice for a proof.

Showing the uniqueness of the procedural specifications for the Ackermann and Fibonacci functions is more challenging. For Ackermann, one starts with induction on x keeping the quantification for y. The base case then follows after discrimination on y. For the successor case, one does a nested induction on y. The base case follows with the outer induction hypothesis. The successor case follows using both inductive hypotheses.

Proving uniqueness of the Fibonacci specification requires another idea. Here we prove

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

rather than just $\forall n.\ fn = gn$ to obtain a strong enough inductive hypothesis.

Exercise 6.6.1 Do the following proofs with the proof assistant:

a) Define the unfolding function for truncating subtraction and show that all functions satisfying it agree.
b) Show that all functions satisfying the unfolding function for the Fibonacci function agree.
c) Show that all functions satisfying the unfolding function for the Ackermann function agree.

6.7 Executive Summary

Inductive function definitions come in a format such that for every inductive type a universal inductive function can be obtained. This **universal eliminator** takes continuations for the value constructors of the type as arguments (we have been using variables like e and e_i for continuations). A particular inductive function for the type can then be obtained by providing the particular continuations. We recall the definition of the universal eliminator for booleans to make the point:

$$\begin{aligned}
\mathsf{E_B} &: \ \forall p^{\mathsf{B}\to\mathbb{T}}.\ p\,\mathbf{T} \to p\,\mathbf{F} \to \forall x.px \\
\mathsf{E_B}\,p\,e_1 e_2\,\mathbf{T} &:= \ e_1 \\
\mathsf{E_B}\,p\,e_1 e_2\,\mathbf{F} &:= \ e_2
\end{aligned}$$

If a constructor is recursive, the respective continuation will take as arguments the recursive calls licensed by structural recursion. Since computation in type theory terminates, setting up recursions for non-recursive inductive functions doesn't hurt. We recall the definition of the universal eliminator for numbers to make the point:

$$\begin{aligned}
\mathsf{E_N} &: \ \forall p^{\mathsf{N}\to\mathbb{T}}.\ p0 \to (\forall n.\ pn \to p(\mathsf{S}n)) \to \forall n.\,pn \\
\mathsf{E_N}\,p\,e_1 e_2\,0 &:= \ e_1 \\
\mathsf{E_N}\,p\,e_1 e_2(\mathsf{S}n) &:= \ e_2\,n\,(\mathsf{E_N}\,p\,e_1 e_2 n)
\end{aligned}$$

To obtain the necessary generality for proving (case analysis and induction), the return types of universal eliminators are obtained with return type functions provided as arguments. The conversion rule is crucial for the use of return type functions.

The universal eliminator for the empty type

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

is special in that it doesn't need a return type function.

We remark that the eliminators for conjunction and disjunction

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

we considered in Chapter 3 (Figure 3.2) don't use return type functions. The reason is that in ordinary mathematical reasoning propositions don't talk about their proofs, so there is no need to account for this dependency.

Functions taking their return type as argument are polymorphic in the number of their arguments. For instance:

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

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

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

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 straightforward:

$$\eta(0,y) := (\mathsf{S}y, 0)$$
$$\eta(\mathsf{S}x, y) := (x, \mathsf{S}y)$$

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

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

by induction on n.

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

For the successor case we prove

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

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

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

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

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

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 $f : X \rightarrow Y$ and $g : Y \rightarrow X$ 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 much elegance at all levels. Cantor's pairing function is a great example for an educated Programming 1 course addressing functional programming and program verification.

It is interesting to look up Cantor's pairing function in the mathematical literature and in Wikipedia, where the computational aspects of the construction are ignored as much as possible. There one typically starts with the encoding function and uses the Gaussian sum formula to avoid the recursion. Then injectivity and surjectivity of the encoding function are shown, which non-constructively yields the existence of the decoding function. The simple recursive definition of the decoding function does not appear.

8 Existential Quantification

An existential quantification $\exists x^t.\, s$ says that the predicate $\lambda x^t.s$ is *satisfiable*, that is, that there is some u such that the proposition $(\lambda x^t.s)u$ is provable. Following this idea, a basic proof of $\exists x^t.\, s$ is a pair (u, v) consisting of a *witness* $u : t$ and a *certificate* $v : (\lambda x^t.s)u$. This design may be realized with an inductive type definition.

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).

8.1 Inductive Definition and Basic Facts

We first assume a formation constant

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

so that we can write an existential quantifications as function applications (as usual, X is treated as implicit argument):

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

Next we assume an introduction constant

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

so that we can prove an existential quantification $\exists x^t.\, s$ by providing a **witness** $u : t$ and a **certificate** $v : (\lambda x^t.s)u$. Finally, we assume an **elimination constant**

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

so that given a proof of an existential quantification we can prove an arbitrary proposition Z by assuming that there is a witness and certificate as asserted by the existential quantification.

We will see that the constants E and $\mathsf{M}_\exists$ provide us with all the proof rules we need for existential quantification. As usual, the definitions of the constants are not needed for proving with existential quantifications.

The constants ex and E can be defined with an inductive type definition:

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

The inductive type definition for ex and E has two *parameters* where the type of the second parameter p depends on the first parameter X. This is the first time we see such a parameter dependence. The inductive definitions for pair types and conjunctions also have two parameters, but there is no dependency. Also, the definition for existential quantification is the first time we see a parameter (p) that is not a type. Moreover, the proof constructor E comes with an additional dependency between its first proper argument x and the type px of its second proper argument. Again, this is the first time we see such a dependency. Inductive type definitions with dependencies between parameters and proper arguments of constructors are standard in computational type theory.

The elimination constant $\mathsf{M}_\exists$ can now be defined as an inductive function:

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

We now recognize $\mathsf{M}_\exists$ as the simply typed match function for existential types. When convenient, we will use the match notation

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

for applications of $\mathsf{M}_\exists$. Note that the elimination restriction applies to all inductive propositions $\mathsf{ex}\,Xp$.

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. Each of the two conversions can be justified with either the η- or the β-law for λ-abstractions. We also have

$$(\exists x.px)=\mathsf{ex}(\lambda x.px)=\mathsf{ex}(p)$$

where the first equation is just a notational change and the second equation is by application of the η-law.

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

	$X:\mathbb{T},\ p:X\to\mathbb{P}$	$\neg(\exists x.px) \longleftrightarrow \forall x.\neg px$	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: $\mathsf{C}\ (\lambda fxa.f(\mathsf{E}_p xa))\ (\lambda fb.\ \text{MATCH}\ b\ [\ \mathsf{E}xa \Rightarrow fxa\ ])$

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

	$X:\mathbb{T},\ p:X\to\mathbb{P}$	$\neg(\exists x.px) \longleftrightarrow \forall x.\neg px$	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
		$\exists x.px$	$\mathsf{E}xa$
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: $\mathsf{C}\ (\lambda fxa.f(\mathsf{E}_p xa))\ (\lambda fb.\ \text{MATCH}\ b\ [\ \mathsf{E}xa \Rightarrow fxa\ ])$

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

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) $(\exists x.px)\to\neg\forall x.\neg px$
c) $((\exists x.px)\to Z)\ \longleftrightarrow\ \forall x.\ px\to Z$
d) $(\exists x.px)\wedge Z\longleftrightarrow\exists x.\ px\wedge Z$
e) $(\exists x.\ px\vee qx)\ \longleftrightarrow\ (\exists x.px)\vee(\exists x.qx)$
f) $\neg\neg(\exists x.px)\ \longleftrightarrow\ \neg\forall x.\neg px$
g) $(\exists x.\neg\neg px)\ \to\ \neg\neg\exists x.px$
h) $\forall X^{\mathbb{P}}.\ X\longleftrightarrow\exists x^{X}.\top$

Exercise 8.1.2 Give a proof term for $(\exists x.px)\to\neg\forall x.\neg px$ using the constants ex, E, and $\mathsf{M}_\exists$. Do not use matches.

Exercise 8.1.3 Verify the following existential characterization of disequality:

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

Exercise 8.1.4 Verify the impredicative characterization of existential quantification:

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

Exercise 8.1.5 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 (Abstract presentation) We have seen that conjunction, disjunction, and propositional equality can be modeled with abstract constants (§5.4). For existential quantification, we may use the constants

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

we have obtained above with inductive definitions.

a) Assuming the constants, prove that the impredicative characterization holds: $\mathsf{Ex}\,Xp \longleftrightarrow \forall Z^{\mathbb{P}}.\,(\forall x.\,px \to Z) \to Z$.

b) Define the constants impredicatively (i.e., not using inductive types).

Exercise 8.1.7 (Intuitionistic drinker) Using excluded middle, one can argue that in a bar populated with at least one person one can always find a person such that if this person drinks milk everyone in the bar drinks milk:

$$\forall X^{\mathbb{T}}\,\forall\,p^{X\to\mathbb{P}}.\ (\exists x^{X}.\top) \to \exists x.\,px \to \forall y.py$$

The fact follows intuitionistically once two double negations are inserted:

$$\forall X^{\mathbb{T}}\,\forall\,p^{X\to\mathbb{P}}.\ (\exists x^{X}.\top) \to \neg\neg\exists x.\,px \to \forall y.\neg\neg\,py$$

Prove the intuitionistic version.

8.2 Barber Theorem

Nonexistence results 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 $\neg(X \longleftrightarrow \neg X)$ as shown in §3.7. ∎

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.

Exercise 8.2.3 Consider the following predicate on types:

$$p(X^{\mathbb{T}}) \;:=\; \exists f g^{X\to X}\forall xy.\;\; fx = y \vee gy = x$$

Prove $p(\mathsf{B})$ and $\neg p(\mathsf{N})$.
Hint: It suffices to consider the numbers 0, 1, 2.

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

a) Prove that all functions $\top \to \top$ have fixed points.

b) Prove that the successor function $\mathsf{S} : \mathsf{N} \to \mathsf{N}$ has no fixed point.

c) For each type $Y = \bot,\ \mathsf{B},\ \mathsf{B} \times \mathsf{B},\ \mathsf{N},\ \mathbb{P},\ \mathbb{T}$ give a function $Y \to Y$ that has no fixed point.

Exercise 8.3.8 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 Executive Summary

We have arrived at a computational type theory where typing is modulo computational equality. There are dependent function types $\forall x^u.v$, applications st, plain definitions, inductive type definitions, and inductive functions definitions. The definitions introduce typed constants, where the constants introduced by plain definitions and inductive function definitions come with equational reduction rules. The resulting reduction system has four essential properties: termination, unique normal forms, type preservation, and canonicity. Types are accommodated as first class values, necessitating a hierarchy of universe types

$$\mathbb{P} \subset \mathbb{T}_1 \subset \mathbb{T}_2 \subset \mathbb{T}_3 \subset \cdots$$

taking types as values. The lowest universe $\mathbb{P}$ is impredicative. There are also lambda expressions with β-reduction and η-equivalence.

In computational type theory, all definable functions are computational. This makes a key difference to set-theoretic mathematics, where functions are merely sets of input-output pairs. Inductive function definitions can be recursive. To ensure termination, recursion must follow the recursion pattern of an inductive type.

Theories are developed as sequences of type-theoretic definitions building on each other. At the lowest level we have definitions accommodating particular propositions. Lemmas and theorems for particular theories (e.g., numbers, lists) are accommodated with declared constants hiding their definitions. Theories will build on each other, but in the end all theories are derived from a small set of type-theoretic principles.

Propositions as Types

Propositions are accommodated as types of the lowest universe. This yields propositions that can quantify over functions, types, propositions, and proofs (proofs appear as elements of propositional types). Propositional equality can be modeled elegantly as Leibniz equality making use of the conversion rule and the impredicativity of $\mathbb{P}$. Powerful lemmas, including induction principles, can be formulated as propositions and can be defined as functions.

Logical reasoning as obtained with the propositions as types approach is intuitionistic reasoning not building in the law of excluded middle. When desired, the law of excluded middle can be assumed.

The propositions as types approach is both natural and powerful. Modeling lemmas as functions and proofs as combination of functions is in perfect correspondence with mathematical practice. Describing functions and combination of functions with terms is an obvious elaboration coming with the benefit that proof checking is obtained as type checking. The propositions as types approach turns out to be a powerful explanation and formalization of what we do with propositions and proofs in mathematical practice. It opens new mathematical possibilities by turning propositions and types into first-class objects. The type-theoretic explanation of the proof rule for induction on numbers is of spectacular elegance. As generations of students have witnessed, informal mathematics just doesn't succeed in giving a clear explanation of what is happening when we do an inductive proof.

Computational type theory gives us an expressive and uniform language for propositions and proofs serving all levels of mathematical reasoning. On the one hand, we can do proofs at a low level with first principles. On the other hand, we can do proofs at a high level using abstractions and lemmas.

Logical constructs like falsity, conjunction, disjunction, existential quantification and equality can be incorporated with typed constants whose definition does not matter for their use. Remarkably, the constants come with functional types providing the proof rules for the logical constructs. The constants may be defined either inductively or impredicatively, where the concrete definitions do not matter for the use of the constructs.[1] The impredicative definitions are purely functional and do not involve inductive definitions.

Proof Assistants

Computational type theories are designed to be implemented as programming languages. We can implement a *verifier* reading a sequence of definitions and checking that everything is well-formed according to the rules of the type theory, a process known as *type checking*. Computational type theories are designed such that type checking can be done algorithmically, and that proof checking is obtained as type checking.

We may assume that a verifier sees a sequence of definitions in fully elaborated form; that is, all implicit arguments have been derived and all notational conveniences (i.e., infix operators) have been removed. This way, the complexity of elaboration can be handled separately by an *elaborator*, and the verifier can be realized with a relatively small program.

At a higher level one has an interactive proof assistant, which is a tool supporting users in developing theories (i.e., sequences of definitions). The user sees the interactive proof assistant as a *command interpreter*. The proof assistant integrates an incremental elaborator and an incremental verifier building a type-checked theory

[1] The inductive definition of equality will be discussed in Chapter 29.

definition by definition. There is also an embedded *tactic interpreter* for type-driven incremental top-down construction of terms. The tactic interpreter executes commands called *tactics* contributing to a term construction initiated by the command interpreter. Besides simple tactics, there are *automation tactics* building complete proofs in one go. Powerful automation tactics exist for propositional and arithmetic reasoning.

The top level of an interactive proof assistant provides commands for constructing terms using the tactic interpreter, type checking, simplifying, and evaluating terms, defining and assuming constants, hiding definitions of constants, querying existing definitions, establishing notations and implicit arguments, setting the details of printing, and loading libraries.

For the engineering of a proof assistant, the separation of verification, elaboration, and incremental term construction with tactics is essential. Concerning the software effort needed, verification ranks lowest, elaboration ranks in the middle, and tactics rank highest. By design, everything produced by tactics and elaboration is checked by the *kernel*, the software component responsible for verification. This way the trusted base of an interactive proof assistant can be kept small.

Recursion in inductive function definitions is tuned down by a *guard condition*. A guard condition must be decidable and must ensure termination. We are assuming a simple and well-understood guard condition in this text. More permissive guard conditions are being used in proof assistants.

The computational type theory presented in this text is compatible with what is implemented by the proof assistant Coq. We take the freedom to assume features not directly available in Coq. Most notably, we use inductive and plain function definitions where Coq only provides plain constant definitions. We make no effort to cover all features of Coq. Every chapter of the text comes with a Coq file realizing the development of the chapter in Coq.

Further Remarks

1. It much simplifies the realization of a proof assistant (and a verifier in particular) that propositions and proofs are derived notions and that proof checking is obtained as type checking.
2. It is fascinating to see how the mathematical notions of propositions, proofs, and theorems are reduced to the computational principles of a type theory.
3. An important aspect of mathematical proving is subgoal and assumption management. In the propositions as types approach subgoal management boils down to type-driven construction of terms, and assumption management is obtained as nesting of typed lambda abstractions and let expressions.
4. A collapsed universe hierarchy $\mathbb{P} \subset \mathbb{T}$ with $\mathbb{T} : \mathbb{T}$ would be nice but is not an option since the *vicious cycle* $\mathbb{T} : \mathbb{T}$ destroys canonicity and consistency of the

system. Nevertheless, for most developments, we can ignore universe levels and have the elaborator check that universe levels can be consistently assigned.

5. We will eventually see methods providing for the construction of functions specified with general terminating recursion.

10 Sum and Sigma Types

In this chapter we learn about certifying functions whose types are described with sum and sigma types. These notions are at the very heart of type theory and will play a main role from now on. Given these notions, structures from mathematical proofs generalize into structures for computational functions.

Sum and sigma types may be seen as computational variants of disjunctions and existential quantifications. While sum types provide disjoint unions of types, sigma types are dependent pair types where the type of the second component depends on the first component (similar to a dependent function type where the type of the result depends on the argument). With sum and sigma types we can write function types specifying an input-output relation. Using such informative function types, we can construct functions together with their correctness proofs, which often is superior to a separate construction of the function and its correctness proof. We speak of certifying functions if the type of the function includes the relational specification of the function. It turns out that the abstract techniques for proof construction (e.g., induction) apply to the construction of certifying functions starting from their types, thus eliminating the need to start with defining equations. Certifying functions are an essential feature of constructive type theories having no equivalent in set-theoretic mathematics. With certifying types we can describe computational situations often lacking adequate descriptions in set-theoretic language.

We also consider computational injections and bijections of types, which are the main tool for relating types. An injection is a pair of an embedding function and its inverse function, and a bijection is a symmetric injection where embedding and inverse function can be swapped. The types of injections and bijections are dependent tuple types.

10.1 Sum Types

We define the family of **sum types** inductively as follows:[1]

$$+\,(X : \mathbb{T},\, Y : \mathbb{T}) : \mathbb{T} \;::=\; \mathsf{L}(X) \mid \mathsf{R}(Y)$$

[1] Relying on the context for disambiguation, we reuse the names L and R also used for the proof constructors of disjunctions.

Sum types come with 3 constructors:

$$+ : \mathbb{T} \to \mathbb{T} \to \mathbb{T}$$
$$\mathsf{L} : \forall X^{\mathbb{T}} Y^{\mathbb{T}}.\ X \to X + Y$$
$$\mathsf{R} : \forall X^{\mathbb{T}} Y^{\mathbb{T}}.\ Y \to X + Y$$

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**.

Sum types are computational variants of disjunctions. In contrast to disjunctions, sum types are not restricted to propositions and are not subject to the elimination restriction. We define a simply-typed match function for sum types as follows:

$$\mathsf{M}_+ : \forall XYZ^{\mathbb{T}}.\ X + Y \to (X \to Z) \to (Y \to Z) \to Z$$
$$\mathsf{M}_+\, XYZ\, (\mathsf{L}\, x)\, e_1 e_2 := e_1 x$$
$$\mathsf{M}_+\, XYZ\, (\mathsf{R}\, y)\, e_1 e_2 := e_2 y$$

As usual, we will use the notation

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

We can use sum types to construct **finite types** of any cardinality:

$\bot$	no element
$\bot + \top$	1 element
$(\bot + \top) + \top$	2 elements
$((\bot + \top) + \top) + \top$	3 elements

We will refer to the types $\bot$ and $\top$ as **void** and **unit** if we are in a context where the fact they are propositions does not matter.

An important application of sum types are so-called **decision types**:

$$\mathcal{D}(X^{\mathbb{T}}) : \mathbb{T} := X + (X \to \bot)$$

We call the 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. In particular, if X is a proposition, a decision of type $\mathcal{D}(X)$ carries either a proof of X or a proof of $\neg X$.

Fact 10.1.1 (Laws for decisions)

1. $\mathcal{D}(\top)$ and $\mathcal{D}(\bot)$.
2. $\forall XY^{\mathbb{T}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \to Y)$.
3. $\forall XY^{\mathbb{P}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \wedge Y)$.
4. $\forall XY^{\mathbb{P}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \vee Y)$.
5. $\forall X^{\mathbb{T}}.\ \mathcal{D}(X) \to \mathcal{D}(X \to \bot)$.
6. $\forall XY^{\mathbb{P}}.\ (X \longleftrightarrow Y) \to (\mathcal{D}(X) \to \mathcal{D}(Y))$.

Proof Straightforward using simply-typed matches. ■

In words we may say, that decidable propositions are closed under the propositional connectives, and that decidability is invariant under propositional equivalence.

Given that decisions are not propositions, it will be handy to have a notation for **propositional equivalence of types**:

$$X \Leftrightarrow Y \;:=\; (X \to Y) \times (Y \to X)$$

We can now write and prove the following fact.

Fact 10.1.2 $\forall XY^{\mathbb{T}}.\ (X \Leftrightarrow Y) \to (\mathcal{D}(X) \Leftrightarrow \mathcal{D}(Y))$.

We define a **universal eliminator for sum types** providing for dependently typed matches:

$$\mathsf{E}_+ :\ \forall XY^{\mathbb{T}}\ \forall p^{X+Y\to\mathbb{T}}.\ (\forall x.\, p(\mathsf{L}\,x)) \to (\forall y.\, p(\mathsf{R}\,y)) \to \forall a.\ pa$$

$$\mathsf{E}_+\ XY p e_1 e_2\ (\mathsf{L}\,x) \;:=\; e_1 x$$

$$\mathsf{E}_+\ XY p e_1 e_2\ (\mathsf{R}\,y) \;:=\; e_2 y$$

Finally, we formulate the *correspondence* between the propositional connectives $\wedge$, $\vee$, $\longleftrightarrow$ and the type constructors $\times$, $+$, $\Leftrightarrow$ with a table:

$\times$	$+$	$\Leftrightarrow$	$\bot$	$\top$	$\to$	computational types in $\mathbb{T}$
$\wedge$	$\vee$	$\longleftrightarrow$	$\bot$	$\top$	$\to$	propositional types in $\mathbb{P}$

The table also includes $\bot$, $\top$, and $\to$, which transparently serve both levels.

Exercise 10.1.3 Give all elements of the type $((\bot + \top) + \top) + \top$.

Exercise 10.1.4 Prove Facts 10.1.1 and 10.1.2.

Exercise 10.1.5 Define a function $\forall X^{\mathbb{T}}\, f^{X\to\mathsf{B}}\, x^X.\ \mathcal{D}(fx = \mathbf{T})$.

Exercise 10.1.6 Prove $\forall X^{\mathbb{T}}.\ (\mathcal{D}(X) \to \bot) \to \bot$.

Exercise 10.1.7 Recall that there is no elimination restriction for $P \longleftrightarrow Q$.

a) Prove $\forall PQ^{\mathbb{P}}.\ (P \longleftrightarrow Q) \Leftrightarrow (P \Leftrightarrow Q)$.

b) Explain why $\forall X^{\mathbb{T}}.\ X \wedge X$ does not type check.

Exercise 10.1.8 Define a *truncation function* $\forall PQ^{\mathbb{P}}.\ P + Q \to P \vee Q$ for sum types. Note that a converse function $\forall PQ^{\mathbb{P}}.\ P \vee Q \to P + Q$ cannot be defined because of the elimination restriction.

Exercise 10.1.9 Define functions as follows:

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

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

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

Exercise 10.1.10 (Constructor laws for sum types)
Prove the constructor laws for sum types using the simply typed match function:

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

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

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

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

Exercise 10.1.11 Prove that double negated disjunction agrees with double negated sum: $\neg\neg(P \vee Q) \longleftrightarrow \neg(P + Q \to \bot)$.

Exercise 10.1.12 (Functional characterization)
Prove $X + Y \Leftrightarrow \forall Z^{\mathbb{T}}.\ (X \to Z) \to (Y \to Z) \to Z$.
Note that the equivalences is analogous to the impredicative characterization of disjunctions.

10.2 Computational Lemmas and Certifying Functions

We have modeled propositions as types such that proving a proposition amounts to constructing a value of the propositional type. With this representation of propositions and proofs in computational type theory, we still use informal proof-oriented language to construct proof terms. In contrast to the basic term-oriented language, proof-oriented language talks at a higher level omitting bureaucratic details. A striking example is induction, where proof-oriented language elegantly handles recursion without explicitly defining recursive functions.

It turns out that proof-oriented language generalizes smoothly to the level of computational types. For instance, if we construct a function of the type

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

we can talk about it in proof-oriented language as if we were constructing an inductive proof of the proposition (Fact 6.3.1)

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

This becomes very concrete when we work with the proof assistant, where both types can be stated as lemmas and where both functions can be constructed with literally the same script. In fact, the interactive proof mode of Coq is designed for types in general, not just propositions.

So from now on, a lemma may have any type, and the construction of a value for a type may be carried out in proof-oriented language and may be called a proof. Thus we may say *prove a type* or *show a type* in place of *construct a value of a type* or *construct a function of a given type.*

We speak of a *propositional lemma* if the type of the lemma is propositional, and of a *computational lemma* if the type of the lemma is not propositional. Propositional lemmas are typically accommodated as declared constants hiding their proofs. For computational lemmas it sometimes is convenient to not hide their proofs so that they can contribute to computational equality. A typical example is a certifying equality decider

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

where equations like

$$(\text{IF } d\,2\,3 \text{ THEN } \mathbf{T} \text{ ELSE } \mathbf{F}) = \mathbf{F}$$

hold by computational equality. Nevertheless, we will use computational lemmas such that the type of the lemma matters for its use, not the details of its definition.

We speak of a **certifying function** if the target type of a function is non-propositional but involves propositions. A typical example of a certifying function is a certifying equality decider of the type

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

From the type it is clear that we have a function that given two numbers decides whether they are equal or not. In fact, the function returns the decision together with a proof certifying the correctness of the decision. This is in contrast to the simple function type

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

which admits all functions returning a boolean. We may say that the type $\forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y)$ is *more informative* than the type $\mathsf{N} \to \mathsf{N} \to \mathsf{B}$, or that the type $\forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y)$ is *certifying* while the type $\mathsf{N} \to \mathsf{N} \to \mathsf{B}$ is not.

A certifying type may be seen as a *specification* of a function. When we construct a function for a certifying type, we construct both a computational function and a correctness proof for the computational function. Often the combined or interleaved construction of the function and the proof is more convenient than giving separate constructions of the function and the proof. For an equality decider for numbers, we could first construct a simply typed function

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

and afterwards a correctness proof

$$\forall xy.\ x = y \longleftrightarrow fxy = \mathbf{T}$$

To conclude, we state the existence of a certifying equality decider for numbers as a lemma for future reference.

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

Proof Same as for Fact 6.3.1. ∎

Exercise 10.2.2 Prove the claims of Fact 10.1.1 using proof-oriented language. Note that with the exception of (1) functions are constructed.

Exercise 10.2.3 Prove $\forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y)$ and $\forall xy^{\mathsf{N}}.\ (x = y) \vee (x \neq y)$ with the proof assistant using the same script for both proofs.

Exercise 10.2.4 Define a function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$ and prove $x = y \longleftrightarrow fxy = \mathbf{T}$.

10.3 Discrete Types

We call a type X **discrete** if we have 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 a type function

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

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

Fact 10.3.1 (Transport of equality deciders)

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. $\forall XY^{\mathbb{T}}.\ \mathcal{E}(X + Y) \to \mathcal{E}(X)$.
5. $\forall XY^{\mathbb{T}}.\ \mathcal{E}(X \times Y) \to Y \to \mathcal{E}(X)$.
6. $\forall XY^{\mathbb{T}}.\ \mathcal{E}(X \times Y) \to \mathcal{E}(Y \times X)$.
7. $\forall XY^{\mathbb{T}}.\ \mathcal{E}(X + Y) \to \mathcal{E}(Y + X)$.

Proof $\mathcal{E}(\mathsf{N})$ is Fact 10.2.1. The remaining claims have straightforward proofs. ∎

Fact 10.3.2 Discreteness propagates backwards through injective functions: $\forall XY^{\mathbb{T}} \forall f^{X \to Y}.\ \mathsf{injective}\, \mathsf{f} \to \mathcal{E}(Y) \to \mathcal{E}(X)$.

Exercise 10.3.3 Proof the claims of Facts 10.3.1 and 10.3.2.

Exercise 10.3.4 Prove $\mathcal{E}(\top \to \bot)$.

10.4 Dependent Pair Types

We have seen pair types $(x, y) : X \times Y$ fixing their component types independently of each other. It turns out that *dependent pair types*

$$\Sigma x : X.\ px$$

where the type of the second component depends by means of a type function $p : X \to \mathbb{T}$ on the first component have many applications. For instance, the type

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

specifies a certifying function that given two numbers x and y yields the distance z between the numbers and a proof that the number returned is in fact the distance.

Dependent pair types $\Sigma x : X.\ px$ are similar to dependent function types

$$\forall x : X.\ px$$

in how they accommodate the dependency and how they generalize their simply typed versions $X \times Y$ and $X \to Y$. This similarity is well reflected in the quantifier-style notations.

A substantial difference between dependent function types and dependent pair types is the fact that dependent function types are native to the type theory while dependent pair types are obtained as inductive types.

Dependent pair types $\Sigma x:X.\,px$ are a computational variant of existential propositions $\exists x:X.\,px$, obtained by generalizing p from a predicate to a type function $p^{X\to\mathbb{T}}$. This leads us to the inductive definition of **dependent pair types**:

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

Similar to the notation $\exists x.s$ for existential types $\mathsf{ex}\,(\lambda x.s)$, we use the notation $\Sigma x.s$ for dependent pair types $\mathsf{sig}\,(\lambda x.s)$. The full types of the constructors for dependent pair types are

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

We will write (x,a) or $(x,a)_p$ for a **dependent pair** $\mathsf{E}\,Xpxa$ and call x and a the **first** and **second component** of the pair. When p is a predicate, we may refer to x as the *witness* and to a as the *certificate*, as we did for existential quantification. We usually write $\mathsf{sig}\,p$ for $\mathsf{sig}\,Xp$. Finally, following common speak, we will often say **sigma type** for dependent pair type.

We extend the correspondence table between propositional type constructors and computational type constructors with $\forall$, $\exists$, and Σ:

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

We define the **simply typed match function** for dependent pair types following the definition of the match function for existential quantifications (§8.1):

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

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

We also define the **universal eliminator for dependent pair types** providing for dependently typed matches:

$$\begin{aligned}\mathsf{E}_\Sigma:&\ \forall X^{\mathbb{T}}\,\forall p^{X\to\mathbb{T}}\,\forall q^{\mathsf{sig}\,p\to\mathbb{T}}.\ (\forall xc.\,q\,(x,c))\to\forall a.\,qa\\ \mathsf{E}_\Sigma\,X\,p\,q\,e\,(x,c)\ :=&\ \ exc\end{aligned}$$

We now construct a certifying distance function.

Fact 10.4.1 (Certifying distance function)
$\forall xy^{\mathsf{N}}\,\Sigma z^{\mathsf{N}}.\ (x+z=y)\vee(y+z=x).$

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

We will see many more certifying functions with dependent pair types as result types.

We have argued before that one can translate between certifying and boolean equality deciders. Using dependent pair types, we can now elegantly specify the translations between certifying and boolean deciders. We introduce a notation for certifying decider types:

$$\mathsf{dec}\, p \;:=\; \forall x.\ \mathcal{D}(px)$$

Fact 10.4.2 (Translations between certifying and boolean deciders)

1. $\forall X\, \forall p^{X \to \mathbb{P}}.\ \ \mathsf{dec}\, p \Leftrightarrow \Sigma f^{X \to \mathsf{B}}.\, \forall x.\ px \longleftrightarrow fx = \mathbf{T}$
2. $\forall X^{\mathbb{T}}.\ \mathcal{E}(X) \Leftrightarrow \Sigma f^{X \to X \to \mathsf{B}}.\, \forall xy.\ x = y \longleftrightarrow fxy = \mathbf{T}$

Proof Exercise 10.4.3. ■

Note that fact specifies the translations and also asserts their existence.

Exercise 10.4.3 Prove the statements of Fact 10.4.2.

Exercise 10.4.4 Define the simply typed match function for sigma types using the universal eliminator for sigma types.

Exercise 10.4.5 Define a *truncation function* $\forall X^{\mathbb{T}}\, \forall p^{X \to \mathbb{P}}.\ \mathsf{sig}\, p \to \mathsf{ex}\, p$ for sigma types using the simply typed match function. Note that the converse function cannot be defined because of the elimination restriction.

Exercise 10.4.6 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 10.4.7 (Functional characterization)
Prove $\mathsf{sig}\, p \;\Leftrightarrow\; \forall Z^{\mathbb{T}}.\ (\forall x.\ px \to Z) \to Z$ using the simply typed match function. Note that the equivalence is analogous to the impredicative characterization of existential quantification.

Exercise 10.4.8 (Equational match law for sum types)
Construct a certifying function $\forall a^{X+Y}.\ (\Sigma x.\ a = \mathsf{L}\,x) + (\Sigma y.\ a = \mathsf{R}\,y)$ using the universal eliminator for sum types. Convince yourself that the simply typed match function for sums does not suffice for the construction.

Exercise 10.4.9 Construct a certifying function $\forall x^{\mathsf{N}}.\ (x = 0) + (\Sigma k.\ x = \mathsf{S}k)$.

Exercise 10.4.10 (Certifying division by 2)
Define a certifying function $\forall x^{\mathsf{N}}\,\Sigma n.\ (x = 2n) + (x = 2n + 1)$.

Exercise 10.4.11 (Certifying division by 2)
Assume a function $F : \forall x^{\mathsf{N}}\,\Sigma n.\ (x = 2n) + (x = 2n + 1)$.

a) Use F to define a function that for a number x yields a pair (n, k) such that $x = 2n + k$ and k is either 0 or 1. Prove the correctness of your function.
b) Use F to define a function that tests whether a number is even. Prove the correctness of your function.

Exercise 10.4.12 (Certifying Distance)
Assume a function $D : \forall xy^{\mathsf{N}}\,\Sigma z.\ (x+z = y)+(y+z = x)$ and prove the following:

a) $\pi_1(Dxy) = (x - y) + (y - x)$.
b) $\pi_1(D\,3\,7) = 4$.
c) $x - y = \text{IF } \pi_2(Dxy) \text{ THEN } 0 \text{ ELSE } \pi_1(Dxy)$.

Note that a definition of D is not needed for the proofs since all information needed about D is in its type. Hint: For (a) and (c) discriminate on Dxy and simplify. What remains are equations involving truncating subtraction only.

10.5 Projections and Skolem Correspondence

We assume a type function $p : X \to \mathbb{T}$ and define **projections** yielding the first and the second component of dependent pairs $a^{\mathsf{sig}\,p}$:

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

Note that the type of π_2 is given using the projection 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 a conversion step unfolding the definition of π_1.

We now use the projections to define a translation function

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

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

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

decomposing a certifying function $\forall x\,\Sigma y.pxy$ into a simply typed 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 correspondence between computational types.

Fact 10.5.1 (Skolem correspondence)
$\forall XY^{\mathbb{T}}\ \forall p^{X\to Y\to\mathbb{T}}.\ (\forall x\,\Sigma y.\ pxy) \Leftrightarrow (\Sigma f\ \forall x.\ px(fx))$.

Proof The translation $\to$ can be defined as $\lambda F.\ (\lambda x.\ \pi_1(Fx),\ \lambda x.\ \pi_2(Fx))$. The converse translation $\leftarrow$ can be defined as $\lambda ax.\ (\pi_1 ax,\ \pi_2 ax)$. ■

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

The Skolem correspondence (Fact 10.5.1) is of practical importance. Often we will prove a computational lemma $\forall x\,\Sigma y.\ pxy$ to then obtain a function $f^{X\to Y}$ satisfying the *specification* $\forall x.\ px(fx)$.

We use the term Skolem correspondence since there is a ressemblance with the correspondence for Skolem functions in first-order logic.

Exercise 10.5.2 Define a simply typed match 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 10.5.3 Express π_1 with the simply typed match function M_Σ. Convince yourself that π_2 cannot be expressed with M_Σ.

Exercise 10.5.4 Express the projections π_1 and π_2 for sigma types with terms t_1 and t_2 using the universal eliminator E_Σ such that $\pi_1 \approx t_1$ and $\pi_2 \approx t_2$.

Exercise 10.5.5 (Eta law) Prove the eta law $\mathsf{E}\,(\pi_1 a)(\pi_2 a) = a$ for dependent pairs $a : \mathsf{sig}\,p$. Convince yourself that the proof requires the universal eliminator E_Σ and cannot be done with the match function M_Σ.

Exercise 10.5.6 (Propositional Skolem) Due to the elimination restriction for existential quantification, the direction $\to$ of the Skolem correspondence cannot be shown for all types X and Y if Σ-quantification is replaced with existential quantification. (The unprovability persists if excluded middle is assumed.) There are two noteworthy exceptions. Prove the following:

a) $\forall Y^{\mathbb{T}}\, \forall p^{\mathsf{B}\to Y\to\mathbb{P}}.\ (\forall x \exists y.\ pxy) \to \exists f\, \forall x.\ px(fx)$.

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

Remarks: (1) The boolean version (a) generalizes to all finite types X presented with a covering list. (2) The unprovability of the propositional Skolem correspondence persists if the law of excluded is assumed. The difficulty is in proving the existence of the function f since functions must be constructed with computational principles. (3) In the literature, f is often called a choice function and the direction $\to$ of the Skolem correspondence is called a choice principle.

Exercise 10.5.7 (Existential quantification) Existential quantifications $\mathsf{ex}\, X p$ are subject to the elimination restriction if and only if X is not a propositional types. Thus a function extracting the witness can only be defined if X is a proposition.

a) Define projections π_1 and π_2 for quantifications $\mathsf{ex}\, X p$ where X is a proposition.

b) Prove $a = \mathsf{E}\,(\pi_1 a)(\pi_2 a)$ for all $a : \mathsf{ex}\, X p$ where X is a proposition.

Exercise 10.5.8 (Injectivity laws)
One would think that the injectivity laws for dependent pairs

$$\mathsf{E}\, x c = \mathsf{E}\, x' c' \ \to\ x = x'$$
$$\mathsf{E}\, x c = \mathsf{E}\, x c' \ \to\ c = c'$$

are both provable. While the first law is easy to prove, the second law cannot be shown in general in computational type theory. This certainly conflicts with intuitions that worked well so far. The problem is with subtleties of dependent type checking. In Chapter 29, we will show that the second injectivity law does hold if the type 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 on paper, check it with Coq to find out where it breaks. The obvious proof idea that rewrites $\pi_2(\mathsf{E}\, x c)$ to $\pi_2(\mathsf{E}\, x c')$ does not work since there is no well-typed rewrite predicate validating the rewrite.

10.6 Injections and Bijections

Given a function $f^{X\to Y}$ we say that a function $g^{Y\to X}$ **inverts** f if $\forall x.\ g(fx) = x$. We also say that g is an **inverse function** for f. We may picture the inversion property as a roundtrip property allowing us to go with f from X to Y and to return with g from Y to X to exactly the x we started from. It will be convenient to have the **inversion predicate**

$$\begin{aligned} \mathsf{inv} :\ & \forall XY^{\mathbb{T}}.\ (X \to Y) \to (Y \to X) \to \mathbb{P} \\ \mathsf{inv}_{XY}\, g f \ :=\ & \forall x.\ g(fx) = x \end{aligned}$$

Fact 10.6.1 (Inverse functions)

1. $\mathsf{inv}\, g f \to \mathsf{injective}\, f \wedge \mathsf{surjective}\, g$.
2. $\mathsf{inv}\, g f \to \mathsf{injective}\, g \vee \mathsf{surjective}\, f \to \mathsf{inv}\, f g$.
3. All inverse functions of a surjective function agree:
   $\mathsf{surjective}\, f \to \mathsf{inv}\, g f \to \mathsf{inv}\, g' f \to \forall y. gy = g' y$.

Proof The proofs are straightforward but interesting. Exercise. ∎

We define **injection types**

$$\mathcal{I}(X : \mathbb{T},\ Y : \mathbb{T}) : \mathbb{T} \ ::=\ \mathsf{I}(f : X \to Y,\ g : Y \to X,\ \mathsf{inv}\, g f)$$

and call their inhabitants **injections**. An injection $\mathcal{I}XY$ is as an embedding of the type X into the type Y where different elements of X are mapped to different elements of Y. We say that **X embeds into Y** or that **X is a retract of Y** if there is an injection $\mathcal{I}XY$.

Technically, injection types are specialized dependent tuple types. An injection type $\mathcal{I}XY$ can be expressed as a nested sigma type $\Sigma f^{X \to Y}\, \Sigma g^{Y \to X}.\ \mathsf{inv}\, g f$.

Fact 10.6.2 (Reflexivity and Transitivity)
$\mathcal{I}XX$ and $\mathcal{I}XY \to \mathcal{I}YZ \to \mathcal{I}XZ$.

Proof Exercise. ∎

Fact 10.6.3 (Transport)
$\mathcal{I}XY \to \mathcal{E}(Y) \to \mathcal{E}(X)$.

Proof Exercise. ∎

Fact 10.6.4 $\mathcal{I}(X \to \mathsf{B})X \to \bot$. That is, $X \to \mathsf{B}$ does not embed into X.

Proof Follows from Fact 8.3.4 since the inverse function of the embedding function is surjective. ∎

We define **bijection types**

$$\mathcal{B}(X:\mathbb{T},\ Y:\mathbb{T}):\mathbb{T}\ ::=\ \mathsf{B}(f:X\to Y,\ g:Y\to X,\ \mathsf{inv}\,g f,\ \mathsf{inv}\,f g)$$

and call their inhabitants **bijections**. A bijection consists of two functions inverting each other in both directions.

We say that two types X and Y are **in bijection** if there is a bijection $\mathcal{B}XY$. Bijectivity is a basic notion in mathematics. A bijection between X and Y establishes a one-to-one correspondence between the elements of X and the elements of Y. Speaking informally, a bijection between X and Y says that X and Y are renamed versions of each other. In contrast to a bijection $\mathcal{B}XY$, the target type Y of an injection $\mathcal{I}XY$ may have elements not appearing as images of elements of the source type X.

Fact 10.6.5 $\mathcal{B}XY \to \mathcal{I}XY \times \mathcal{I}YX$.

Fact 10.6.6 $\mathcal{B}XY \to \mathcal{I}XZ \to \mathcal{I}YZ$.

Fact 10.6.7 (Reflexivity, Symmetry, Transitivity)
Bijectivity is a computational equivalence relation on types:
$\mathcal{B}XX$, $\mathcal{B}XY \to \mathcal{B}YX$, and $\mathcal{B}XY \to \mathcal{B}YZ \to \mathcal{B}XZ$.

Fact 10.6.8 All empty types are in bijection: $(X \to \bot) \to (Y \to \bot) \to \mathcal{B}XY$.

Proof We have functions $X \to \bot$ and $Y \to \bot$. Computational falsity elimination gives us functions $X \to Y$ and $Y \to X$. The inversion properties hold vacuously with the assumptions. ■

Note that computational falsity elimination is essential in this proof.

We have already established a prominent bijection.

Fact 10.6.9 N and $\mathsf{N} \times \mathsf{N}$ are in bijection.

Proof Cantor pairing as developed in Chapter 7. ■

Exercise 10.6.10 Prove the claims of all facts stated without proofs.

Exercise 10.6.11 Give a function $f^{\mathsf{N}\to\mathsf{N}}$ that has two non-agreeing inverse functions.

Exercise 10.6.12 (Boolean functions) Prove the following:

a) A function $\mathsf{B} \to \mathsf{B}$ is injective if and only if it is surjective.

b) Every injective function $\mathsf{B} \to \mathsf{B}$ agrees with either the identity or boolean negation.

c) Every injective function $\mathsf{B} \to \mathsf{B}$ has itself as unique inverse function:
$\forall f^{\mathsf{B}\to\mathsf{B}}.\ \mathsf{inv}\,f f \wedge (\forall g.\ \mathsf{inv}\,g f \to \mathsf{agree}\,g f)$.

Hint: A proof assistant helps to manage the necessary case analysis.

Exercise 10.6.13 Show that the following types are in bijection using bijection types.

a) B and $\top + \top$.

b) $X \times Y$ and $Y \times X$.

c) $X + Y$ and $Y + X$.

d) X and $X \times \top$.

Exercise 10.6.14 Show that $\mathcal{B}XY$ and $\Sigma f^{X \to Y} \Sigma g^{Y \to X}.\ \mathsf{inv}\, g f \wedge \mathsf{inv}\, f g$ are in bijection:

a) $\mathcal{B}XY \Leftrightarrow \Sigma f^{X \to Y} \Sigma g^{Y \to X}.\ \mathsf{inv}\, g f \wedge \mathsf{inv}\, f g$.

b) $\mathcal{B}\ (\mathcal{B}XY)\ (\Sigma g^{Y \to X}.\ \mathsf{inv}\, g f \wedge \mathsf{inv}\, f g)$.

Exercise 10.6.15 Prove $\mathcal{B}\,\mathsf{N}\,\mathsf{B} \to \bot$.

Exercise 10.6.16
Prove that bijections transport equality deciders: $\mathcal{B}XY \to \mathcal{E}(X) \to \mathcal{E}(Y)$.

Exercise 10.6.17 (Product and sum types are in bijection with sigma types)
Sigma types can express pair types $X \times Y$ and sum types $X + Y$ up to bijection.

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.

The functions for the bijections can be defined using the simply typed match functions. The proofs of the roundtrip equations, however, require the universal eliminators (with one exception).

10.7 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 can be 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

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

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 $\mathcal{O}(X)$. Moreover, $\mathcal{O}^n(\bot)$ is a finite type with exactly n elements. For instance, the elements of $\mathcal{O}^3(\bot)$ are $\emptyset$, $^\circ\emptyset$, $^{\circ\circ}\emptyset$. We refer to the types $\mathcal{O}^n(\bot)$ as **numeral types** and reserve the notation $\mathsf{F}_n := \mathcal{O}^n\bot$.

Fact 10.7.1 (Transport)
Option types transport equality deciders in both directions:
$\forall X^{\mathbb{T}}.\ \mathcal{E}(X) \Leftrightarrow \mathcal{E}(\mathcal{O}(X))$.

Proof Exercise. ∎

Fact 10.7.2 (Numeral types) All numeral types are discrete: $\forall n. \mathcal{E}(\mathcal{O}^n(\bot))$.

Proof By induction on n using Facts 10.3.1(1) and 10.7.1. ∎

Exercise 10.7.3 Define the universal eliminator for option types and prove the constructor laws. Hint: Follows with the techniques we have seen for numbers.

Exercise 10.7.4 Prove $\forall a^{\mathcal{O}(X)}.\ a \neq \emptyset \Leftrightarrow \Sigma x.\ a = {}^\circ x$.
Note that direction $\rightarrow$ needs computational falsity elimination.

Exercise 10.7.5 Prove $\forall f^{X \rightarrow \mathcal{O}(Y)}.\ (\forall x.\ fx \neq \emptyset) \rightarrow \forall x \Sigma y.\ fx = {}^\circ y$.
Note the need for computational falsity elimination. Show that assuming the above claim yields computational falsity elimination in the form $\forall X^{\mathbb{T}}.\ \bot \rightarrow X$ (instantiate with $X := \bot$, $Y := X$, and $f = \lambda\_.\emptyset$).

Exercise 10.7.6 Prove $\forall x^{\mathcal{O}^3(\bot)}.\ x = \emptyset \vee x = {}^\circ\emptyset \vee x = {}^{\circ\circ}\emptyset$.

Exercise 10.7.7 (Bijectivity) Show that the following types are in bijection:

a) $\top$ and $\mathcal{O}(\bot)$.
b) B and $\mathcal{O}^2(\bot)$.
c) $\mathcal{O}(X)$ and $X + \top$.
d) N and $\mathcal{O}(\mathsf{N})$.

Exercise 10.7.8 Prove $\mathcal{B}XY \rightarrow \mathcal{B}(\mathcal{O}X)(\mathcal{O}Y)$.

Exercise 10.7.9 (Decidable quantification)
Let d be a certifying decider for $p : \mathcal{O}^n(\bot) \rightarrow \mathbb{T}$. Prove the following:

a) $(\Sigma x.px) + (\forall x.px \rightarrow \bot)$
b) $\mathcal{D}(\forall x.px)$
c) $\mathcal{D}(\Sigma x.px)$

Hints: Use induction on the cardinality n. With Coq use an inductive function $\mathsf{Fin} : \mathsf{N} \rightarrow \mathbb{T}$ with $\mathsf{Fin}\,0 := \bot$ and $\mathsf{Fin}\,(\mathsf{S}n) := \mathcal{O}(\mathsf{Fin})$ to obtain the numeral types $\mathcal{O}^n(\bot)$.

Exercise 10.7.10 (Counterexample) Find a type X and functions $f : X \to \mathcal{O}(X)$ and $g : \mathcal{O}(X) \to X$ such that you can prove $\mathsf{inv}\, g\, f$ and disprove $\mathsf{inv}\, f\, g$.

Exercise 10.7.11 (Truncating subtraction with flag)
Define a recursive function $f : \mathsf{N} \to \mathsf{N} \to \mathcal{O}(\mathsf{N})$ that yields $^\circ(x - y)$ if the subtraction $x - y$ doesn't truncate, and $\emptyset$ if the subtraction $x - y$ truncates. Prove the equation $fxy = \text{IF } y - x \text{ THEN } {}^\circ(x - y) \text{ ELSE } \emptyset$.

Exercise 10.7.12 (Kaminski reloaded)
Prove $\forall f^{\mathcal{O}^3(\bot) \to \mathcal{O}^3(\bot)} \forall x.\ f^8(x) = f^2(x)$.
Hint: Prove $\forall x^{\mathcal{O}^3(\bot)}.\ x = \emptyset \lor x = {}^\circ\emptyset \lor x = {}^{\circ\circ}\emptyset$ and use it to enumerate x, fx, f^2x, and f^3x. This yields 3^4 cases, all of which are solved by Coq's congruence tactic.

10.8 Bijection Theorem for Option Types

Given a bijection between $\mathcal{O}(X)$ and $\mathcal{O}(Y)$, we can construct a bijection between X and Y. This intuitively simple result needs a technically involved proof using a certifying function.

Suppose f and g provide a bijection between $\mathcal{O}(X)$ and $\mathcal{O}(Y)$. We first define a bijective function $X \to Y$. To map x, we look at $f(^\circ x)$. If $f(^\circ x) = {}^\circ y$, we map x to y. If $f(^\circ x) = \emptyset$, we have $f\emptyset = {}^\circ y$ for some y and map x to y. The other direction is symmetric.

Defining a function $X \to Y$ as described above involves a computational falsity elimination. For this reason it is crucial that the function is first obtained as a certifying function so that the definition of the function is not needed for proving its required properties.

Lemma 10.8.1 Let g invert $f^{\mathcal{O}(X) \to \mathcal{O}(Y)}$. Then
$\forall x\, \Sigma y.\ \text{MATCH } f(^\circ x)\ [\, ^\circ y' \Rightarrow y = y' \mid \emptyset \Rightarrow f\emptyset = {}^\circ y\,]$.

Proof Case analysis of $f(^\circ x)$. If $f(^\circ x) = {}^\circ y$, we return y. If $f(^\circ x) = \emptyset$, we do case analysis on $f\emptyset$. If $f\emptyset = {}^\circ y$, we return y. Otherwise, we have a contradiction since g inverts f. We finish with computational falsity elimination. ∎

Theorem 10.8.2 (Bijection) $\forall XY.\ \mathcal{B}\,(\mathcal{O}(X))\,(\mathcal{O}(Y)) \to \mathcal{B}XY$.

Proof Let f and g provide a bijection $\mathcal{B}\,(\mathcal{O}(X))\,(\mathcal{O}(Y))$. By Lemma 10.8.1 we obtain functions $f' : X \to Y$ and $g' : Y \to X$ such that

$$\forall x.\ \text{MATCH } f(^\circ x)\ [\, ^\circ y \Rightarrow f'x = y \mid \emptyset \Rightarrow f\emptyset = {}^\circ f'x\,]$$
$$\forall y.\ \text{MATCH } g(^\circ y)\ [\, ^\circ x \Rightarrow g'y = x \mid \emptyset \Rightarrow g\emptyset = {}^\circ g'y\,]$$

We show $g'(f'x) = x$, the other inversion follows analogously. We discriminate on $f^\circ x$. If $f^\circ x = {}^\circ y$, we have $g^\circ y = {}^\circ x$ and $g'(f'x) = x$ follows. If $f^\circ x = \emptyset$, we have $f\emptyset = {}^\circ y$ for some y and $g'(f'x) = x$ follows. ■

Exercise 10.8.3 Prove the bijection theorem for options with the proof assistant not looking at the code we provide. Formulate a lemma providing for the two symmetric cases in the proof of Theorem 10.8.2.

10.9 Truncations

We now define a type constructor $\Box : \mathbb{T} \to \mathbb{P}$ mapping a type X to a proposition $\Box X$ such that $\Box X$ is provable if and only if X is inhabited:

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

We call $\Box$ **truncation operator** and $\Box X$ the **truncation of** X. Moreover, we may read a proposition $\Box X$ as X **is inhabited**. Truncation deletes computational information but keeps propositional information. The elimination restriction applies to truncations $\Box X$ except if X is a proposition. It turns out that conjunction, disjunction, and existential quantification can be characterized by the truncations of their computational counterparts (pair types, sum types, and sigma types).

Fact 10.9.1 (Logical truncations)

1. $(P \wedge Q) \ \longleftrightarrow \ \Box(P \times Q)$.
2. $(P \vee Q) \ \longleftrightarrow \ \Box(P + Q)$.
3. $(\exists x.px) \ \longleftrightarrow \ \Box(\Sigma x.px)$.

Fact 10.9.2 (Characterizations of truncations)

1. $\forall X^{\mathbb{T}}.\ \Box X \longleftrightarrow \forall P^{\mathbb{P}}.\ (X \to P) \to P$.
2. $\forall X^{\mathbb{T}}.\ \Box X \longleftrightarrow \exists x^X.\top$.
3. $\forall P^{\mathbb{P}}.\ \ \Box(P) \longleftrightarrow P$.

Exercise 10.9.3 Define a simply typed match function for inhabitation types and prove the facts stated above.

Exercise 10.9.4 Prove the following truncation laws:

a) $X \to \Box X$

b) $\Box X \to (X \to \Box Y) \to \Box Y$

c) $\Box X \to (X \to \bot) \to \bot$

d) $\mathsf{XM} \to \Box X \vee (X \to \bot)$

Exercise 10.9.5 Think of $\square(\mathsf{sig}\, p)$ as existential quantification and prove the following:

a) $\forall x^X.\ px \to \square(\mathsf{sig}\, p)$.

b) $\forall Z^{\mathbb{P}}.\ \square(\mathsf{sig}\, p) \to (\forall x.\, px \to Z) \to Z$.

Exercise 10.9.6 (Advanced material) We define the type functions

$$\begin{aligned}\mathsf{choice}\ X\,Y \ &:=\ \forall p^{X\to Y\to\mathbb{P}}.\ (\forall x \exists y.\, pxy) \to \exists f\, \forall x.\, px(fx)\\ \mathsf{witness}\ X \ &:=\ \forall p^{X\to\mathbb{P}}.\ \mathsf{ex}\, p \to \mathsf{sig}\, p\end{aligned}$$

You will show that there are translations between $\forall XY^{\mathbb{T}}.\, \mathsf{choice}\ X\,Y$ and $\square(\forall X^{\mathbb{T}}.\, \mathsf{witness}\ X)$. The translation from choice to witness needs to navigate cleverly around the elimination restriction. The presence of the inhabitation operator is essential for this direction.

a) Prove $\square(\forall X^{\mathbb{T}}.\, \mathsf{witness}\ X) \to (\forall XY^{\mathbb{T}}.\, \mathsf{choice}\ X\,Y)$.

b) Prove $(\forall XY^{\mathbb{T}}.\, \mathsf{choice}\ X\,Y) \to \square(\forall X^{\mathbb{T}}.\, \mathsf{witness}\ X)$.

c) Convince yourself that the equivalence

$$(\forall XY^{\mathbb{T}}.\, \mathsf{choice}\ X\,Y) \longleftrightarrow \square(\forall X^{\mathbb{T}}.\, \mathsf{witness}\ X)$$

is not provable since the two directions require different universe levels for X and Y.

Hints. For (a) use $f := \lambda x.\, \pi_1(WY(px)(Fx))$ where W is the witness operator and F is the assumption from the choice operator. For (b) use the choice operator with the predicate $\lambda a^{\Sigma(X,p).\,\mathsf{ex}\,p}.\, \lambda b^{\Sigma(X,p).\,\mathsf{sig}\,p}.\ \pi_1 a = \pi_1 b$ where $p^{X\to\mathbb{P}}$. Keeping the arguments of the predicate abstract makes it possible to obtain the choice function f before the inhabitation operator is removed. The proof idea is taken from the Coq library ChoiceFacts.

10.10 Notes

Mathematics comes with a rich language for describing proofs. Using this language, we can write informal proofs for human readers that can be elaborated into formal proofs when needed. The tactic level of the Coq proof assistant provides an abstraction layer for the elaboration of informal proofs making it possible to delegate to the proof assistant many of the details coming with formal proofs.

It turns out that the idea of informal proof extends to the construction of *certifying functions*, which are functions whose type encompasses an input-output relation. The *proof-style construction* of certifying functions turns out to be beneficial in practice. It comes for free in a proof assistant since the tactic level addresses types in general, not just propositional types. The proof-style construction of certifying functions is guided by the specifying type and uses high-level building blocks like induction. Typically, 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 $\forall x.\, px(fx)$.

Most propositions have functional readings. Once we describe propositions as computational types using sum and sigma types, their proofs become certifying functions that may be used in computational contexts. Certifying functions carry their specifications in their types and may be seen as computational lemmas. Like propositional lemmas, certifying functions are best described with high-level proof outlines, which may be translated into formal proof terms using the tactic interpreter of a proof assistant.

Product, sum, and sigma types are obtained as inductive types. In contrast to the propositional variants, where simply typed eliminators suffice, constructions involving product, sum, and sigma types often require dependently typed eliminators. Existential quantifications and sigma types are distinguished from the other inductive types we have encountered so far in that their value constructors model a dependency between witness and certificate using a type function.

With certifying functions, the elimination restriction of the type theory plays a prominent role. A simplified computational type theory would not have a special universe for propositions and thus avoid the elimination restriction. Such a theory would model propositions as computational types using product, sum, and sigma types. Adding an impredicative universe of propositions pays off in that excluded middle can be assumed without destroying the computational interpretation of sum and sigma types. Computational type theory without a special universe of propositions is known as Martin-Löf type theory [20]. Having an impredicative universe of propositions is a key feature of the computational type theory underlying the Coq proof assistant [9].

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 explicit models of computation (e.g., Turing machines).

We call a predicate *decidable* if it can be characterized with a decision function. Decidable predicates are algorithmically decidable. Moreover, decidable predicates are logically decidable in that the law of excluded middle holds 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) follow from the propagation laws for *decisions* (i.e., the 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 do have equality deciders.

We also considered *option types*, an inductively defined variant of sum types extending a given base type with a new element. Option types preserve discreteness of their base type. Based on option types one can define finite types.

We saw several function constructions in this chapter where computational falsity elimination was essential.

A significant construction in this chapter is the bijection theorem for option types. We learned that such functions are best defined as certifying functions so that proofs about such functions do not require the concrete definition of the function.

11 Extensionality

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

11.1 Extensionality Assumptions

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 assumptions expressible as propositions:

- **Function extensionality**
  FE $:=\ \forall X^{\mathbb{T}}\ \forall p^{X\to\mathbb{T}}.\ \forall f g^{\forall x.px}.\ (\forall x. fx = gx) \to f = g$
- **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 ensures 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 (§29.2).

We can represent boolean functions $f^{B \to B}$ as boolean pairs $(f\,\mathbf{T}, f\,\mathbf{F})$. Under FE, the boolean function can be fully recovered from the pair.

Fact 11.1.1 $\mathsf{FE} \to \forall f^{B \to B}.\ \ f = (\lambda ab.\ \text{IF } b \text{ THEN } \pi_1 a \text{ ELSE } \pi_2 a)\,(f\,\mathbf{T}, f\,\mathbf{F})$.

Exercise 11.1.2 Prove the following:

a) $\mathsf{FE} \to \forall f g^{B \to B}.\ f\mathbf{T} = g\mathbf{T} \to f\mathbf{F} = g\mathbf{F} \to f = g$.

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

Exercise 11.1.3 Prove the following:

a) $\mathsf{FE} \to \forall f^{\top \to \top}.\ f = \lambda a^{\top}.a$.

b) $\mathsf{FE} \to \mathcal{B}\,(\top \to \top)\,\top$.

c) $\mathsf{FE} \to \mathsf{B} \neq (\top \to \top)$.

d) $\mathsf{FE} \to \mathcal{B}\,(\mathsf{B} \to \mathsf{B})\,(\mathsf{B} \times \mathsf{B})$.

e) $\mathsf{FE} \to \mathcal{E}(\mathsf{B} \to \mathsf{B})$.

11.2 Set Extensionality

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

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

Exercise 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 We define **set extensionality** as

$$\mathsf{SE} := \forall X^{\mathbb{T}}\,\forall pq^{X \to \mathbb{P}}.\ (\forall x.\ px \longleftrightarrow qx) \to p = q$$

Prove the following:

a) $\mathsf{FE} \to \mathsf{PE} \to \mathsf{SE}$.

b) $\mathsf{SE} \to \mathsf{PE}$.

c) $\mathsf{SE} \to (\forall x.\ x \in p \longleftrightarrow x \in q) \to p = q$.

d) $\mathsf{SE} \to p - (q \cup r) = (p - q) \cap (p - r)$.

11.3 Proof Irrelevance

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

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

Note that PI says that all propositions are unique.

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

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

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

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 unique since $\top$ is unique. The claim follows. ∎

Exercise 11.3.3 Prove $\mathcal{D}(\mathsf{unique}(\top + \bot))$ and $\mathcal{D}(\mathsf{unique}(\top + \top))$.

Exercise 11.3.4 Prove the following for all types X:

a) $\mathsf{unique}(X) \to \mathcal{E}(X)$.

b) $X \to \mathsf{unique}(X) \to \mathcal{B}X\top$.

Exercise 11.3.5 Prove the following:

a) Uniqueness propagates forward through surjective functions:
$\forall XY^{\mathbb{T}}\forall f^{X \to Y}.\ \mathsf{surjective}\,\mathsf{f} \to \mathsf{unique}(X) \to \mathsf{unique}(X)$.

b) Uniqueness propagates backwards through injective functions:
$\forall XY^{\mathbb{T}}\forall f^{X \to Y}.\ \mathsf{injective}\,\mathsf{f} \to \mathsf{unique}(Y) \to \mathsf{unique}(X)$.

Exercise 11.3.6 Prove $\mathsf{FE} \to \mathsf{unique}\,(\top \to \top)$.

Exercise 11.3.7 Assume PI and $p^{X \to \mathbb{P}}$. Prove $\forall xy\ \forall ab.\ x = y \to (x, a)_p = (y, b)_p$.

Exercise 11.3.8 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 11.3.9 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.

Exercise 11.3.10 Assume functions $E : \mathbb{P} \to A$ and $D : A \to \mathbb{P}$ embedding $\mathbb{P}$ into a proposition A. That is, we assume $\forall P^{\mathbb{P}}.\ D(EP) \longleftrightarrow P$. Prove that A is not unique. Remark: Later we will show Coquand's theorem (32.4.1), which says that $\mathbb{P}$ embeds into no proposition.

11.4 Notes

There is general agreement that a computational type theory should be extensional, that is, prove FE and PE. In our case, we may assume FE and PE as constants. There are general results saying that adding the extensionality assumptions is consistent, that is, does not enable a proof of falsity. There is research underway aiming at a computational type theory integrating extensionality assumptions 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 assumed constants may get stuck on one of the constants before a canonical term is reached.

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

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 the 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

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

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.

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

Exercise 12.1.5 Make sure you can prove the de Morgan laws for disjunction and existential quantification (not using XM).

Exercise 12.1.6 Prove that $\forall PQR^{\mathbb{P}}.\ (P \to Q) \vee (Q \to R)$ is equivalent to 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 (De Morgan for universal quantification)
Prove XM $\longleftrightarrow \forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}}.\ \neg(\forall x.px) \to (\exists x.\neg px)$.
Hint: For direction $\leftarrow$ prove $P \vee \neg P$ with $X := P \vee \neg P$ and $\lambda\_.\bot$, and exploit that $\neg\neg(P \vee \neg P)$ is provable.

Exercise 12.1.9 (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 Wodka if this person drinks Wodka.

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 Wodka. 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.

Exercise 12.1.10 (Drinker implies excluded middle)
When formulated in full generality

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

the drinker proposition implies excluded middle. The proof was found by Dominik Kirst in March 2023 and goes as follows. We show $P \vee \neg P$ for some proposition P. To do so, we instantiate the drinker proposition with the type $X := \mathcal{O}(P \vee \neg P)$ and the predicate defined by $p(^\circ\_) := \neg P$ and $p(\emptyset) := \top$. We now obtain some a such that $pa \to \forall y.py$. If a is obtained with some, a gives us a proof of the claim $P \vee \neg P$. Otherwise, we have $\forall y.py$ and prove $\neg P$. We assume P and obtain a contradiction with the some case of p.

Kirst's proof exploits that we are in a logical framework where proofs are values. There is a paper [27] discussing the drinker paradox and suggesting it does not imply excluded middle in a more conventional logical framework.

Exercise 12.1.11 (Dual drinker)
Prove that the so-called dual drinker proposition

$$\forall X\ \forall p^{X\to\mathbb{P}}.\ X \to \exists x.\ (\exists y.py) \to px$$

is equivalent to 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 23.7.2).

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:

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

Exercise 12.2.2 Prove the equivalences of Fact 12.2.1.

Exercise 12.2.3 Prove the following propositions:

$$\begin{aligned} \neg(P \land Q) &\longleftrightarrow \neg\neg(\neg P \lor \neg Q) \\ (\neg P \to \neg Q) &\longleftrightarrow \neg\neg(Q \to P) \\ (\neg P \to \neg Q) &\longleftrightarrow (Q \to \neg\neg P) \\ (P \to Q) &\to \neg\neg(\neg P \lor Q) \end{aligned}$$

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

Exercise 12.2.5 Prove the following implications:

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

Also prove the converse directions using excluded middle.

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

$$\begin{gathered} P \lor \neg P \\ \neg\neg P \to P \\ \neg(P \land Q) \to \neg P \lor \neg Q \\ (\neg P \to \neg Q) \to Q \to P \\ ((P \to Q) \to P) \to P \\ (P \to Q) \to \neg P \lor Q \\ (P \to Q) \lor (Q \to P) \end{gathered}$$

Exercise 12.2.7 (Double negation shift)
An prominent logical law is double negation shift:

$$\mathsf{DNS} \;:=\; \forall X^{\mathbb{T}}\, \forall p^{X \to \mathbb{P}}.\, (\forall x.\, \neg\neg px) \to \neg\neg\forall x.\, px$$

DNS is provable for finite types but unprovable in general. Prove the following:

a) $\forall p^{\mathsf{B}\to\mathbb{P}}.\ (\forall x.\ \neg\neg px) \to \neg\neg\forall x.\ px$

b) $\neg\neg\mathsf{XM} \longleftrightarrow \mathsf{DNS}$

c) $\forall X^{\mathbb{T}}\ \forall p^{X\to\mathbb{P}}.\ \neg\neg(\forall x.\ px) \to (\forall x.\ \neg\neg px)$

Hint: Direction $\leftarrow$ of (b) follows with $\lambda P.P \vee \neg P$.

Exercise 12.2.8 (Double negation shift for existential quantification)
Prove $\mathsf{XM} \longleftrightarrow \forall X^{\mathbb{T}}\ \forall p^{X\to\mathbb{P}}.\ (\neg\neg\exists x.\ px) \to \exists x.\ \neg\neg px$.
Hint: For direction $\leftarrow$ prove $P \vee \neg P$ with $X := P \vee \neg P$ and $p := \lambda\_.\top$, and exploit that $\neg\neg(P \vee \neg P)$ is provable.

12.3 Definite Propositions

We define **definite propositions** as propositions for which excluded middle holds:

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

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

We may see definite propositions as propositionally decided propositions. Computationally decided propositions are always propositionally decided, but not necessarily vice versa.

Fact 12.3.2

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

Fact 12.3.3 (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 12.3.4 (Definite de Morgan) $\mathsf{definite}\ P \vee \mathsf{definite}\ Q \ \to\ \neg(P \wedge Q) \ \longleftrightarrow\ \neg P \vee \neg Q$.

Exercise 12.3.5 Prove the above facts.

12.4 Stable Propositions

We define **stable propositions** as propositions where double negation elimination is possible:

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

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

Fact 12.4.1 $\mathsf{XM} \longleftrightarrow \forall P^{\mathbb{P}}.\ \mathsf{stable}\, P$.

Definite propositions are stable, but not necessarily vice versa.

Fact 12.4.2 Definite propositions are stable: $\forall P^{\mathbb{P}}.\ \mathsf{definite}\, P \to \mathsf{stable}\, P$.

A negated proposition $\neg P$ where P is a variable is stable but not definite.

Fact 12.4.3 (Characterization) $\mathsf{stable}\, P \longleftrightarrow \exists Q^{\mathbb{P}}.\ P \longleftrightarrow \neg Q$.

Corollary 12.4.4 Negated propositions are stable: $\forall P^{\mathbb{P}}.\ \mathsf{stable}(\neg P)$.

Fact 12.4.5 $\top$ and $\bot$ are stable.

Fact 12.4.6 (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 12.4.7 (Extensionality) Stability is invariant under propositional equivalence: $(P \longleftrightarrow Q) \to \mathsf{stable}\, P \to \mathsf{stable}\, Q$.

Fact 12.4.8 (Classical reasoning rules for stable claims)

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

The rules say that when we prove a stable claim, we can assume for every proposition P that it is definite or stable. Note that the second rule follows from the first rule since definiteness implies stability.

Exercise 12.4.9 Prove the above facts.

Exercise 12.4.10 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) $\mathsf{stable}\, Q \to \neg(P_1 \land P_2) \to (\neg P_1 \lor \neg P_2 \to Q) \to Q$.

c) $\mathsf{stable}\, Q \to (\neg P_1 \to \neg P_2) \to ((P_2 \to P_1) \to Q) \to Q$.

Exercise 12.4.11 Prove $(\forall x.\, \mathsf{stable}\,(px)) \to \neg(\forall x.px) \longleftrightarrow \neg\neg\exists x.\neg px$.

Exercise 12.4.12 Prove $\mathsf{FE} \to \forall f g^{\mathsf{N} \to \mathsf{B}}.\ \mathsf{stable}(f = g)$.

Exercise 12.4.13 Prove $\mathsf{XM} \longleftrightarrow \forall P^{\mathbb{P}}\, \exists Q^{\mathbb{P}}.\ P \longleftrightarrow \neg Q$.

Exercise 12.4.14 We define **classical variants** of conjunction, disjunction, and existential quantification:

$$
\begin{array}{rcll}
P \land_c Q & := & (P \to Q \to \bot) \to \bot & \neg(P \to \neg Q) \\
P \lor_c Q & := & (P \to \bot) \to (Q \to \bot) \to \bot & \neg P \to \neg\neg Q \\
\exists_c\, x.px & := & (\forall x.\, px \to \bot) \to \bot & \neg(\forall x.\, \neg px)
\end{array}
$$

The definitions are obtained from the impredicative characterizations of $\land$, $\lor$, 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) $P \land Q \to P \land_c Q$ and $P \land_c Q \longleftrightarrow \neg\neg(P \land Q)$.

b) $P \lor Q \to P \lor_c Q$ and $P \lor_c Q \longleftrightarrow \neg\neg(P \lor Q)$.

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

d) $P \lor_c \neg P$.

e) $\neg(P \land_c Q) \longleftrightarrow \neg P \lor_c \neg Q$.

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

g) $P \land_c Q$, $P \lor_c Q$, and $\exists_c\, x.\, px$ are stable.

12.5 Variants of Excluded Middle

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

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

TVS is equivalent to the conjunction of XM and PE.

Fact 12.5.1 $\mathsf{TVS} \longleftrightarrow \mathsf{XM} \land \mathsf{PE}$.

Proof We show TVS → 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 TVS → XM and XM ∧ PE → TVS are also straightforward. ■

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

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

Altogether we have the following hierarchy: TVS ⇒ XM ⇒ IXM ⇒ WXM.

Exercise 12.5.2 Prove TVS $\longleftrightarrow \forall XYZ{:}\mathbb{P}.\ X = Y \vee X = Z \vee Y = Z$. Note that the equivalence characterizes TVS without using $\top$ and $\bot$.

Exercise 12.5.3 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 12.5.4 Prove $(\forall X^{\mathbb{T}}.\ X = \top \vee X = \bot) \to \bot$.

Exercise 12.5.5 (Weak excluded middle)

a) Prove XM → WXM.

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

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

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 → XM.

Exercise 12.5.6 (Implicational excluded middle)

a) Prove XM → IXM.

b) Prove IXM → 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 → IXM, IXM → XM, and XM → TVS.

12.6 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 may have social reasons tracing back to early advocates of intuitionistic reasoning who 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 assumptions 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 implications:

$$\mathsf{independent}\,(P) \to \mathsf{consistent}\,(P)$$
$$\mathsf{consistent}\,(P) \to \neg\mathsf{disprovable}\,(P)$$
$$\mathsf{provable}\,(P) \to \neg\mathsf{independent}\,(P)$$

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 (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 13.1.2 (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 13.1.1. ∎

Exercise 13.1.3 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 13.1.4 Let $P \to Q$ be provable. Show that P and Q are both independent if P is consistent and Q is unprovable.

Exercise 13.1.5 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 $\neg\mathsf{provable}\,(\bot) \longleftrightarrow \neg\forall P^{\mathbb{P}}.\ \mathsf{provable}\,(P)$.

13.2 Consistency

Fact 13.2.1 (Consistency) The following propositions are equivalent:

1. $\neg$ provable $(\bot)$.
2. consistent $(\neg\bot)$.
3. $\exists P.$ consistent (P).
4. $\forall P.$ provable $(P) \to$ consistent (P).
5. $\forall P.$ disprovable $(P) \to \neg$provable (P).

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

$2 \to 3$. Trivial.

$3 \to 1$. Suppose P is consistent. We assume provable $\bot$ and show provable $(\neg P)$. Follows by PK.

$1 \to 4$. We assume that $\bot$ is unprovable, P is provable, and $\neg P$ is provable. By PMP we have provable $\bot$. Contradiction.

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

$1 \to 5$. Follows with PMP.

$5 \to 1$. Assume disprovable $(\bot) \to \neg$provable $(\bot)$. It suffices to show disprovable$(\neg\bot)$, which follows with PI. ∎

Exercise 13.2.2 We may consider more abstract provability predicates

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

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

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

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

Exercise 13.2.3 (Hilbert style assumptions) The assumptions PI, PK, and PC can be obtained from the simpler assumptions

$$\text{PK}' : \forall PQ.\ \text{provable}\,(P \to Q \to P)$$
$$\text{PS} : \forall PQZ.\ \text{provable}\,((P \to Q \to Z) \to (P \to Q) \to P \to Z)$$

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.

Part II

Numbers and Lists

14 Numbers

Numbers $0, 1, 2, \ldots$ constitute the basic infinite data structure. Starting from the inductive definition of numbers, we develop a computational theory of numbers based on computational type theory. The main topic of this chapter is the ordering of numbers. In the next few chapters we will explore Euclidean division, least witnesses, size recursion, and greatest common divisors. There is much beauty in developing the theory of numbers from first principles. The art is building up the right definitions and the right theorems in the right order (variation of a statement by Kevin Buzzard).

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 10.2.1. ■

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 [24]):

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

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 14.6.13.

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 $x + Sy = S(x + y)$ 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 $x \neq x + Sy$.

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

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

14.5 Order

We define the order relation on numbers using truncating subtraction:

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

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:

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

Fact 14.5.1 The following equations hold by computational equality:

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

We define several certifying operators that for two numbers decide how they are related by the order.

Fact 14.5.2 (Case analysis)

1. $\mathcal{D}(x \leq y)$
2. $(x \leq y) + (y < x)$
3. $(x < y) + (x = y) + (y < x)$ (trichotomy)
4. $x \leq y \to (x < y) + (x = y)$

Proof All four claims follow by induction on x with y quantified followed by discrimination on y. Claim 1 may also be obtained as a consequence of Fact 14.1.2, and Claim 3 may also be obtained as consequence of Claims 2 and 4. ■

Fact 14.5.3 (Contraposition) $\neg(y < x) \to x \le y$.

Proof Follows with Fact 14.5.2 (2). ■

Lemma 14.5.4 $x \le y \to x + (y - x) = y$.

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 $x \le y$ and show $\mathsf{S}(x + (y - x)) = \mathsf{S}y$, which follows by the inductive hypothesis. ■

Fact 14.5.5 (Existential Characterization) $x \le y \longleftrightarrow \exists k.\ x + k = y$.

Proof Direction $\to$ follows with Lemma 14.5.4, and direction $\leftarrow$ follows with Fact 14.4.1 (3). ■

14.6 More Order

Fact 14.6.1

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

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

Reflexivity follows with Fact 14.4.1 (4).

For transitivity, we assume $x + a = y$ and $y + b = z$ using Fact 14.5.5. Then $z = x + a + b$. Thus $x \le z$ by (1).

For antisymmetry, we assume $x + a = y$ and $x + a \le x$ using Fact 14.5.5. By (3) we have $a = 0$, and thus $x = y$. ■

Fact 14.6.2 (Strict transitivity)

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

Proof We show (1), (2) is similar. Using Fact 14.5.5, 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 14.6.1 (1). ∎

Fact 14.6.3

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

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 (2). Claim 3 follows from (2). For Claim 4 we assume $x + a = y$ using Fact 14.5.5 and show $x \leq x + a + z$ using Fact 14.6.1 (1). Claim 5 follows from (4). Claim 6 follows with discrimination on y and (5). ∎

Fact 14.6.4 (Equality by Contradiction) $\neg(x < y) \to \neg(y < x) \to x = y$.

Proof Follows by contraposition (Fact 14.5.3) and antisymmetry. ∎

Fact 14.6.5 $x - y \leq x$

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 14.6.1 (2). The claim follows by transitivity. ∎

Exercise 14.6.6 (Tightness) Prove $x \leq y \leq \mathsf{S}x \to x = y \lor y = \mathsf{S}x$.

Exercise 14.6.7 (Negation Facts) Formulate Facts 14.5.3 and 14.6.4 as equivalences and prove them.

Exercise 14.6.8 Prove $x \leq y \longleftrightarrow x < y \lor x = y$.

Exercise 14.6.9 Prove $y > 0 \to y - \mathsf{S}x < y$.

Exercise 14.6.10 Prove $x + y \leq x + z \to y \leq z$.

Exercise 14.6.11 Define a function $\forall xy.\ x \leq y \to \Sigma k.\ x + k = y$.

Exercise 14.6.12 Define a boolean decider for $x \leq y$ and prove its correctness.

Exercise 14.6.13 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.

Exercise 14.6.14 Prove $\neg\exists x^{\mathsf{N}}\forall y.\ y \leq x \to \exists z.\ z < y$.

14.7 Automation Tactic Lia

Coq's automation tactic *lia* (linear integer arithmetic) provides an abstract treatment of numbers that frees us from knowing the basic definitions and the basic lemmas. This makes a dramatic difference for more involved proofs where otherwise the low level details of arithmetic would be overwhelming.

Lia cannot do type sums. Still, constructing a certifying decider like

$$\forall xy.\ (x \leq y) + (y < x)$$

can be done without recursion. The trick is doing a computational case analysis on $x - y$, which determines the result decision and then use lia to obtain the certificates.

From now on we will write proofs involving numbers assuming the abstraction level provided by lia.

Exercise 14.7.1 (Certifying deciders with lia)
Define deciders of the following types using lia and not using induction.

a) $\forall xy.\ (x \leq y) + (y < x)$
b) $\forall xy.\ (x < y) + (x = y) + (y < x)$
c) $\forall xy.\ (x \leq y) + \neg(x \leq y)$
d) $\forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y)$

14.8 Bounded Quantification

We will use the notations

$$\begin{aligned} \forall k{\leq}n.\ pk &:= \forall k.\ k \leq n \to pk \\ \exists k{\leq}n.\ pk &:= \exists k.\ k \leq n \land pk \\ \Sigma k{\leq}n.\ pk &:= \Sigma k.\ k \leq n \times pk \end{aligned}$$

for bounded quantifications over numbers.

We speak of a **decidable type function** $p^{X \to \mathbb{T}}$ if we have a certifying decider $\forall x.\, \mathcal{D}(px)$. Formally, we define **decider types**

$$\mathsf{dec}\, p \;:=\; \forall x.\, \mathcal{D}(px) \qquad\qquad (p : X \to \mathbb{T})$$

Fact 14.8.1 (Bounded Quantification)
Let $p^{\mathsf{N} \to \mathbb{T}}$ be a decidable type function on numbers. Then there are decision functions as follows:

1. $\forall n.\, (\Sigma k{\le}n.\, pk) + (\forall k{\le}n.\, pk \to \bot)$
2. $\forall n.\, \mathcal{D}(\forall k{\le}n.\, pk)$
3. $\forall n.\, \mathcal{D}(\Sigma k{\le}n.\, pk)$

Proof By induction on n and using that equality of numbers is decidable. ∎

Exercise 14.8.2 The detailed proof of Fact 14.8.1 is interesting and teaches a few lessons to the beginner. Do it with Coq using lia.

The variant of Fact 14.8.1 where $k \le n$ is replaced with $k < n$ has a less redundant proof since the base cases of the inductions are vacuously satisfied. Fact 14.8.1 can be recovered from the variant by using it with the type function $\lambda k.p(\mathsf{S}k)$. Prove the variant and obtain Fact 14.8.1 from it using Coq.

14.9 Complete Induction

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. Computationally, complete induction says that when we compute a function for a number x, we can obtain the function for all $y < x$ by recursion.

Fact 14.9.1 (Complete Induction)
$\forall p^{\mathsf{N} \to \mathbb{T}}.\, (\forall x.\, (\forall y.\, y < x \to py) \to px) \to \forall x.px.$

Proof 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 to prove the equivalent claim

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

by structural induction on the upper bound 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 14.6.2). ∎

Note that the definition of the function $\forall nx.\ x < n \to px$ needed for the complete induction operator employs computational falsity elimination in the base case.

Chapter 17 will introduce a generalization of complete induction called size recursion.

Section 15.4 will discuss interesting examples for the use of complete induction.

Exercise 14.9.2 Do this exercise using the automation tactic lia.

a) Define a certifying function $\forall xy.\ (x \leq y) + (y < x)$.

b) Prove a complete induction lemma.

c) Formulate the procedural specification

$$f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$
$$f\,x\,y \;=\; \text{IF } \ulcorner x \leq y \urcorner \text{ THEN } x \text{ ELSE } f\,(x - \mathsf{S}y)\,y$$

as an unfolding function using the function from (a). A function realizing the procedural specification computes the remainder of the Euclidean division of x by $\mathsf{S}y$ by means of repeated subtraction. Note that y stays unchanged throughout the recursion.

d) Prove that all functions satisfying the procedural specification agree. Hint: Use complete induction on x. There is no obvious proof using structural induction.

e) Let f be a function satisfying the procedural specification.

   i) Prove $\forall xy.\ fxy \leq y$.

   ii) Prove $\forall xy.\ \Sigma k.\ x = k \cdot \mathsf{S}y + fxy$.

We postpone the construction of a function satisfying the procedural specification since this requires additional techniques we will study in Chapter 15.

14.10 Notes

Our definition of the order predicate deviates from Coq's inductive definition. Coq comes with a very helpful automation tactic lia for linear arithmetic that proves almost all of the results in this chapter and that frees the user from knowing the exact definitions and lemmas. For lia is does not matter whether our definition or Coq's definition of order is used. All our further Coq developments will rely on lia and Coq's definition of order.

The inclined reader may compare the computational development of arithmetic given here with Landau's [19] classical development from 1929.

15 Euclidean Division

We study functions for Euclidean division. Besides a function computing by structural recursion, we consider an algorithm obtaining quotient and remainder with repeated subtraction. The study of the repeated subtraction algorithm requires the use of complete induction.

15.1 Certifying Version

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 $b \le y$. We will construct functions that given x and y compute a and b. We first define a **relational specification**:

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

Given $\delta xyab$, we say that a is the **quotient** and b is the **remainder** of x and $\mathsf{S}y$. Considering Euclidean division for x and $\mathsf{S}y$ instead of x and y eliminates the division-by-zero problem.

To compute a and b from x and y, we need an algorithm. We start with a naive algorithm that recurses on x:

- If $x = 0$, then $a = b = 0$.
- If $x = \mathsf{S}x'$, recursion gives us a' and b' such that $x' = a' \cdot \mathsf{S}y + b'$ and $b' \le y$. Now we distinguish two cases:
  - If $b' = y$, then $a = \mathsf{S}a'$ and $b = 0$.
  - If $b' \ne y$, then $a = a'$ and $b = \mathsf{S}b'$.

We describe the algorithm with three so-called *derivation rules* for δ. The rules are formulated as propositions formulating the correctness conditions for the algorithm. There is a separate derivation rule for each case the algorithm considers. Note how the rules account for recursion.

Fact 15.1.1 (Derivation rules)
The following rules (i.e., propositions) hold for all numbers x, y, a, b:

- $\delta_1 :\quad \delta 0 y 0 0$
- $\delta_2 :\quad \delta xyab \to b = y \to \delta(\mathsf{S}x)y(\mathsf{S}a)0$
- $\delta_3 :\quad \delta xyab \to b \ne y \to \delta(\mathsf{S}x)ya(\mathsf{S}b)$

Proof Straightforward. Rule δ_3 follows with Fact 14.5.2 (4). ■

The derivation rules have operational readings. Given x and y, one can determine numbers a and b such that $\delta xyab$ holds using the derivation rules and recursion on x:

- If $x = 0$, then $a = b = 0$.
- If $x = \mathsf{S}x'$, $\delta x'ya'b'$, and $b' = y$, then $a = \mathsf{S}a'$ and $b = 0$.
- If $x = \mathsf{S}x'$, $\delta x'ya'b'$, and $b' \neq y$, then $a = a'$ and $b = \mathsf{S}b'$.

If you were to reinvent the algorithm and its correctness proof, you might start with the specification δ and decide on structural recursion on x and on the equality test in the successor case. The derivation rules then appear as proof obligations for the correctness proof.

We first construct a certifying division function.

Fact 15.1.2 (Certifying division function) $\forall xy.\Sigma ab.\ \delta xyab$.

Proof By induction on x with y fixed. In the base case ($x = 0$) we choose $a = b = 0$ following δ_1. In the successor case, we have $x = a \cdot \mathsf{S}y + b$ and $b \leq y$ by the inductive hypothesis (i.e., by recursion) and need to show $\mathsf{S}x = a' \cdot \mathsf{S}y + b'$ and $b' \leq y$. If $b = y$, we choose $a' = \mathsf{S}a$ and $b' = 0$ following δ_2. If $b \neq y$, we choose $a' = a$ and $b' = \mathsf{S}b$ following δ_3. ■

Corollary 15.1.3 (D and M)
There are functions $D^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ and $M^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ such that $\forall xy.\ \delta xy(Dxy)(Mxy)$.

Proof Let $F : \forall xy.\Sigma ab.\ \delta xyab$. We define D and M as $Dxy := \pi_1(Fxy)$ and $Mxy := \pi_1(\pi_2(Fxy))$. Now $\pi_2(\pi_2(Fxy))$ is a proof of $\delta xy(Dxy)(Mxy)$ (up to conversion). ■

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

15.2 Simply Typed Version

The algorithm underlying the proof of Fact 15.1.2 can be formulated explicitly with a simply typed function:

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

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

We say that a function $f^{N \to N \to N \times N}$ **satisfies the relational specification** δ if $\forall xy.\ \delta xy\,(\pi_1(fxy))\,(\pi_2(fxy))$.

Fact 15.2.1 (Correctness) Δ satisfies δ.

Proof By induction on x with y fixed. The base case follows with δ_1. 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 $\delta(\mathsf{S}x)y\,(\pi_1(\Delta(\mathsf{S}x)y))\,(\pi_2(\Delta(\mathsf{S}x)y))$ using δ_2 and δ_3. ■

The proofs of Facts 15.1.2 and 15.2.1 are very similar. If you verify the proofs by hand, you will find the proof of Fact 15.1.2 simpler since it doesn't have to verify that Δ is doing the right thing.

15.3 Uniqueness

Next, we show the uniqueness of δ. We choose a detailed proof using induction.

Fact 15.3.1 (Uniqueness)
$(b \le y) \to (b' \le y) \to (a \cdot \mathsf{S}y + b = a' \cdot \mathsf{S}y + b') \to a = a' \wedge b = b'$.

Proof By induction on a with a' quantified, followed by discrimination on a'.

The case $a = a' = 0$ is straightforward.

Assume $a = 0$ and $a' = \mathsf{S}a_2$. Then $b = \mathsf{S}y + a_2 \cdot \mathsf{S}y + b'$ and $b \le y$. Contradiction by strictness (Fact 14.6.3 (2)).

The case $a = \mathsf{S}a_1$ and $a' = 0$ is symmetric.

Let both a and a' be successors. Then $a_1 \cdot \mathsf{S}y + b = a_2 \cdot \mathsf{S}y + b'$ by injectivity of S and injectivity of $+$ (Fact 14.2.2). Thus $a_1 = a_2$ and $b = b'$ by the inductive hypothesis. ■

Corollary 15.3.2 $\delta xyab \to \delta xya'b' \to a = a' \wedge b = b'$.

Corollary 15.3.3 All functions satisfying δ agree.

Proof Let f and g be functions satisfying δ. We show $fxy = gxy$. We have $fxy = (a, b)$, $gxy = (a', b')$, $a \cdot \mathsf{S}y + b = a' \cdot \mathsf{S}y + b'$, and $b, b' \le y$. By Fact 15.3.1 we have $a = a'$ and $b = b'$. The claim follows. ■

The uniqueness of δ has important applications. To see one application, we give two additional derivation rules for δ describing an algorithm that determines a and b by subtracting $\mathsf{S}y$ from x as long as $x > y$. We speak of **repeated subtraction**.

Fact 15.3.4 (Derivation rules) The following rules hold for all numbers x, y, a, b:

- $\delta_4: \quad x \le y \to \delta x y 0 x$
- $\delta_5: \quad x > y \to \delta(x - \mathsf{S}y) y a b \to \delta x y (\mathsf{S}a) b$

Proof Rule δ_4 is obvious. For rule δ_5 we assume $x > y$, $x - \mathsf{S}y = a \cdot \mathsf{S}y + b$, and $b \le y$, and show $x = \mathsf{S}y + (a \cdot \mathsf{S}y + b)$. By the first assumption it suffices to show $x = \mathsf{S}y + (x - \mathsf{S}y)$, which holds by Lemma 14.5.4. ∎

Fact 15.3.5 For all numbers x and y the functions D and M from Corollary 15.1.3 satisfy the following equations:

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

Proof By Facts 15.1.3 and 15.3.2 (uniqueness of δ) it suffices to show

$$\begin{gathered} \delta x y \,(\text{IF } \ulcorner x \le y \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(D(x - \mathsf{S}y)y)) \\ (\text{IF } \ulcorner x \le y \urcorner \text{ THEN } x \text{ ELSE } M(x - \mathsf{S}y)y) \end{gathered}$$

We do case analysis on $(x \le y) + (x > y)$. If $x \le y$, the claim reduces to $\delta x y 0 x$, which follows with δ_4. If $x > y$, the claim reduces to

$$\delta x y \,(\mathsf{S}(D(x - \mathsf{S}y)y))\,(M(x - \mathsf{S}y)y)$$

which with δ_5 reduces to

$$\delta (x - \mathsf{S}y) y \,(D(x - \mathsf{S}y)y)\,(M(x - \mathsf{S}y)y)$$

which is an instance of Corollary 15.1.3. ∎

Fact 15.3.5 is remarkable, both as it comes to the result and to the proof. It states the important result that the functions D and M we have constructed with structural recursion on numbers satisfy procedural specifications employing repeated subtraction. Moreover, the proof shows a new pattern. It hinges on the uniqueness of the relational specifications δ, the rules δ_4 and δ_5 explaining the role of subtraction, and Corollary 15.1.3 specifying D and M in terms of δ (no further information about D and M is needed).

Exercise 15.3.6 Show uniqueness of δ using trichotomy for a and a'. This way arithmetical reasoning without induction suffices for the proof. There is an extension nia of lia that knows about multiplication and can handle the cases obtained with trichotomy.

Exercise 15.3.7 Let $F : \forall xy. \Sigma ab.\ \delta xyab$ and $fxy := (\pi_1(Fxy),\ \pi_1(\pi_2(Fxy)))$.

a) Prove that f satisfies the relational specification $\delta xy(\pi_1(fxy))(\pi_2(fxy))$.

b) Prove that f satisfies the procedural specification
$fxy = \text{IF } \ulcorner x \le y \urcorner \text{ THEN } (0, x) \text{ ELSE LET } (a, b) = f\,(x - \mathsf{S}y)\,y \text{ IN } (\mathsf{S}a, b)$.

Remark: Both proof are straightforward when done with a proof assistant. Checking the details rigorously is annoyingly tedious if done by hand. The second proof best follows the proof of Fact 15.3.5 using uniqueness of δ (Fact 15.3.2) and the derivation rules for repeated subtraction. No induction is needed. A closely related proof will appear with Fact 15.4.2.

Exercise 15.3.8 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 15.3.9 Prove $x \cdot \mathsf{SS}z + 1 \neq y \cdot \mathsf{SS}z + 0$ using uniqueness of Euclidean division (Fact 15.3.1).

Exercise 15.3.10 We define **divisibility** and **primality** as follows:

$$k \mid x \ := \ \exists n.\ x = n \cdot k$$
$$\mathsf{prime}\,x \ := \ x \ge 2 \land \forall k.\ k \mid x \to k = 1 \lor k = x$$

Prove that both predicates are decidable. Hint: First prove

$$x > 0 \to x = n \cdot k \to n \le x$$
$$x > 0 \to k \mid x \to k \le x$$

and then exploit that bounded quantification preserves decidability (Fact 14.8.1).

15.4 Repeated Subtraction with Complete Induction

A common algorithm for Euclidean division is *repeated subtraction*: Subtract $\mathsf{S}y$ from x as often as it can be done without truncation; then the number of subtractions is the quotient, and the part of x remaining is the remainder of the division.

The procedural specification for a function realizing this algorithm looks as follows:

$$f : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \times \mathsf{N}$$
$$f x y \;=\; \text{IF } \ulcorner x \le y \urcorner \text{ THEN } (0, x) \text{ ELSE LET } (a, b) = f\,(x - \mathsf{S}y)\,y \text{ IN } (\mathsf{S}a, b)$$

The recursion in the specification is not structural. However, the algorithm terminates since each recursion step decreases x. Using complete induction, we can obtain a certifying function for Euclidean division using the algorithm,.

Fact 15.4.1 (Certifying division with complete induction) $\forall xy. \Sigma ab.\ \delta xyab$.

Proof By complete induction on x with y fixed. Following the repeated subtraction algorithm, we consider two cases.

If $x \le y$, we choose $a = 0$ and $b = x$ and observe that $\delta xyab$ holds by δ_4.

If $x < y$, we have $x - \mathsf{S}y < x$ and complete induction gives us a and b such that $\delta(x - \mathsf{S}y)yab$. By δ_5 we now have $\delta xy(\mathsf{S}a)b$. ■

It turns out that a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\times\mathsf{N}}$ satisfies the relational specification δ if and only if it satisfies the **procedural specification** of repeated substraction:

$$\forall xy.\ f x y \;=\; \text{IF } \ulcorner x \le y \urcorner \text{ THEN } (0, x) \text{ ELSE LET } (a, b) = f\,(x - \mathsf{S}y)\,y \text{ IN } (\mathsf{S}a, b)$$

Fact 15.4.2 Every function satisfying the procedural specification satisfies δ.

Proof Let f satisfy the procedural specification. We show $\delta xy(\pi_1(fxy))(\pi_2(fxy))$ by complete induction on x. Following the procedural specification, we consider two cases.

1. $x \le y$. Then $fxy = (0, x)$ and the claim follows with δ_4.
2. $y < x$. Then we have $f(\mathsf{S}x - y)y = (a, b)$ and $fxy = (\mathsf{S}a, b)$. The claim $\delta xy(\mathsf{S}a, b)$ follows with δ_5 and the inductive hypothesis. ■

Fact 15.4.3 Every function satisfying δ satisfies the procedural specification.

Proof Let f satisfy δ and let U be the unfolding function for the procedural specification. Because of the uniqueness of δ (Fact 15.3.3) it suffices to show that Uf satisfies δ. We fix x and y and consider two cases following the procedural specification:

1. $x \le y$. Then we have to show $\delta xy0x$, which follows by δ_4.
2. $y < x$. Let $(a, b) = f(x - \mathsf{S}y)y$. Then we have to show $\delta xy(\mathsf{S}a)b$, which follows by δ_5 and $\delta(\mathsf{S}x - y)yab$. ■

Corollary 15.4.4 A function satisfies the relational specification δ if and only if it satisfies the procedural specification of repeated subtraction.

Proof Facts 15.4.2 and 15.4.3 . ∎

Corollary 15.4.5 All function satisfying the procedural specification of repeated subtraction agree.

Proof Facts 15.4.4 and 15.3.3. ∎

To be understandable, our proof of Fact 15.4.3 uses informal language and omits formal details. It takes considerable effort to verify the details of the proof by hand. In contrast, the Coq formulation of the proof is both rigorous and concise.

Exercise 15.4.6 (Uniqueness) Show the uniqueness of the following procedural specifications using complete induction.

a) $fxy = \text{IF } \ulcorner x \le y \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f\,(x - \mathsf{S}y)\,y)$

b) $fxy = \text{IF } \ulcorner x \le y \urcorner \text{ THEN } x \text{ ELSE } f\,(x - \mathsf{S}y)\,y$

c) $fxy = \text{IF } \ulcorner x \le y \urcorner \text{ THEN } (0, x) \text{ ELSE LET } (a, b) = f\,(x - \mathsf{S}y)\,y \text{ IN } (\mathsf{S}a, b)$

Exercise 15.4.7 Let $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ be a function satisfying

$$f\,x\,y \;:=\; \text{IF } \ulcorner x \le y \urcorner \text{ THEN } x \text{ ELSE } f\,(x - \mathsf{S}y)\,y$$

Prove the following properties of f using complete induction.

a) $\forall xy.\ fxy \le y$

b) $\forall xy.\ \Sigma k.\ x = k \cdot \mathsf{S}y + fxy$

Exercise 15.4.8 (Euclidean quotient)
We consider $\gamma\,xya := (a \cdot \mathsf{S}y \le x < \mathsf{S}a \cdot \mathsf{S}y)$.

a) Show that γ specifies the Euclidean quotient: $\gamma\,xya \longleftrightarrow \exists b.\ \delta xyab$.

b) Show that γ is unique: $\gamma xya \to \gamma xya' \to a = a'$.

c) Show that every function $f^{\mathsf{N} \to \mathsf{N} \to \mathsf{N}}$ satisfies

$$(\forall xy.\ \gamma\,xy(fxy)) \;\longleftrightarrow\; \forall xy.\ fxy = \text{IF } \ulcorner x \le y \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f(x - \mathsf{S}y)y)$$

d) Consider the function

$$\begin{aligned} &f : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &f0yb \;:=\; 0 \\ &f(\mathsf{S}x)yb \;:=\; \text{IF } \ulcorner b = y \urcorner \text{ THEN } \mathsf{S}(fxy0) \text{ ELSE } fxy(\mathsf{S}b) \end{aligned}$$

Show $\forall xy. \gamma\,xy(fxy0)$; that is, $fxy0$ is the Euclidean quotient of x and $\mathsf{S}y$.
Hint: For a proof the claim must be generalized.
Prove $b \le y \to \gamma\,(x + b)\,y\,(Dxyb)$.

Exercise 15.4.9 (Euclidean remainder)
Give defining equations for a function $f^{N \to N \to N \to N}$ such that $f x y 0$ computes the Euclidean *remainder* of x and $\mathsf{S}y$ by structural recursion on x. Prove the correctness of your function.

15.5 Summary

We studied two algorithms for Euclidean division, taking a relational specification for Euclidean division as starting point. Both algorithms yield certifying functions for Euclidean division. The naive algorithm employs structural recursion and can be immediately realized as a simply typed function. This is not the case for the repeated-subtraction algorithm. To obtain the certifying function, we use structural induction for the naive algorithm and complete induction for the repeated-subtraction algorithm. In each case we obtain proof obligations for the different cases considered by the algorithm. The proof obligations can be shown as lemmas and may be interpreted as derivations rules formulating the algorithms declaratively and without explicit recursion.

An important result is the uniqueness of the relational specification. Using uniqueness, we can show that a function satisfies the relational specification if and only if it satisfies the procedural specification for repeated subtraction. Using this result, we see that the function realizing the naive algorithm satisfies the procedural specification of repeated subtraction.

The proofs in this section often involve considerable formal detail, which is typical for program verifications. They are good examples of proofs whose construction and analysis profits much from working with a proof assistant. When done by hand, the amount of detail needed for rigorous proofs can be overwhelming. So one is forced to do the proofs informally omitting formal details, which is error-prone and requires considerable training to be reliable.

16 Least Witnesses

We will consider functions computing least witnesses for decidable predicates $p^{N \to \mathbb{P}}$. We study simply typed and certifying versions of such functions. Moreover, we show that a satisfiable predicate on numbers has a least witness if and only if the law of excluded middle holds.

The naive solution to the problem is a linear search, testing $p0, p1, \ldots, pk$ starting from 0 until a k satisfying p is found. Since linear search doesn't necessarily terminate, we will employ a search taking a step index limiting the available tests.

16.1 Least Witness Predicate

In this chapter, p will denote a predicate $N \to \mathbb{P}$ and n and k will denote numbers. We say that n is a **witness of p** if pn is provable, and that **p is satisfiable** if $\exists x.px$ is provable. 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 16.1.1

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 14.5.3) 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 $\neg pn$. ■

Exercise 16.1.2 (Euclidean Division) Prove the following equivalence:
$(x = a \cdot \mathsf{S}y + b \wedge b \leq y) \longleftrightarrow (\mathsf{least}\,(\lambda a.\ x < \mathsf{S}a \cdot \mathsf{S}y)\,a \wedge b = x - a \cdot \mathsf{S}y)$.

Exercise 16.1.3 (Subtraction) Prove that $x - y$ is the least z such that $x \leq y + z$:
$x - y = z \longleftrightarrow \mathsf{least}\,(\lambda z. x \leq y + z)\,z$.

Exercise 16.1.4 Prove $\mathsf{safe}\ p(\mathsf{S}n) \longleftrightarrow \mathsf{safe}\ pn \wedge \neg pn$.

16.2 Least Witness Functions

We assume a decidable predicate $p^{\mathsf{N}\to\mathbb{P}}$. We will consider functions that given n either return the least witness of p or the assurance that n is safe for p. More concretely, our functions will satisfy the **relational specification**

$$\delta xy \;:=\; (\mathsf{least}\,py \wedge y < x) \vee (\mathsf{safe}\,py \wedge y = x)$$

We say that f **satisfies** δ or that f **is a least witness function** if $\forall x.\,\delta x(fx)$. Given the specification and the restriction to structural recursion, we arrive at the following least witness function:

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

We say that G performs a **direct search**.

Fact 16.2.1 G is a least witness function.

Proof By induction on n. The successor case follows the case analysis of G. ■

It turns out that δ is functional. Thus all least witness functions agree with G.

Fact 16.2.2 δ is functional: $\delta xy \to \delta xy' \to y = y'$.

Proof We consider the four cases resulting from the disjunctions δxy and $\delta xy'$. One case follows with the uniqueness of least, the others are straightforward. ■

Fact 16.2.3 $pn \to x \geq n \to \delta xy \to \mathsf{least}\,py$.

Proof Straightforward considering the two cases specified by δxy. ■

Corollary 16.2.4 Every least witness function satisfies $px \to n \geq x \to \mathsf{least}\,p(fn)$.

Corollary 16.2.5 $px \to n \geq x \to \mathsf{least}\,p(Gn)$.

Proof Facts 16.2.1 and 16.2.4. ■

Fact 16.2.6 (Least witness operator)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\,\mathsf{dec}\,p \to \mathsf{sig}\,p \to \mathsf{sig}(\mathsf{least}\,p)$.

Proof Immediate from Fact 16.2.5. ■

Corollary 16.2.7 $\forall p^{N\to\mathbb{P}}.\ \mathsf{dec}\, p \to \mathsf{ex}\, p \to \mathsf{ex}(\mathsf{least}\, p)$.

Proof Given that we have to construct a proof, we can assume pn. This gives us $\mathsf{sig}\, p$ and thus we can obtain a least witness with Fact 16.2.6. ∎

Fact 16.2.8 (Decidability)
$\forall p^{N\to\mathbb{P}}.\ \mathsf{dec}\, p \to \mathsf{dec}\,(\mathsf{least}\, p)$.

Proof We show $\mathcal{D}(\mathsf{least}\, pn)$. If $\neg pn$, we have $\neg\mathsf{least}\, pn$. Otherwise we assume pn. Thus $\mathsf{least}\, p(Gn)$ by Fact 16.2.5. If $n = Gn$, we are done. If $n \neq Gn$, we assume $\mathsf{least}\, pn$ and obtain a contradiction with the uniqueness of $\mathsf{least}\, p$ (Fact 16.1.1). ∎

The elegant development presented here depends crucially on the use and the particular formulation of the relational specification δ. It took several iterations until we arrived at this design. Initially, we tried to construct a function G satisfying the statement of Corollary 16.2.5. This statement has the problem that inductive proofs will no go through.

Exercise 16.2.9 Assume a decidable predicate $p^{N\to\mathbb{P}}$. Construct certifying functions as follows:

a) $\forall x.\Sigma y.\ (\mathsf{least}\, py \wedge y < x) + (\mathsf{safe}\, py \wedge y = x)$

b) $\forall x.\ \mathsf{sig}(\mathsf{least}\, p) + \mathsf{safe}\, pn$

c) $\forall x.\ px \to \mathsf{sig}(\delta x)$

Follow the design of G but do not use the formal development for G.

Exercise 16.2.10 Assume a decidable predicate $p^{N\to\mathbb{P}}$. Give the defining equations for a function $D : N \to \mathcal{O}(N)$ such that

$$\forall n.\ \text{MATCH}\ Dn\ [\ {}^\circ x \Rightarrow \mathsf{least}\, p\, x \wedge x < n \mid \emptyset \Rightarrow \mathsf{safe}\, p\, n\ ]$$

Verify that your function satisfies this property.

Exercise 16.2.11 (Least witness functions)
Let $p^{N\to\mathbb{P}}$ be a decidable predicate and $f^{N\to N}$ be a least witness function for p. Prove the following:

a) $pn \to \mathsf{least}\, p\,(fn)$

b) $pm \to m \le n \to \mathsf{least}\, p(fn)$

c) $p(fn) \to \mathsf{least}\, p\,(fn)$

d) $\mathsf{least}\, p\, n \to fn = n$

e) $\mathsf{least}\, p\, n \to n \le m \to fm = n$

f) $fm \le m \to p(fm) \to m \le n \to fm = fn$

16.3 Step-Indexed Linear Search

The standard algorithm for computing least witnesses is **linear search**: One tests pk for $k = 0, 1, 2, \ldots$ until the first k satisfying p is found. Linear search terminates if and only if p has a witness. While linear search can be realized easily in a procedural programming language, realizing linear search with a terminating function in computational type theory requires a modification. The standard trick is to realize linear search with an extra argument called a *step index* limiting the number of search steps. We define a function Lnk which tests p starting from k for at most n steps:

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

Note that L recurses on the step index n.

We will show that $\lambda n.Ln0$ is a least witness function. For the inductive proof to go through, we need to consider Lnk for all safe k.

Fact 16.3.1 $\mathsf{safe}\,pk \to \delta(n + k)(Lnk)$.

Proof By induction on n with k quantified. In the successor case we follow the case analysis of L. ■

Note that the precondition $\mathsf{safe}\,pk$ serves as an invariant for the loop described by the tail-recursive function L.

Corollary 16.3.2 $\lambda n.Ln0$ is a least witness function.

Proof Fact 16.3.1 with $k = 0$. ■

Exercise 16.3.3 Write a certifying function

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

doing step-indexed linear search. Follow the design of L but do not use the formal development for L.

Exercise 16.3.4 Prove $\forall mnk.\ pm \to m \le n + k \to \mathsf{safe}\,p\,k \to \mathsf{least}\,p\,(Lnk)$.

Exercise 16.3.5 Prove $\forall n.\ Ln0 = Gn$

16.4 Least Witnesses and Excluded Middle

If we want a propositional least witness operator using $\exists$ in place of Σ, *logical decidability* of p suffices.

Lemma 16.4.1 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall k.\ pk \vee \neg pk) \to \forall n.\ \mathsf{ex}(\mathsf{least}\,p) \vee \mathsf{safe}\,pn.$

Proof By induction on n. ∎

Lemma 16.4.2 $\forall p^{\mathsf{N}\to\mathbb{P}}.\ (\forall k.\ pk \vee \neg pk) \to \mathsf{ex}\,p \to \mathsf{ex}(\mathsf{least}\,p).$

Proof Follows with Lemma 16.4.1. ∎

We can now show that the law of excluded middle

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

holds if and only if every satisfiable predicate on the numbers has a least witness.

Fact 16.4.3 $\mathsf{XM} \ \longleftrightarrow\ (\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{ex}\,p \to \mathsf{ex}(\mathsf{least}\,p)).$

Proof Direction $\to$ follows with Lemma 16.4.2. 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 $pn := \text{MATCH}\ n\ [\,0 \Rightarrow P \mid \mathsf{S}\_ \Rightarrow \top\,]$. 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 $p0$. ∎

Exercise 16.4.4 Prove that the following propositions are equivalent.

1. XM
2. $\forall pn.\ \mathsf{safe}\,pn \vee \mathsf{ex}(\mathsf{least}\,p)$
3. $\forall p.\ \mathsf{ex}\,p \to \mathsf{ex}(\mathsf{least}\,p).$

17 Size Recursion and Step-Indexing

Size recursion generalizes structural recursion such that recursion is possible for all smaller arguments, where smaller arguments are determined by a numeric size function. In contrast to structural recursion, where the argument must be from an inductive type, size recursion accommodates arguments from any type that has a numeric size function. The construction of a size recursion operator is straightforward; it generalizes the construction of a complete induction operator.

Size recursion provides us with a flexible induction principle for proofs. Proofs by induction on the size of objects are frequently used in mathematical developments.

Given a size recursion operator, we can construct certifying functions using recursion schemes expressible with size recursion. It may be convenient to accommodate such recursion schemes with a customized recursion operator incorporating the case analysis and encapsulating the termination proof.

Our main examples will be Euclidean division and greatest common divisors. We don't assume the presentation of Euclidean division in Chapter 15.

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. Interestingly, the construction of satisfying functions with step-indexing does not require size recursion.

17.1 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 14.9.1).

It turns out that a size recursion operator can be defined with structural recursion on numbers, following the idea we have seen before 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 17.1.1 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:

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

Note that the definition of the function R in the proof involves computational falsity elimination in the base case.

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 17.1.2 (Size Recursion Operator)

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

Proof Straightforward with Lemma 17.1.1. ∎

Often it is helpful to define specialized size recursion operators. We will often use a specialized size recursion operator for binary type functions.

Fact 17.1.3 (Binary size recursion operator)

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

Proof Size recursion on $X \times Y$ using the type function $\lambda a.\ p(\pi_1 a)(\pi_2 a)$ and the size function $\lambda a.\ \sigma(\pi_1 a)(\pi_2 a)$. ∎

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 17.1.4 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 17.1.5 Let f be a function $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ satisfying the following equation:

$$f x y = \begin{cases} x & \text{if } x \le y \\ f(x - \mathsf{S}y)y & \text{if } x > y \end{cases}$$

Prove the following using size recursion:

a) $\forall xy.\ fxy \le y$

b) $\forall xy \Sigma k.\ \ x = k \cdot \mathsf{S}y + fxy$

17.2 Euclidean Division

Euclidean division counts how often $\mathsf{S}y$ can be subtracted from x without truncation. We specify this operation with a **relational specification**

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

and say that a function $f^{\mathsf{N} \to \mathsf{N} \to \mathsf{N}}$ **satisfies δ** if $\forall xy.\ \delta xy(fxy)$. In addition, we fix a **procedural specification** of Euclidean division by repeated subtraction using the unfolding function

$$\Delta : \ (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$

$$\Delta f x y \ = \ \begin{cases} 0 & \text{if } x \le y \\ \mathsf{S}(f(x - \mathsf{S}y)y) & \text{if } x > y \end{cases}$$

We say that a function $f^{\mathsf{N} \to \mathsf{N} \to \mathsf{N}}$ **satisfies Δ** if f agrees with Δf (using formal language, $\forall xy.\ fxy = \Delta fxy$).

Given the formal definitions, we would like to prove:

1. f satisfies δ if and only if f satisfies Δ.
2. There is a function satisfying δ and satisfying Δ.
3. All functions satisfying δ or satisfying Δ agree.

Fact 17.2.1 All functions satisfying Δ agree.

Proof Let $\forall xy.\ fxy = \Delta fxy$ and $\forall xy.\ gxy = \Delta gxy$. We prove $fxy = gxy$ by size induction on x. By the assumptions it suffices to show $\Delta fxy = \Delta gxy$. Case analysis on $(x \le y) + (x > y)$. If $x \le y$, we have to show $0 = 0$. If $x > y$, we have to show $\mathsf{S}(f(x - \mathsf{S}y)y) = \mathsf{S}(g(x - \mathsf{S}y)y)$, which follows by the inductive hypothesis. ■

From the procedural specification one can obtain so-called derivation rules for the relational specification that must be satisfied for the correctness of the procedural specification.

Fact 17.2.2 (Derivation rules) δ satisfies the following rules:

- $\delta_1: \quad x \le y \to \delta x y 0$
- $\delta_2: \quad x > y \to \delta(x - \mathsf{S}y)yz \to \delta xy(\mathsf{S}z)$

Proof Straightforward. ■

Fact 17.2.3 (Functionality) $\forall xyzz'.\ \delta xyz \to \delta xyz' \to z = z'$.

Proof Straightforward. ■

Functionality is an important property of the relational specification δ. We also say that δ is **unique**.

Corollary 17.2.4 All functions satisfying δ agree.

From the procedural specification we also obtain a customized recursion operator.

Lemma 17.2.5 (Euclidean recursion operator)

$$\begin{aligned}
&\forall y^{\mathsf{N}}\, \forall p^{\mathsf{N}\to\mathbb{T}}.\\
&(\forall x.\ x \le y \to px) \to\\
&(\forall x.\ x > y \to p(x - \mathsf{S}y) \to px) \to\\
&\forall x.\ px
\end{aligned}$$

Proof By size recursion on x and case analysis on $(x \le y) + (x > y)$. ■

Note that the customized recursion operator incorporates the case analysis and the termination of the procedural specification.

It is now straightforward to construct a function satisfying δ.

Fact 17.2.6 $\forall xy\,\Sigma z.\ \delta xyz$.

Proof We construct this function using the Euclidean recursion operator, which gives us two subgoals for $x \le y$ and $x > y$. The first subgoal follows with δ_1. The second subgoal follows with δ_2 and the inductive hypothesis. ■

Using the Skolem correspondence, we now have a reducible function D satisfying δ. We can now prove equations like $D\,15\ 3 = 3$ by computational equality. Given the function Div from Fact 17.2.6, we can define a simply typed Euclidean division function

$$\begin{aligned}
\mathsf{div} &: \mathsf{N} \to \mathsf{N} \to \mathsf{N}\\
\mathsf{div}\ x\ 0 &:= 0\\
\mathsf{div}\ x\ (\mathsf{S}y) &:= \pi_1(\mathsf{Div}\ x\ y)
\end{aligned}$$

such that equations like $\mathsf{div}\ 133\ 12 = 11$ hold by computational equality.

Fact 17.2.7 Every function satisfying Δ satisfies δ.

Proof Let $\forall xy.\ fxy = \Delta fxy$. We show $\delta xy(fxy)$ by size recursion on x. By the assumption it suffices to show $\delta xy(\Delta fxy)$. Case analysis on $(x \le y) + (x > y)$, which yields the subgoals $\delta xy0$ and $\delta xy(\mathsf{S}(f(x-\mathsf{S}y))y)$. The first subgoal follows with δ_1. The second subgoal follows with δ_2 and the inductive hypothesis. ■

Fact 17.2.8 Every function satisfying δ satisfies Δ.

Proof Let f satisfy δ. We show $fxy = \Delta fxy$. By functionality of δ and the assumption it suffices to show $\delta xy(\Delta fxy)$. Case analysis on $(x \le y) + (x > y)$, which yields the subgoals $\delta xy0$ and $\delta xy(\mathsf{S}(f(x - \mathsf{S}y))y)$, which follow by δ_1 and δ_2. ■

Exercise 17.2.9 We specify a remainder function with a predicate

$$\rho xyz := (z \le x \land \exists k.\ x = k \cdot Sy + z)$$

a) Give an unfolding function R for a procedural specification obtaining remainders with repeated subtraction.
b) Give the derivation rules induced by R.
c) Convince yourself that the customized recursion operator for R agrees with the Euclidean recursion operator.
d) Establish the equivalence between ρ and R.
e) Construct a certifying function satisfying ρ using Euclidean recursion.
f) Show that all functions satisfying R agree.

Exercise 17.2.10
Prove $\forall xy^{\mathsf{N}}\ \Sigma ab^{\mathsf{N}}.\ (x = a \cdot \mathsf{S}y + b) \land (b \le y)$ using Euclidean recursion.

17.3 Greatest Common Divisors

Next we construct and verify a function computing greatest common divisors (GCDs) following the scheme we have used for Euclidean division. This time we work with an abstract relational specification to emphasize that only certain properties of the concrete relational specification are needed.

Definition 17.3.1 A **gcd relation** is a predicate $\gamma^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathbb{P}}$ satisfying the following conditions for all numbers x, y, z:

- γ_1 : $\gamma 0 y y$ (zero rule)
- γ_2 : $\gamma y x z \to \gamma x y z$ (symmetry rule)
- γ_3 : $x \le y \to \gamma x (y - x) z \to \gamma x y z$ (subtraction rule)

A **functional** gcd relation satisfies the additional condition

- γ_{fun}: $\gamma x y z \to \gamma x y z' \to z = z'$

We say that a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ **satisfies** a gcd relation γ if $\forall x y.\ \gamma x y (f x y)$.

A proposition $\gamma x y z$ may be read as saying that z is the GCD of x and y. We refer to the conditions γ_1, γ_2, and γ_3 as rules to highlight their computational interpretation:

- The GCD of x and 0 is x.
- The GCD of x and y is the GCD of y and x.
- The GCD of x and y is the GCD of x and $y - x$ if $x \le y$.

Fact 17.3.2 All functions satisfying a functional gcd relation agree.

Proof Straightforward. ∎

Definition 17.3.3
We fix a **procedural specification of GCDs** using the unfolding function

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

We say that a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ **satisfies Γ** if $\forall x y.\ f x y = \Gamma f x y$.

Given a functional gcd relation γ, we will prove:

1. f satisfies γ if and only if f satisfies Γ.
2. There is a function satisfying γ and satisfying Γ.
3. All functions satisfying γ or satisfying Γ agree.

Fact 17.3.4 All functions satisfying Γ agree.

Proof Let $\forall xy.\ fxy = \Gamma fxy$ and $\forall xy.\ gxy = \Gamma gxy$. We prove $fxy = gxy$ by size induction on $x+y$. By the assumptions it suffices to show $\Gamma fxy = \Gamma gxy$. Since the base cases follow by computational equality, it suffices to show $\Gamma f(\mathsf{S}x)(\mathsf{S}y) = \Gamma g(\mathsf{S}x)(\mathsf{S}y)$. Case analysis on $(x \le y) + (x > y)$. If $x \le y$, we have to show $f(\mathsf{S}x)(y - x) = g(\mathsf{S}x)(y - x)$, which follows by the inductive hypothesis. The other case is symmetric. ∎

Note that the proof explicates that the procedural specification "terminates" since the sum $x + y$ of the arguments x and y is decreased upon "recursion".

We identify a customized recursion operator for GCD computation. This time we base the customized recursion operator on the abstract derivation rules coming with the definition of gcd relations.

Lemma 17.3.5 (GCD recursion operator)

$$\begin{array}{l} \forall p^{\mathsf{N} \to \mathsf{N} \to \mathbb{T}}. \\ (\forall y.\ p0y) \to \\ (\forall xy.\ pxy \to pyx) \to \\ (\forall xy.\ x \le y \to px(y-x) \to pxy) \to \\ \forall xy.\ pxy \end{array}$$

Proof By binary size recursion on $x + y$ considering four disjoint cases: $x = 0$, $y = 0$, $x \le y$, and $y < x$. ∎

Fact 17.3.6 Let γ be a gcd relation. Then $\forall xy\, \Sigma z.\ \gamma xyz$.

Proof We construct the function using the gcd recursion operator, which gives us three subgoals. The first subgoal follows with γ_1. The second subgoal follows with γ_2. The third subgoal follows with γ_3 and the inductive hypothesis. ∎

Using the Skolem correspondence, we now have a reducible function G satisfying Γ. We can now prove equations like $G\,49\,63 = 7$ by computational equality.

Fact 17.3.7 Every function satisfying Γ satisfies every gcd relation.

Proof Let $\forall xy.\ fxy = \Gamma fxy$ and let γ be a gcd relation. We show $\gamma xy(fxy)$ using size recursion on $x + y$. By the assumption it suffices to show $\gamma xy(\Gamma fxy)$. We consider four disjoint cases: $x = 0$, $y = 0$, $x \le y$, and $y < x$. The base cases follow by γ_1 and γ_2. The remaining cases follow by γ_2 and γ_3 and the inductive hypothesis. ∎

Fact 17.3.8 Every function satisfying a functional gcd relation satisfies Γ.

Proof Let γ be a functional gcd relation and let f satisfy γ. We show $fxy = \Gamma fxy$. By the functionality of γ and the assumption it suffices to show $\gamma xy(\Gamma fxy)$. We consider four disjoint cases: $x = 0$, $y = 0$, $x \leq y$, and $y < x$. The base cases follow with γ_1 and γ_2. The remaining cases follow with γ_2 and γ_3. ■

Definition 17.3.9 (Concrete gcd relation) We define the **divisors** of a number and the concrete gcd relation as follows:

$$n \mid x \;:=\; \exists k.\; x = k \cdot n \qquad\qquad n \textbf{ divides } x$$
$$\gamma xyz \;:=\; \forall n.\; n \mid z \;\longleftrightarrow\; n \mid x \wedge n \mid y$$

We will show that γ is a functional gcd relation. We start with the relevant facts about divisibility.

Fact 17.3.10

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

Proof Claims 1–4 have straightforward proofs unfolding the definition of divisibility. For (5), we consider $y = 0$ and $y > 0$. For $y = 0$, we obtain $x = 0$ by (4) with $n := Sx$ and (1). For $y > 0$, we obtain $x \leq y$ by (3) and (1). ■

Fact 17.3.11 The concrete gcd relation is a functional gcd relation.

Proof Condition γ_1 follows with Fact 17.3.10 (1). Condition γ_2 is obvious from the definition. Condition γ_3 follows with Fact 17.3.10 (2). The functionality of γ follows with Fact 17.3.10 (5) and antisymmetry. ■

Fact 17.3.12 $g^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfies Γ if and only if $\forall xyn.\; n \mid gxy \longleftrightarrow n \mid x \wedge n \mid y$.

Proof Facts 17.3.7, 17.3.11, and 17.3.8. ■

Exercise 17.3.13 (GCDs with modulo operation)
Assume G and M are functions $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ satisfying the equations

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

You will show that G computes GCDs.

Let γ be a gcd relation. Prove the following claims.

a) $Mxy \le y$.

b) $\gamma(Myx)(Sx)z \to \gamma(Sx)yz$.

c) G satisfies γ.

d) $\forall xyn.\ n \mid Gxy \longleftrightarrow n \mid x \wedge n \mid y$.

e) A function satisfies Γ if and only if it satisfies the equations for G.

Hints: Claim (a) follows by size induction on x with y fixed. Claim (b) follows by size induction on y with x fixed. Claim (c) follows by size induction on x with y quantified using (b) and (a). Claim (d) follows from (c) and Fact17.3.11. Claim (e) follows with (d) and Facts 17.3.12 and 17.3.11.

Exercise 17.3.14 Prove the following facts about functional gcd relations.

a) All functional gcd relations agree.

b) If f satisfies Γ, then $\lambda xyz.\ fxy = z$ is a functional gcd relation.

c) A functional gcd relation exists if and only if a function satisfying Γ exists.

Use Facts 17.3.6, 17.3.7, and 17.3.8. Do not use the concrete gcd relation (i.e., Fact 17.3.11). Note that the above facts give us two abstract characterizations of GCDs: Either as a functional gcd relation or as a function satisfying the procedural specification Γ.

17.4 Step-Indexed Function Construction

Using size recursion, we could show with routine proofs that the procedural specifications of Euclidean division and greatest common divisors have unique solutions. Using size-recursion, we also could construct functions satisfying the procedural specifications using a complementary arithmetic specification. In both cases the functionality of the arithmetic specification was essential. We will now introduce a technique called *step indexing* providing for the direct construction of functions satisfying procedural specifications. Step indexing doesn't require an arithmetic specification and works whenever the termination of the procedural specification can be argued with an arithmetic size function. Moreover, step indexing doesn't require size recursion.

Suppose we have a procedural specification whose termination can be argued with an arithmetic size function. Then we can define an auxiliary function taking the size (a number) as an additional argument called *step index* and arrange things such that the recursion is structural recursion on the step index. We obtain the specified function by using the auxiliary function with a sufficiently large step index.

We demonstrate the technique with the procedural specification of GCDs

(Definition 17.3.3). Here the step-indexed auxiliary function comes out as follows:

$$\begin{aligned} G\,0\,x\,y &:= 0 \\ G\,(\mathsf{S}n)\,x\,y &:= \Gamma\,(Gn)\,x\,y \end{aligned}$$

The essential result about G is *index independence*: $Gnxy = Gn'xy$ whenever the step indices are large enough.

Lemma 17.4.1 (Index independence)
$\forall nn'xy.\ (n > x + y) \to (n' > x + y) \to Gnxy = Gn'xy.$

Proof By induction on n with n', x, and y quantified. The base case has a contradictory assumption. In the successor case, we destructure n'. The case $n' = 0$ has a contradictory assumption. If $n = \mathsf{S}n_1$ and $n' = \mathsf{S}n_1'$, we have $\Gamma\,(Gn_1)xy = \Gamma\,(Gn_1')xy$. We destructure x. The base case holds by computational equality. Next we destructure y, where the base case again holds by computational equality. The claim now follows by case analysis on $(x' \le y') + (x' > y')$ using the inductive hypothesis. ∎

Fact 17.4.2 $\lambda xy.\,G(\mathsf{S}(x+y))xy$ satisfies Γ.

Proof Let $g := \lambda xy.\,G(\mathsf{S}(x+y))xy$. We show $G(\mathsf{S}(x+y))xy = \Gamma gxy$. For $x = 0$ and $x = \mathsf{S}x' \land y = 0$ the claim holds by computational equality. It remains to show $G(\mathsf{S}(\mathsf{S}x' + \mathsf{S}y'))(\mathsf{S}x')(\mathsf{S}y') = \Gamma g(\mathsf{S}x')(\mathsf{S}y')$. The claim now follows by case analysis on $(x' \le y') + (x' > y')$ using index independence (Lemma 17.4.1). ∎

Exercise 17.4.3 (Euclidean division) Construct a function satisfying the procedural specification of Euclidean division in §17.2 using step indexing.

Exercise 17.4.4 (Fibonacci)
Recall the procedural specification of the Fibonacci function in Figure 1.5.

a) Show that all functions satisfying the procedural specification agree.
Hint: Use size induction.

b) Construct and verify a function satisfying the procedural specification using step indexing.

Exercise 17.4.5 (GCDs with modulo operation)
Consider procedural specifications for functions $M, G : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ as follows:

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

a) Define the unfolding functions for M and G.

b) Show that the procedural specifications are unique using size recursion.
c) Construct and verify functions M and G satisfying the procedural specifications using step indexing.

Exercise 17.4.6 Nonterminating procedural specifications may be unsatisfiable or may have disagreeing solutions.

a) Give a function $F^{(N\to N)\to N\to N}$ such that $\forall f^{N\to N}.\ \neg\forall x.\ fx = Ffx$.
b) Give a function $F^{(N\to N)\to N\to N}$ and functions $f^{N\to N}$ and $g^{N\to N}$ satisfying F such that $fx \neq gx$ for all x.
c) Convince yourself that unsatisfiable procedural specifications are unique.

We conjecture that all terminating procedural specifications are satisfiable and unique. Note that the notion of termination is informal.

17.5 Summary

In this chapter we have studied terminating procedural specifications and their relationship with relational specifications. We considered procedural specifications whose termination can be argued with an arithmetic size function. For the examples we considered, we made the following observations:

1. Functions satisfying procedural specifications can be constructed with step-indexing and structural recursion making use of the size function certifying termination.
2. Index independence of step-indexed functions follows with structural induction on the step index.
3. Correctness of a satisfying function obtained with a step-indexed function follows with index independence.
4. Uniqueness of procedural specifications follows with size recursion.
5. That a function satisfying a procedural specification satisfies a relational specification can be shown with size recursion.
6. Given a function f satisfying a unique relational specification δ, f satisfies a procedural specification Γ if Γf satisfies δ.
7. From the procedural specification and the size function one may obtain a customised recursion operator with size recursion.
8. From the procedural specification one may obtain derivation rules for relational specifications such that the procedural specification is correct for every relational specification satisfying the rules.

The relational specifications we considered were all unique (i.e., functional). This has the consequence that a function satisfies the relational specification if and only if it satisfies the concomitant procedural specification. Moreover, the relational specification can be characterized as a functional relation satisfying the derivation rules obtained from the procedural specification (see Exercise 17.3.14).

The DNF solver appearing in §24.4 is an interesting example for the use of a customized recursion operator (DNF recursion §24.5).

It is common to say that a function f satisfying a procedural specification Φ is a *fixed point of* Φ. In fact, by definition f satisfies Φ if and only if the functions Φf and f agree for all arguments. This gives us $\Phi f = f$ if we assume function extensionality.

18 Existential Witness Operators

In this chapter, we will define an existential witness operator (EWO)

$$W : \quad \forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{dec}\, p \to \mathsf{ex}\, p \to \mathsf{sig}\, p$$

that for decidable predicates on numbers obtains a satisfying number from a satisfiability proof. The interesting point about an existential witness operator is that it obtains a computational witness from a propositional satisfiability proof.

EWOs are required for various computational constructions. They exist for all types that embed into the type of numbers.

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 of 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 $\exists n.pn$.

18.1 EWO Basics

We define the type of EWOs for a type X as follows:

$$\mathsf{EWO}\, X^{\mathbb{T}} \ := \ \forall p^{X\to\mathbb{P}}.\ \mathsf{dec}\, p \to (\exists x.px) \to (\Sigma x.px)$$

Fact 18.1.1 The types $\bot$, $\top$, and B have EWOs.

Proof In all three cases computational falsity elimination is essential. For $\bot$ an EWO is trivial since it is given an element of $\bot$. For $\top$ and B we can check p for all elements and obtain a contradiction if p holds for no element. ■

Fact 18.1.2 (Disjunctive EWOs)
Let p and q be decidable predicates on a type with an EWO.
Then there is a function $(\mathsf{ex}\, p \vee \mathsf{ex}\, q) \to (\mathsf{sig}\, p + \mathsf{sig}\, q)$.

Proof Use the EWO for X with the predicate $\lambda x.px \vee qx$. ■

Next we show that all numeral types $\mathcal{O}^n(\bot)$ have EWOs. The key insight for this result is that EWOs transport from X to $\mathcal{O}(X)$. It turns out that EWOs also transport backwards from $\mathcal{O}(X)$ to X.

Fact 18.1.3 (Option types transport EWOs)
$\mathsf{EWO}\,(\mathcal{O}(X)) \Leftrightarrow \mathsf{EWO}\, X$.

Proof Suppose p is a decidable and satisfiable predicate on X. Then

$$\lambda a.\ \text{MATCH}\ a\ [\ {}^\circ x \Rightarrow px \mid \emptyset \Rightarrow \bot\ ]$$

is a decidable and satisfiable predicate on $\mathcal{O}(X)$. The EWO for $\mathcal{O}(X)$ gives us ${}^\circ x$ such that px.

For the other direction, suppose p is a decidable and satisfiable predicate on $\mathcal{O}(X)$. Then $\lambda x.p({}^\circ x)$ is a decidable and satisfiable predicate on X. The EWO for X gives x with $p({}^\circ x)$. ■

Corollary 18.1.4 (Numeral types have EWOs)
The numeral types $\mathcal{O}^n(\bot)$ have EWOs.

Proof By induction on n using Facts 18.1.1 and 18.1.3. ■

It turns out that injections transport EWOs backwards. This result will be useful once we have shown that N has an EWO.

Fact 18.1.5 (Injections transport EWOs)
$\mathcal{I}XY \to \mathsf{EWO}\, Y \to \mathsf{EWO}\, X$.

Proof Let $\mathsf{inv}_{XY} f g$. To show that there is an EWO for X, we assume a decidable and satisfiable predicate $p^{X \to \mathbb{P}}$. Then $\lambda y.p(gy)$ is a decidable and satisfiable predicate on Y. The EWO for Y now gives us a y such that $p(gy)$. ■

Fact 18.1.6 (Co-inverse for surjective function)
Let $f^{X \to Y}$ be a surjective function with $\mathsf{EWO}\ X$ and $\mathcal{E}\, Y$. Then $\Sigma g.\, \mathsf{inv}\, f g$.

Proof It suffices to construct a function $\forall y.\Sigma x.\, fx = y$. We fix y and use the EWO for X to obtain x with $fx = y$. This works since f is surjective and equality on Y is decidable. ■

Fact 18.1.7 (Inverse for bijective function)
Let $f^{X \to Y}$ be a bijective function with $\mathsf{EWO}\ X$ and $\mathcal{E}\,Y$. Then $\Sigma g.\ \mathsf{inv}\,gf \wedge \mathsf{inv}\,fg$.

Proof Fact 18.1.6 gives us g with $\mathsf{inv}\,fg$. Now $\mathsf{inv}\,gf$ follows since f is injective (Fact 10.6.1). ∎

18.2 Linear Search Types

We now approach the construction of an EWO for N. The technique we have used for $\bot$, $\top$, and B does not apply since N has infinitely many elements. We now need to do a linear search $n = 0, 1, 2, \ldots$ until pn holds. We will be able to do this search with a computational predicate Tpn such that $pn \to Tpn$ and Tpn provides for a computational recursion $k = n, \mathsf{S}n, \mathsf{SS}n, \ldots$ until pk holds.

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 an EWO (existential 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 18.2.1 (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: Assume the definition gives you the constants

$$\begin{aligned} B &: \mathbb{T} \\ C &: (B \to \bot) \to B \\ M &: \forall Z.\, B \to ((B \to \bot) \to Z) \to Z \end{aligned}$$

First define a function $f : B \to \bot$ using the matching constant M.

Exercise 18.2.2 With linear search types we can express an empty propositional type allowing for computational elimination:

$$V : \mathbb{P} \;::=\; T(\lambda n.\bot)0$$

Define a function $V \to \forall X^{\mathbb{T}}.\, X$.

Exercise 18.2.3 Higher-order recursion offers yet another possibility for defining an empty propositional type allowing for computational elimination:

$$V : \mathbb{P} \;::=\; C(\top \to V)$$

Define a function $V \to \forall X^{\mathbb{T}}.\, X$.

18.3 EWO for the Type of Numbers

We now assume that p is a decidable predicate on numbers. We will define an EWO

$$W : \mathsf{ex}\, p \to \mathsf{sig}\, p$$

from two functions

$$\begin{aligned} W' &: \;\forall n.\, Tn \to \mathsf{sig}\, p \\ V &: \;\forall n.\, pn \to T0 \end{aligned}$$

The idea is to first obtain a derivation $d : T0$ using V and the witness of the proof of $\mathsf{ex}\, p$, and then obtain a computational witness using W' and the derivation $d : T0$.

We define W' by recursion on Tn:

$$\begin{aligned} &W' : \forall n.\, Tn \to \mathsf{sig}\, p \\ &W'\, n\, (C\varphi) \;:=\; \begin{cases} (n, h) & \text{if } h : pn \\ W'\, (\mathsf{S}n)\, (\varphi h) & \text{if } h : \neg pn \end{cases} \end{aligned}$$

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

$$\forall n.\ pn \to Tn \tag{18.1}$$
$$\forall n.\ T(\mathsf{S}n) \to Tn \tag{18.2}$$

Using recursion on n, function (18.2) yields a function

$$\forall n.\ Tn \to T0 \tag{18.3}$$

Using function (18.1), we have a function $V : \forall n.\ pn \to T0$ as required.

Theorem 18.3.1 (EWO for N) The type N has an EWO.

Proof Using V we obtain a derivation $d : T0$ from the witness of the proof of $\mathsf{ex}\,p$. There is no problem with the elimination restriction since $T0$ is a proposition. Now W' yields a computational witness for p. ∎

Corollary 18.3.2 Every type that embeds into N has an EWO: $\mathcal{I}X\mathsf{N} \to \mathsf{EWO}\,X$.

Proof Theorem 18.3.1 and Fact 18.1.5. ∎

Fact 18.3.3 $\mathsf{N} \times \mathsf{N}$ has an EWO.

Proof Follows with Fact 18.3.2 since $\mathsf{N} \times \mathsf{N}$ and N are in bijection (Fact 10.6.9). ∎

Corollary 18.3.4 (Binary EWO)
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 the EWO for $\mathsf{N} \times \mathsf{N}$ and the predicate $\lambda a.\ p(\pi_1 a)(\pi_2 a)$. ∎

Exercise 18.3.5 Point out where in the defining equation of W' it is exploited that linear search types are computational (i.e., no elimination restriction).

Exercise 18.3.6 Verify that the definition of W does not require computational falsity elimination.

Exercise 18.3.7 Let p be a decidable predicate on numbers. Define a function $\forall n.\ Tn \to \Sigma k.\ k \geq n \wedge pk$ by recursion on Tn.

18.4 More EWOs

Fact 18.4.1 (Existential least witness operator)
There is a function $\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{dec}\,p \to \mathsf{ex}\,p \to \mathsf{sig}(\mathsf{least}\,p)$.

Proof There are two ways to construct the operator using W. Either we use Fact 16.2.6 that gives us a least witness for a witness, or Fact 16.2.8 and Corollary 16.2.7 that tell us that $\mathsf{least}\,p$ is a decidable and satisfiable predicate. ∎

The existential least witness operator is obviously extensional; that is, the witness computed does not depend on the decider and does not change if we switch to an equivalent predicate.

Fact 18.4.2 (Extensional EWO) Assume $W : \forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{dec}\,p \to \mathsf{ex}\,p \to \mathsf{sig}(\mathsf{least}\,p)$ and two predicates $p, p' : \mathsf{N} \to \mathbb{P}$ with deciders d and d' and satisfiability proofs $h : \mathsf{ex}\,p$ and $h' : \mathsf{ex}\,p'$. Then $\pi_1(Wpdh) = \pi_1(Wp'd'h')$ whenever $\forall n.\ pn \longleftrightarrow p'n$.

Proof Straightforward. Exercise. ∎

The following fact was discovered by Andrej Dudenhefner in March 2020.

Fact 18.4.3 (Discreteness via step-indexed boolean 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 EWO for numbers 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 18.4.4 (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 18.4.5 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 18.4.6 Construct an EWO $\mathsf{ex}\,p \to \mathsf{sig}\,p$ for decidable predicates p on booleans. Exploit that there are only two a priori known candidates for a witness. Note that computational elimination on $\bot$ is needed.

Exercise 18.4.7 (EWO for boolean tests)
A basic EWO applies to boolean tests:

$$\forall f^{\mathsf{N}\to\mathsf{B}}.\ (\exists n.\ fn = \mathbf{T}) \to (\Sigma n.\ fn = \mathbf{T})$$

a) Define an EWO for boolean tests
   i) using the EWO W.
   ii) from scratch using customized linear search types.
b) Using an EWO for boolean tests, define
   i) an EWO for decidable predicates on numbers.
   ii) an EWO

$$\forall X^{\mathbb{T}}\ \forall p^{X\to\mathbb{P}}.\ \mathsf{sig}(\mathsf{enum}\, p) \to \mathsf{ex}\, p \to \mathsf{sig}\, p$$

   for *enumerable* predicates

$$\mathsf{enum}\ p^{X\to\mathbb{P}}\ f^{\mathsf{N}\to\mathcal{O}(X)}\ :=\ \forall x.\ px \longleftrightarrow \exists n.\ fn = {}^{\circ}x$$

18.5 Eliminator and Existential Characterization

We define an eliminator for linear search types:

$$E_T : \forall q^{\mathsf{N}\to\mathbb{T}}.\ (\forall n.\ (\neg pn \to q(\mathsf{S}n)) \to qn) \to \forall n.\ Tn \to qn$$
$$E_T\, q\, f\, n(C\varphi)\ :=\ fn(\lambda h.\, E_T\, q\, f\, (\mathsf{S}n)(\varphi h))$$

The eliminator may be understood as a generalization of the definition of W'. In fact, W' and

$$E_T\ (\lambda n.\mathsf{sig}p)\ (\lambda nf.\textsc{match}\ulcorner pn\urcorner[\,\mathsf{L}\,h \Rightarrow (n,h)\mid \mathsf{R}\,h \Rightarrow fh\,])$$

are equal up to computational equality.

The eliminator E_T provides for induction on derivation of Tn. 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 18.5.1 Define W' with the eliminator E_T for T.

Exercise 18.5.2 (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 18.5.3 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.

18.6 Notes

We proved transportation lemmas for EWOs and established EWOS on finite types and the type of numbers using linear search types. For simplicity we did not require extensionality for EWOs, but all transportation lemmas and all concrete EWOS we constructed can be also obtained for extensional EWOs. See Fact 18.4.2 for extensionality of EWOs.

With linear search types we have seen computational propositions that go beyond of 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 recursion terminates is a basic design feature of computational type theories.

19 Lists

Finite sequences $[x_1, \dots, x_n]$ are omnipresent in mathematics and computer science, appearing with different interpretations and notations, for instance, as vectors, strings, or states of stacks and queues. In this chapter, we study inductive list types providing a recursive representation for finite sequences whose elements are taken from a base type. Besides numbers, lists are the most important recursive data type in computational type theory. Lists have much in common with numbers, given that recursion and induction are linear for both data structures. Lists also have much in common with finite sets, given that both 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.

19.1 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, $x_1, \dots, x_n$ the **elements** of the list, and the numbers $0, \dots, n-1$ the **positions** of the list. An element may appear at more than one position in a list. For instance, $[2, 2, 3]$ is a list of length 3 that has 2 elements, where the element 2 appears at positions 0 and 1.

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 **universal eliminator for lists** as follows:

$$\begin{aligned} \mathsf{E}_{\mathcal{L}} :\ & \forall X^{\mathbb{T}}\, p^{\mathcal{L}(X) \to \mathbb{T}}.\ p\,[] \to (\forall x A.\ pA \to p(x :: A)) \to \forall A.\, pA \\ \mathsf{E}_{\mathcal{L}}\, X p e_1 e_2\, [] &:= e_1 \\ \mathsf{E}_{\mathcal{L}}\, X p e_1 e_2\, (x :: A) &:= e_2 x A (\mathsf{E}_{\mathcal{L}}\, X p e_1 e_2 A) \end{aligned}$$

The eliminator provides for inductive proofs, recursive function definitions, and structural case analysis.

Fact 19.1.1 (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 19.1.2 (Decidable Equality) 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 19.1.1. In case both lists are nonempty with heads x and y, an additional case analysis on $x = y$ is needed. ∎

Exercise 19.1.3 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ \mathcal{D}(A = [])$.

Exercise 19.1.4 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ (A = []) + \Sigma x B.\ A = x :: B$.

19.2 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:

$$\begin{aligned} \mathsf{len}\,[x_1,\dots,x_n] &= n && \textbf{length} \\ [x_1,\dots,x_m] +\!\!+ [y_1,\dots,y_n] &= [x_1,\dots,x_m,y_1,\dots,y_n] && \textbf{concatenation} \\ f@[x_1,\dots,x_n] &= [f@x_1,\dots,f@x_n] && \textbf{map} \end{aligned}$$

Formally, we define the operations as recursive functions:

$$\begin{aligned} &\mathsf{len} : \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathsf{N} \\ &\quad \mathsf{len}\,[] := 0 \\ &\quad \mathsf{len}\,(x :: A) := S\,(\mathsf{len}\,A) \end{aligned}$$

$$\begin{aligned} &+\!\!+ : \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathcal{L}(X) \to \mathcal{L}(X) \\ &\quad [] +\!\!+ B := B \\ &\quad (x :: A) +\!\!+ B := x :: (A +\!\!+ B) \end{aligned}$$

$$\begin{aligned} &@ : \forall XY^{\mathbb{T}}.\ (X \to Y) \to \mathcal{L}(X) \to \mathcal{L}(Y) \\ &\quad f@[] := [] \\ &\quad f@(x :: A) := fx :: (f@A) \end{aligned}$$

Note that we accommodate X and Y as implicit arguments for readability.

Fact 19.2.1

1. $A +\!\!+ (B +\!\!+ C) = (A +\!\!+ B) +\!\!+ C$ (associativity)
2. $A +\!\!+ [] = A$
3. $\mathsf{len}\,(A +\!\!+ B) = \mathsf{len}\,A + \mathsf{len}\,B$
4. $\mathsf{len}\,(f@A) = \mathsf{len}\,A$
5. $\mathsf{len}\,A = 0 \longleftrightarrow A = []$

Proof The equations follow by induction on A. The equivalence follows by case analysis on A. ■

19.3 Membership

Informally, we may characterize **membership** in lists with the equivalence

$$x \in [x_1, \dots, x_n] \longleftrightarrow x = x_1 \vee \cdots \vee x = x_n \vee \bot$$

Formally, we define the **membership predicate** by structural recursion on lists:

$$\begin{aligned} (\in) :\ & \forall X^{\mathbb{T}}.\, X \to \mathcal{L}(X) \to \mathbb{P} \\ (x \in []) &:= \bot \\ (x \in y :: A) &:= (x = y \vee x \in A) \end{aligned}$$

We treat the type argument X of the membership predicate as implicit argument. If $x \in A$, we say that x is an **element** of A.

Fact 19.3.1 (Existential Characterization) $x \in A \longleftrightarrow \exists A_1 A_2.\ A = A_1 + x :: A_2$.

Proof Direction $\to$ follows by induction on A. The nil case is contradictory. In the cons case a case analysis on $x \in a :: A'$ closes the proof with the inductive hypothesis.

Direction $\leftarrow$ follows by induction on A_1. ■

Fact 19.3.2 $\forall x a^X \forall A^{\mathcal{L}(X)}.\ \mathcal{E}(X) \to x \in a :: A \to (x = a) + (x \in A)$.

Proof Straightforward. ■

Fact 19.3.3 (Factorization) $\forall x^X A^{\mathcal{L}(X)}.\ \mathcal{E}(X) \to x \in A \to \Sigma A_1 A_2.\ A = A_1 + x :: A_2$.

Proof By induction on A. The nil case is contradictory. In the cons case a case analysis using Fact 19.3.2 closes the proof. ■

Fact 19.3.4 (Decidable Membership) $\forall x^X \forall A^{\mathcal{L}(X)}.\ \mathcal{E}(X) \to \mathcal{D}(x \in A)$.

Proof By induction on A. ■

Fact 19.3.5 (Membership laws)

1. $x \in A + B \longleftrightarrow x \in A \vee x \in B$.
2. $x \in f@A \longleftrightarrow \exists a.\ a \in A \wedge x = fa$.

Proof By induction on A. ■

Fact 19.3.6 (Injective map)
injective $f \to fx \in f@A \to x \in A$.

Proof Follows with 19.3.5(2). ∎

Recall that bounded quantification over numbers preserves decidability (Fact 14.8.1). Similarly, quantification over the elements of a list preserves decidability. In fact, quantification over the elements of a list is a form of bounded quantification.

We will use the notations

$$\begin{aligned}
\forall x{\in}A.\ px &:= \forall x.\ x \in A \to px \\
\exists x{\in}A.\ px &:= \exists x.\ x \in A \land px \\
\Sigma x{\in}A.\ px &:= \Sigma x.\ x \in A \times px
\end{aligned}$$

for quantifications over the elements of a list.

Fact 19.3.7 (Bounded Quantification) Let $p^{X \to \mathbb{T}}$ be a decidable type function. Then there are decision functions as follows:

1. $\forall A.\ (\Sigma x{\in}A.\ px) + (\forall x{\in}A.\ \neg px)$
2. $\forall A.\ \mathcal{D}(\forall x{\in}A.\ px)$
3. $\forall A.\ \mathcal{D}(\Sigma x{\in}A.\ px)$

Proof By induction on A. ∎

Exercise 19.3.8
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 19.3.9 (EWO) Let p be a decidable predicate on a type X.
Construct a function $\forall A.\ (\exists x \in A.\ px) \to (\Sigma x \in A.\ px)$.

19.4 Inclusion and 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 computational type theory is that sets in general are noncomputational objects.

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 19.4.1 List inclusion $A \subseteq B$ is reflexive and transitive. List equivalence $A \equiv B$ is reflexive, symmetric, and transitive.

Fact 19.4.2 We have the following properties for membership, inclusion, and equivalence of lists.

$x \notin []$	$x \in [y] \longleftrightarrow x = y$
$[] \subseteq A$	$A \subseteq [] \to A = []$
$x \in y :: A \to x \neq y \to x \in A$	$x \notin y :: A \to x \neq y \wedge x \notin A$
$A \subseteq B \to x \in A \to x \in B$	$A \equiv B \to x \in A \longleftrightarrow x \in B$
$A \subseteq B \to x :: A \subseteq x :: B$	$A \equiv B \to x :: A \equiv x :: B$
$A \subseteq B \to A \subseteq x :: B$	$x :: A \subseteq B \longleftrightarrow x \in B \wedge A \subseteq B$
$x :: A \subseteq x :: B \to x \notin A \to A \subseteq B$	$x :: A \subseteq [y] \longleftrightarrow x = y \wedge A \subseteq [y]$
$x :: A \equiv x :: x :: A$	$x :: y :: A \equiv y :: x :: A$
$x \in A \to A \equiv x :: A$	
$x \in A + B \;\longleftrightarrow\; x \in A \vee x \in B$	
$A \subseteq A' \to B \subseteq B' \to A + B \subseteq A' + B'$	$A + B \subseteq C \;\longleftrightarrow\; A \subseteq C \wedge B \subseteq C$

Proof Except for the membership fact for concatenation, which already appeared as Fact 19.3.5, all claims have straightforward proofs not using induction. ∎

Fact 19.4.3 (Rearrangement)
$x \in A \to \exists B.\; A \equiv x :: B \wedge \mathsf{len}\, A = \mathsf{len}(x :: B)$.

Proof Follows with Fact 19.3.1. There is also a direct proof by induction on A. ∎

Fact 19.4.4 (Rearrangement)
$\mathcal{E}(X) \to \forall x^X.\; x \in A \to \Sigma B.\; A \equiv x :: B \wedge \mathsf{len}\, A = \mathsf{len}(x :: B)$.

Proof Follows with Fact 19.3.3. There is also a direct proof by induction on A using Fact 19.3.2. ∎

Fact 19.4.5 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 19.3.4) and bounded quantification preserves decidability (Fact 19.3.7). ■

19.5 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:

$$\begin{aligned} \mathsf{rep} :\ & \mathcal{L}(X) \to \mathbb{P} \\ \mathsf{rep}\,[\,] \ &:=\ \bot \\ \mathsf{rep}\,(x :: A) \ &:=\ x \in A \lor \mathsf{rep}\,A \end{aligned}$$

Fact 19.5.1 (Characterization)
For every list A over a discrete type we have:
$\mathsf{rep}\,A \ \longleftrightarrow\ \exists x A_1 A_2.\ A = A_1 +\!\!+\, x :: A_2 \land x \in A_2$.

Proof By induction on $\mathsf{rep}\,A$ using Fact 19.3.1. ■

We also define a recursive predicate for nonrepeating lists over a base type X:

$$\begin{aligned} \mathsf{nrep} :\ & \mathcal{L}(X) \to \mathbb{P} \\ \mathsf{nrep}\,[\,] \ &:=\ \top \\ \mathsf{nrep}\,(x :: A) \ &:=\ x \notin A \land \mathsf{nrep}\,A \end{aligned}$$

Theorem 19.5.2 (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, which needs decidability of membership (Fact 19.3.4) for the cons case. ■

Corollary 19.5.3 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 19.5.4 (Equivalent nonrepeating list)
For every list A over a discrete type one can obtain an equivalent nonrepeating list B such that $\mathsf{len}\,B \le \mathsf{len}\,A$: $\forall A.\Sigma B.\ B \equiv A \land \mathsf{nrep}\,B \land \mathsf{len}\,B \le \mathsf{len}\,A$.

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 building block for further results (Corollary 19.5.6). Finding this fact took experimentation. To get the taste of it, try to prove that equivalent nonrepeating lists have equal length without looking at our development.

Fact 19.5.5 (Discriminating element)
Every nonrepeating list over a discrete type contains for every shorter list an element not in the shorter list: $\forall AB.\ \mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \Sigma x.\ x \in A \land x \notin B$.

Proof By induction on A with B quantified. The base case follows by computational falsity elimination. For $A = a :: A'$ we do case analysis on $(a \in B) + (a \notin B)$. The case $a \notin B$ is trivial. For $a \in B$, Fact 19.4.4 yields some B' shorter than B such that $B \equiv a :: B'$. The inductive hypothesis now yields some $x \in A'$ such that $x \notin B'$. It now suffices to show $x \notin B$. We assume $x \in B \equiv a :: B'$ and derive a contradiction. Since $x \notin B'$, we have $x = a$, which is in contradiction with $\mathsf{nrep}\,(a :: A')$. ∎

Corollary 19.5.6 Let A and B be lists over a discrete type X. Then:

1. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,A \le \mathsf{len}\,B$.
2. $\mathsf{nrep}\,A \to \mathsf{nrep}\,B \to A \equiv B \to \mathsf{len}\,A = \mathsf{len}\,B$.
3. $A \subseteq B \to \mathsf{len}\,B < \mathsf{len}\,A \to \mathsf{rep}\,A$.
4. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \le \mathsf{len}\,A \to \mathsf{nrep}\,B$.
5. $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \le \mathsf{len}\,A \to B \equiv A$.

Proof Interestingly, all claims follow without induction from Facts 19.5.5, 19.5.1, and 19.5.3.

For (1), assume $\mathsf{len}\,A > \mathsf{len}\,B$ and derive a contradiction with Fact 19.5.5.

Claims (2) and (3) follow from Claim (1), where for (3) we assume $\mathsf{nrep}\,A$ and derive a contradiction (justified by Corollary 19.5.3).

For (4), we assume $\mathsf{rep}\,B$ and derive a contradiction (justified by Corollary 19.5.3). By Fact 19.5.1, 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 show $x \in A$. Exploiting the decidability of membership we assume $x \notin A$ and derive a contradiction. Using Fact 19.5.5 for $x :: A$ and B, we obtain $z \in x :: A$ and $z \notin B$, which is contradictory. ∎

We remark that Corollary 19.5.6 (3) may be understood as a pigeonhole lemma.

Exercise 19.5.7 Prove the following facts about map and nonrepeating lists:

a) $\mathsf{injective}\, f \to \mathsf{nrep}\, A \to \mathsf{nrep}\,(f@A)$.

b) $\mathsf{nrep}\,(f@A) \to x \in A \to x' \in A \to fx = fx' \to x = x'$.

c) $\mathsf{nrep}\,(f@A) \to \mathsf{nrep}\, A$.

Exercise 19.5.8 (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 injective if and only if it is surjective.

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 19.5.6 and Exercise 19.5.7.

We can sharpen the problem of the exercise by asking for a proof that a function $\mathcal{O}^n(X) \to \mathcal{O}^n(X)$ is injective if and only if it is surjective. There should be a proof not using lists. See §21.6.

Exercise 19.5.9 (Factorization) 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 19.5.10 (Partition) The proof of Corollary 19.5.3 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 19.5.11 (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: $\forall p^{X \to \mathbb{T}}.\ p[] \to (\forall x A.\ p(A) \to p(A + [x])) \to \forall A.\ pA$.
Hint: By (a) it suffices to prove $\forall A.\ p(\mathsf{rev}\, A)$, which follows by induction on A.

Exercise 19.5.12 (Equivalent nonrepeating lists) Show that equivalent nonrepeating lists have equal length without assuming discreteness of the base type. Hint: Show $\mathsf{nrep}\, A \to A \subseteq B \to \mathsf{len}\, A \le \mathsf{len}\, B$ by induction on A with B quantified using the rearrangement lemma 19.4.3.

Exercise 19.5.13 (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 $\mathsf{even}\, n + \mathsf{odd}\, n$.

19.6 Lists of Numbers

We now come to some facts about lists of numbers whose truth is intuitively clear but whose proofs are surprisingly tricky. The facts about nonrepeating lists turn out to be essential.

A **segment** is a list containing all numbers smaller than its length:

$$\mathsf{segment}\,A \;:=\; \forall k.\; k \in A \longleftrightarrow k < \mathsf{len}\,A$$

A list of numbers is **serial** if for every element it contains all smaller numbers:

$$\mathsf{serial}\,A \;:=\; \forall n{\in}A.\, \forall k{\leq}n.\; k \in A$$

We will show that a list is a segment if and only if it is nonrepeating and serial.

Fact 19.6.1 The empty list is a segment.

Fact 19.6.2 Segments of equal length are equivalent.

Fact 19.6.3 Segments are serial.

Fact 19.6.4 (Segment existence)
$\forall n.\, \Sigma A.\; \mathsf{segment}\,A \wedge \mathsf{len}\,A = n \wedge \mathsf{nrep}\,A.$

Proof By induction on A. ∎

Fact 19.6.5 Segments are nonrepeating.

Proof Let A be a segment. By Facts 19.6.4 and 19.6.2 we have an equivalent nonrepeating segment B of the same length. Hence A is nonrepeating by Fact 19.5.6(4). ∎

Fact 19.6.6 (Large element)
Every nonrepeating list of numbers of length $\mathsf{S}n$ contains a number $k \geq n$:
$\forall A.\; \mathsf{nrep}\,A \to \mathsf{len}\,A = \mathsf{S}n \to \Sigma k{\in}A.\; k \geq n.$

Proof Let A be a nonrepeating list of numbers of length $\mathsf{S}n$. By Fact 19.3.7(1) we can assume $\forall k{\in}A.\; k < n$ and derive a contradiction. Fact 19.6.4 gives us a nonrepeating list B of length n such that $\forall k.\; k \in B \longleftrightarrow k < n$. Now $A \subseteq B$ and $\mathsf{len}\,B < \mathsf{len}\,A$. Contradiction by Fact 19.5.6(1). ∎

Fact 19.6.7 A nonrepeating serial list is a segment.

Proof Let A be a nonrepeating serial list. If $A = []$, the claim is trivial. Otherwise, $\mathsf{len}\,A = \mathsf{S}n$. Fact 19.6.6 gives us $x \in A$ such that $x \geq n$. Fact 19.6.4 gives us a nonrepeating segment B of length $\mathsf{S}n$. We now see that A is a segment if $A \equiv B$. We have $B \subseteq A$ since A is serial and hence contains all $k \leq n \leq x$. Now $A \subseteq B$ follows with Fact 19.5.6(5). ∎

Fact 19.6.8 (Next number) There is function that for every list of numbers yields a number that is not in the list: $\forall A^{\mathcal{L}(\mathsf{N})}.\ \Sigma n.\ \forall k{\in}A.\ k < n.$

Proof By induction on A. ∎

Fact 19.6.8 says that there are infinitely many numbers. More generally, if we have an injection $\mathcal{I}\mathsf{N}X$, we can obtain a new element generator for X and thus know that X is infinite.

Fact 19.6.9 (New element generator)
Given an injection $\mathcal{I}\mathsf{N}X$, there is function that for every list over X yields a number that is not in the list: $\forall A^{\mathcal{L}(X)}.\ \Sigma x.\ \forall a{\in}A.\ a \neq x.$

Proof Let $\mathsf{inv}_{\mathsf{N}X}\, fg$ and $A^{\mathcal{L}X}$. Then Fact 19.6.8 gives us a number $n \notin g@A$. To show $fn \notin A$, assume $fn \in A$. Then $n = g(fn) \in g@A$ contradicting an above assumption. ∎

Exercise 19.6.10 (Pigeonhole) Prove that a list of numbers whose sum is greater than the length of the list must contain a number that is at least 2:

$$\mathsf{sum}\, A > \mathsf{len}\, A \rightarrow \Sigma x.\ x \in A \land x \geq 2$$

First define the function sum.

Exercise 19.6.11 (Andrej's Puzzle) Assume an increasing function $f^{\mathsf{N}\to\mathsf{N}}$ (that is, $\forall x.\ x < fx$) and a list A of numbers satisfying $\forall x.\ x \in A \longleftrightarrow x \in f@A$. Show that A is empty.

Hint: First verify that A contains for every element a smaller element. It then follows by complete induction that A cannot contain an element.

Exercise 19.6.12 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) $x \in \mathsf{seq}\, nk \longleftrightarrow n \leq x < n + k$

e) $\mathsf{nrep}\, (\mathsf{seq}\, nk)$

19.7 Position-Element Mappings

The positions of a list $[x_1, \ldots, x_n]$ are the numbers $0, \ldots, n-1$. More formally, a number n is a **position** of a list A if $n < \mathsf{len}\, A$. If a list is nonrepeating, we have a bijective relation between the positions and the elements of the list. For instance, the list $[7, 8, 5]$ gives us the bijective relation

$$0 \rightsquigarrow 7, \quad 1 \rightsquigarrow 8, \quad 2 \rightsquigarrow 5$$

It turns out that for a discrete type X we can define two functions

$$\begin{aligned} \mathsf{pos} &: \mathcal{L}(X) \to X \to \mathsf{N} \\ \mathsf{sub} &: X \to \mathcal{L}(X) \to \mathsf{N} \to X \end{aligned}$$

realizing the position-element bijection:

$$\begin{aligned} x \in A \to \mathsf{sub}\, y A\, (\mathsf{pos}\, A x) &= x \\ \mathsf{nrep}\, A \to n < \mathsf{len}\, A \to \mathsf{pos}\, A\, (\mathsf{sub}\, y A n) &= n \end{aligned}$$

The function pos 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 21 for constructing injections and bijections between finite types and in Chapter 22 for constructing injections into N.

Here are the definitions of pos and sub we will use:

$$\begin{aligned} &\mathsf{pos} : \mathcal{L}(X) \to X \to \mathsf{N} \\ &\quad \mathsf{pos}\, []\, x := 0 \\ &\quad \mathsf{pos}\, (a :: A)\, x := \text{IF } \ulcorner a = x \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{pos}\, A x) \end{aligned}$$

$$\begin{aligned} &\mathsf{sub} : X \to \mathcal{L}(X) \to \mathsf{N} \to X \\ &\quad \mathsf{sub}\, y\, []\, n := y \\ &\quad \mathsf{sub}\, y\, (a :: A)\, 0 := a \\ &\quad \mathsf{sub}\, y\, (a :: A)\, (\mathsf{S}n) := \mathsf{sub}\, y A n \end{aligned}$$

Fact 19.7.1 Let A be a list over a discrete type. Then:

1. $x \in A \;\to\; \mathsf{sub}\, a\, A\, (\mathsf{pos}\, A x) = x$
2. $x \in A \;\to\; \mathsf{pos}\, A x < \mathsf{len}\, A$
3. $n < \mathsf{len}\, A \;\to\; \mathsf{sub}\, a A\, n \in A$
4. $\mathsf{nrep}\, A \to n < \mathsf{len}\, A \to \mathsf{pos}\, A\, (\mathsf{sub}\, a\, A\, n) = n$

Proof All claims follow by induction on A. For (3), the inductive hypothesis must quantify n and the cons case needs case analysis on n. ∎

Exercise 19.7.2 Prove $(\forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathsf{N} \to X) \to \bot$.

Exercise 19.7.3 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) $k < \mathsf{len}\, A \to \mathsf{sub}\, Ak = \mathsf{sub}\,(A +\!\!\!+ B)k$

c) $x \in A \to y \in A \to \mathsf{pos}\, Ax = \mathsf{pos}\, Ay \to x = y$

Note that (a) relies on the fact that $\mathsf{pos}\, Ax$ yields the first position of x in A, which matters if x occurs more than once in A.

Exercise 19.7.4 One can realize pos and sub with option types

$$\mathsf{pos} : \ \mathcal{L}(X) \to X \to \mathcal{O}(\mathsf{N})$$
$$\mathsf{sub} : \ \mathcal{L}(X) \to \mathsf{N} \to \mathcal{O}(X)$$

and this way avoid the use of escape values. Define pos and sub with option types for a discrete base type X and verify the following properties:

a) $x \in A \to \Sigma n.\ \mathsf{pos}\, Ax = {}^\circ n$

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) $\mathsf{pos}\, Ax = {}^\circ n \to n < \mathsf{len}\, A$

19.8 Constructive Discrimination Lemma

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

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

We speak of the *classical discrimination lemma*. We have already shown a computational version of the lemma (Fact 19.5.5)

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

replacing XM with an equality decider for the base type X. In this section our main interest is in proving the *constructive discrimination lemma*

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

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 techniques for XM have useful applications.

We first prove the classical discrimination lemma following the proof of Fact 19.5.5.

Lemma 19.8.1 (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. The base case follows by computational falsity elimination. For $A = a :: A'$, we do case analysis on $(a \in B) \lor (a \notin B)$ exploiting XM. The case $a \notin B$ is trivial. For $a \in B$, Fact 19.4.3 yields some B' shorter than B such that $B \equiv a :: B'$. The inductive hypothesis now yields some $x \in A'$ such that $x \notin B'$. It now suffices to show $x \notin B$. We assume $x \in B \equiv a :: B'$ and derive a contradiction. Since $x \notin B'$, we have $x = a$, which contradicts $\mathsf{nrep}\,(a :: A')$. ∎

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 12.4.8 (1)). Moreover, we will use the rule formulated by Fact 12.4.8 (2) to erase the double negation from the inductive hypothesis so that we can harvest the witness.

Lemma 19.8.2 (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. The base case follows by computational falsity elimination. Otherwise, we have $A = a :: A'$. Since the claim is stable, we can do case analysis on $a \in B \lor a \notin B$ (Fact 12.4.8 (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 $a \in B$. Fact 19.4.3 yields some B' shorter than B such that $B \equiv a :: B'$. Using Fact 12.4.8 (2), the inductive hypothesis now gives us $x \in A'$ such that $x \notin B'$. By $\forall P.\ P \to \neg\neg P$ it now suffices to show $x \notin B$, which follows as in the proof of Fact 19.8.1. ∎

Exercise 19.8.3 Prove that the double negation of $\exists$ agrees with the double negation of Σ: $\neg\neg\mathsf{ex}\, p \longleftrightarrow ((\mathsf{sig}\, p \to \bot) \to \bot)$.

19.9 Element Removal

We assume a discrete type X and define a function $A \setminus x$ for **element removal** as follows:

$$\setminus :\ \mathcal{L}(X) \to X \to \mathcal{L}(X)$$
$$[\,] \setminus \_ \ := \ [\,]$$
$$(x :: A) \setminus y \ := \ \text{IF } \ulcorner x = y \urcorner \text{ THEN } A \setminus y \text{ ELSE } x :: (A \setminus y)$$

Fact 19.9.1

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 19.9.2 Prove $x \in A \ \to\ A \equiv x :: (A \setminus x)$.

Exercise 19.9.3 Prove the following equations, which are useful in proofs:

1. $(x :: A) \backslash x = A \backslash x$
2. $x \neq y \to (y :: A) \backslash x = y :: (A \backslash x)$

19.10 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 19.5.6 (3)), and every list is equivalent to a non-repeating list (Fact 19.5.4).

We assume that lists are taken over a discrete type X and define a **cardinality function** as follows:

$$\begin{aligned} \mathsf{card} :\ & \mathcal{L}(X) \to \mathsf{N} \\ \mathsf{card}\,[] \ &:=\ 0 \\ \mathsf{card}(x :: A) \ &:=\ \text{IF } \ulcorner x \in A \urcorner \text{ THEN } \mathsf{card}\,A \text{ ELSE } \mathsf{S}(\mathsf{card}\,A) \end{aligned}$$

Note that we write $\ulcorner x \in A \urcorner$ for the application of the membership decider provided by Fact 19.3.4. We prove that the cardinality function agrees with the cardinalities provided by equivalent nonrepeating lists.

Fact 19.10.1 (Cardinality)

1. $\forall A\, \Sigma B.\ B \equiv A \wedge \mathsf{nrep}\,B \wedge \mathsf{len}\,B = \mathsf{card}\,A.$
2. $\mathsf{card}\,A = n \ \longleftrightarrow\ \exists B.\ B \equiv A \wedge \mathsf{nrep}\,B \wedge \mathsf{len}\,B = n.$

Proof Claim 1 follows by induction on A. Claim 2 follows with Claim 1 and Corollary 19.5.6 (2). ∎

Corollary 19.10.2

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 19.10.1, Corollary 19.5.6, and Corollary 19.5.3. ■

Exercise 19.10.3 Given direct proofs of (1), (4) and (5) of Corollary 19.10.2 by induction on A. Use (1) for (4) and (5).

Exercise 19.10.4 (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 19.10.5 (Disjointness predicate) We define **disjointness** of lists as follows:

$$\mathsf{disjoint}\,A\,B := \neg\exists x.\ x \in A \wedge x \in B$$

Define a recursive predicate $\mathsf{Disjoint} : \mathcal{L}(X) \to \mathcal{L}(X) \to \mathbb{P}$ in the style of the cardinality predicate and verify that it agrees with the above predicate $\mathsf{disjoint}$.

19.11 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 19.11.1 (Compatibility laws for propositional equivalence)
Let $P \longleftrightarrow P'$ and $Q \longleftrightarrow Q'$. Then:

$$P \land Q \longleftrightarrow P' \land Q' \qquad P \lor Q \longleftrightarrow P' \lor Q' \qquad (P \to Q) \longleftrightarrow (P' \to Q')$$
$$\neg P \longleftrightarrow \neg P' \qquad (P \longleftrightarrow Q) \longleftrightarrow (P' \longleftrightarrow Q')$$

Fact 19.11.2 (Compatibility laws for list equivalence)
Let $A \equiv A'$ and $B \equiv B'$. Then:

$$x \in A \longleftrightarrow x \in A' \qquad A \subseteq B \longleftrightarrow A' \subseteq B' \qquad A \equiv B \longleftrightarrow A' \equiv B'$$
$$x :: A \equiv x :: A' \qquad A + B \equiv A' + B' \qquad f@A \equiv f@A'$$

Coq's setoid rewriting facility makes it possible to use the rewriting tactic for rewriting with equivalences, provided the necessary compatibility laws and equivalence relations have been registered with the facility. The compatibility laws for propositional equivalence are preregistered.

Exercise 19.11.3 Which of the compatibility laws are needed to justify rewriting the claim $\neg(x \in y :: (f@A) + B)$ with the equivalence $A \equiv A'$?

20 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.

20.1 Expressions and Evaluation

We will consider arithmetic expressions 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 20.1.1 Do the reduction $E(3+5-2) \succ^* 6$ step by step (at the equational level).

Exercise 20.1.2 Prove some of the constructor laws for expressions. For instance, show that con is injective and that add and sub are disjoint.

Exercise 20.1.3 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.

20.2 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 on a stack A as follows:

$$
\begin{aligned}
&R : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N}) \\
R\ []\ A \ &:=\ A \\
R\ (0 :: C)\ (x_1 :: x_2 :: A) \ &:=\ R\ C\ (x_1 + x_2 :: A) \\
R\ (1 :: C)\ (x_1 :: x_2 :: A) \ &:=\ R\ C\ (x_1 - x_2 :: A) \\
R\ (\mathsf{SS}x :: C)\ A \ &:=\ R\ C\ (x :: A) \\
R\ \_\ \_ \ &:=\ []
\end{aligned}
$$

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: take two numbers from the stack and put their sum on the stack.
- 1: take two numbers from the stack and put their difference on the stack.
- $\mathsf{SS}x$: put x on the 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 20.2.1 Do the reduction $R[5,7,1][] \succ^* [2]$ step by step (at the equational level).

20.3 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:

$$\begin{aligned} &\gamma : \mathsf{exp} \to \mathcal{L}(\mathsf{N}) \\ \gamma x \ &:= \ [\mathsf{SS}x] \\ \gamma(e_1 + e_2) \ &:= \ \gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [0] \\ \gamma(e_1 - e_2) \ &:= \ \gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [1] \end{aligned}$$

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, will fail 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 an elegant correctness theorem telling us much more about code execution than the correctness equation we started with. Formulated in words, the correctness theorem says that executing the code $\gamma e +\!\!+ C$ on a stack A gives the same result as executing the code C on the stack $Ee :: A$.

Theorem 20.3.1 (Correctness) $R\ (\gamma e + C)\ A = R\ C\ (Ee :: A)$.

Proof By induction on e. The case for addition proceeds as follows:

$$
\begin{aligned}
& R\ (\gamma(e_1 + e_2) + C)\ A && \\
= {} & R\ (\gamma e_2 + \gamma e_1 + [0] + C)\ A && \text{definition } \gamma \\
= {} & R\ (\gamma e_1 + [0] + C)\ (Ee_2 :: A) && \text{inductive hypothesis} \\
= {} & R\ ([0] + C)\ (Ee_1 :: Ee_2 :: A) && \text{inductive hypothesis} \\
= {} & R\ C\ ((Ee_1 + Ee_2) :: A) && \text{definition } R \\
= {} & R\ C\ (E(e_1 + e_2) :: A) && \text{definition } E
\end{aligned}
$$

The equational reasoning shown tacitly employs conversion and associativity for concatenation $+$. The details can be explored with the proof assistant. ■

Corollary 20.3.2 $R\ (\gamma e)\ [] = [Ee]$.

Proof Theorem 20.3.1 with $C = A = []$. ■

Exercise 20.3.3 Do the reduction $\gamma(5-2) \succ^* [4, 7, 1]$ step by step (at the equational level).

Exercise 20.3.4 Explore the proof of the correctness theorem starting with the proof script in the accompanying Coq development.

20.4 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 employed:

$$
\begin{aligned}
& \delta : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{exp}) \to \mathcal{L}(\mathsf{exp}) \\
\delta\ []\ A \ &:= \ A \\
\delta\ (0 :: C)\ (e_1 :: e_2 :: A) \ &:= \ \delta\ C\ (e_1 + e_2 :: A) \\
\delta\ (1 :: C)\ (e_1 :: e_2 :: A) \ &:= \ \delta\ C\ (e_1 - e_2 :: A) \\
\delta\ (\mathsf{SS}x :: C)\ A \ &:= \ \delta\ C\ (x :: A) \\
\delta\ \_\ \_ \ &:= \ []
\end{aligned}
$$

The correctness theorem for decompilation closely follows the correctness theorem for compilation.

Theorem 20.4.1 (Correctness) $\delta\ (\gamma e + C)\ B = \delta\ C\ (e :: B)$.

Proof By induction on e. The case for addition proceeds as follows:

$$\begin{aligned}
& \delta\ (\gamma(e_1 + e_2) + C)\ B \\
= {} & \delta\ (\gamma e_2 + \gamma e_1 + [0] + C)\ B && \text{definition } \gamma \\
= {} & \delta\ (\gamma e_1 + [0] + C)\ (e_2 :: B) && \text{inductive hypothesis} \\
= {} & \delta\ ([0] + C)\ (e_1 :: e_2 :: B) && \text{inductive hypothesis} \\
= {} & \delta\ C\ ((e_1 + e_2) :: B) && \text{definition } \delta
\end{aligned}$$

The equational reasoning tacitly employs conversion and associativity for concatenation $+$. ■

Corollary 20.4.2 $\delta\ (\gamma e)\ [] = [e]$.

20.5 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 [22] and Burstall [6]. 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 [7], 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.

21 Finite Types

We may call a type finite if we can provide a list containing all its elements. With this characterization we capture the informal notion of finitely many elements with covering lists. If the list covering the elements of a type is nonrepeating, the length of the list gives us the cardinality of the type (the number of its elements). This leads us to defining a finite type as a tuple consisting of a type, a list covering the type, and an equality decider for elements of the type. The equality decider ensures that the finite type is sufficiently concrete so that we can compute a covering nonrepeating list yielding the cardinality of the type.

With this definition the numeral types $\mathcal{O}^n\bot$ are in fact finite types of cardinality n. We will show that finite types are closed under retracts and that two finite types of equal cardinality are always in bijection. It then follows that finite types have EWOs, and that the class of finite types is generated from the empty type by taking option types and including types that are in bijection with a member of the class. We also show that a function between finite types of the same cardinality

- is injective if and only if it is surjective.
- has an inverse function if it is surjective or injective.
- yields a bijection with every inverse function.

21.1 Coverings and Listings

We prepare the definition of finite types by looking at lists covering all elements of their base type.

A **covering of a type** is a list that contains every member of the type:

$$\mathsf{covering}_X\ A\ :=\ \forall x^X.\ x \in A$$

A **listing of a type** is a nonrepeating covering of the type:

$$\mathsf{listing}_X\ A\ :=\ \mathsf{covering}\ A \wedge \mathsf{nrep}\ A$$

We need a couple of results for coverings and listings of discrete types. First note that all coverings of a type are equivalent lists.

Fact 21.1.1 Given a covering of a discrete type, one can obtain a listing of the type: $\mathcal{E}X \to \mathsf{covering}_X\, A \to \Sigma B.\ \mathsf{listing}_X\, B \wedge \mathsf{len}\, B \le \mathsf{len}\, A$.

Proof Follows with Facts 19.5.4. ∎

Fact 21.1.2 All listings of a discrete type have the same length.

Proof Immediate with Fact 19.5.6 (2). ∎

Fact 21.1.3 Let A and B be lists over a discrete type X. Then:
$\mathsf{listing}\, A \to \mathsf{len}\, B = \mathsf{len}\, A \to (\mathsf{nrep}\, B \longleftrightarrow \mathsf{covering}\, B)$.

Proof Follows with Fact 19.5.6 (4,5). ∎

Exercise 21.1.4 Prove $\mathcal{I}XY \to \mathsf{covering}_Y B \to \Sigma A.\ \mathsf{covering}_X A$.

Exercise 21.1.5 Let A and B be lists over a discrete type X. Prove:

a) $\mathsf{covering}\, A \to \mathsf{nrep}\, B \to \mathsf{len}\, A \le \mathsf{len}\, B \to \mathsf{listing}\, B$.

b) $\mathsf{listing}\, A \to \mathsf{covering}\, B \to \mathsf{len}\, B \le \mathsf{len}\, A \to \mathsf{listing}\, B$.

21.2 Basics of Finite Types

We define **finite types** as discrete types that come with a covering list:

$$\mathsf{fin}\, X^{\mathbb{T}} \;:=\; \mathcal{E}(X) \times \Sigma A.\ \mathsf{covering}_X\, A$$

We may see the values of $\mathsf{fin}\, X$ as handlers providing an equality decider for X and a listing of X. A handler $\mathsf{fin}\, X$ turns a type X into a computational type where we can iterate over all elements and decide equality of elements. Given a handler $\mathsf{fin}\, X$, we can compute a listing of X and thus determine the *cardinality* of X (i.e., the number of elements). It will be convenient to have a second type of handlers

$$\mathsf{fin}_n\, X^{\mathbb{T}} \;:=\; \mathcal{E}(X) \times \Sigma A.\ \mathsf{listing}_X\, A \wedge \mathsf{len}\, A = n$$

declaring the **cardinality** of the type and providing a listing rather than a covering of the type.

Fact 21.2.1 For every type X:

1. $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, X \to m = n$ (uniqueness of cardinality)
2. $\mathsf{fin}\, X \Leftrightarrow \Sigma n.\, \mathsf{fin}_n\, X$ (existence of cardinality)

Proof Facts 21.1.2 and 21.1.1. ∎

Fact 21.2.2 (Empty types)
Finite types with cardinality 0 are the empty types: $\mathsf{fin}_0\,X \Leftrightarrow (X \to \bot)$.

Proof Exercise. ∎

Fact 21.2.3 (Closure under $\mathcal{O}$)
Finite types are closed under $\mathcal{O}$: $\mathsf{fin}_n\,X \to \mathsf{fin}_{\mathsf{S}n}\,(\mathcal{O}X)$.

Proof Fact 10.7.1 gives us an equality decider for $\mathcal{O}X$. Moreover, $\emptyset :: (°)@A)$ is a listing of $\mathcal{O}X$ if A is a listing of X. ∎

Fact 21.2.4 (Numeral types) $\mathsf{fin}_n\,(\mathcal{O}^n\bot)$.

Proof Induction on n using Facts 21.2.2 and 21.2.3. ∎

Finite types are also closed under sums and products.

Fact 21.2.5 If X and Y are finite types, then so are $X + Y$ and $X \times Y$:

1. $\mathsf{fin}_m\,X \to \mathsf{fin}_n\,Y \to \mathsf{fin}_{m+n}(X + Y)$.
2. $\mathsf{fin}_m\,X \to \mathsf{fin}_n\,Y \to \mathsf{fin}_{m\cdot n}(X \times Y)$.

Proof Discreteness follows with Fact 10.3.1. We leave the construction of the listing as exercise. ∎

Quantification over finite types preserves decidability. This fact can be obtained from the fact that quantification over lists preserves decidability.

Fact 21.2.6 (Decidability)
Let p be a decidable predicate on a finite type X. Then:

1. $\mathcal{D}(\forall x.px)$
2. $\mathcal{D}(\exists x.px)$
3. $(\Sigma x.px) + (\forall x.\neg px)$

Proof Follows with Fact 19.3.7. ∎

Fact 21.2.7 (Type of numbers is not finite)
The type N of numbers is not finite: $\mathsf{fin}\,\mathsf{N} \to \bot$.

Proof Suppose N is finite. Then we have a list A containing all numbers. Contradiction with Fact 19.6.8. ∎

Exercise 21.2.8 Prove $\mathsf{fin}_0\,\bot$, $\mathsf{fin}_1\,\top$, and $\mathsf{fin}_2\,\mathsf{B}$.

Exercise 21.2.9 (Double negation shift)
Prove $\forall n\, \forall p^{\mathsf{F}_n \to \mathbb{P}}.\ (\forall x.\ \neg\neg px) \to \neg\neg\forall x.\ px$.

Exercise 21.2.10 (EWO) Give an EWO for finite types.
Hint: Use an EWO for lists as in Exercise 19.3.9.

Exercise 21.2.11 (Pigeonhole)
Prove $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to m > n \to \forall f^{X \to Y}.\ \Sigma x x'.\ x \neq x' \wedge fx = fx'$.
Intuition: If we have m pigeons sitting in $n < m$ holes, there must be two pigeons sitting in the same hole.

21.3 Finiteness by Injection

We can establish the finiteness of a nonempty type by embedding it into a finite type with large enough cardinality. More precisely, a type is finite with cardinality $m \geq 1$ if it can be embedded into a finite type with cardinality $n \geq m$.

Fact 21.3.1 (Finiteness by embedding)
$\mathcal{I}XY \to \mathsf{fin}_n\, Y \to \Sigma m \leq n.\ \mathsf{fin}_m\, X$.

Proof Let $\mathsf{inv}_{XY}\, gf$ and B be a listing of Y such that $\mathsf{len}\, B = n$. Fact 10.3.2 gives us an equality decider for X. Moreover, $g@B$ is a covering of X because of the inversion property. By Fact 21.1.1 we obtain a listing A of X such that $\mathsf{len}\, A \leq \mathsf{len}\,(g@B) = \mathsf{len}\, B = n$. The claim follows. ∎

Corollary 21.3.2 (Alignment)
$\mathcal{I}XY \to \mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to m \leq n$.

Proof Facts 21.3.1 and 21.2.1 (1). ∎

Corollary 21.3.3 (Transport)
$\mathcal{I}XY \to \mathcal{I}YX \to \mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y$.

Proof Follows with Facts 21.3.1 and 21.3.2. ∎

Corollary 21.3.4 (Closure under bijection)
$\mathcal{B}XY \to \mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y$.

Proof Follows with Facts 21.3.3 and 10.6.5. ∎

Corollary 21.3.5 The type N of numbers does not embed into finite types:
$\mathcal{I}\,\mathsf{N}X \to \mathsf{fin}\, X \to \bot$.

Proof Facts 21.3.1 and 21.2.7. ∎

21.4 Existence of Injections

Given two finite types, the smaller one can always be embedded into the larger one. There is the caveat that the smaller type must not be empty so that the embedding function can have an inverse.

Fact 21.4.1 (Existence)
$\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to 1 \le m \le n \to \mathcal{I}XY$.

Proof Let A and B be listings of X and Y. Then A has length m and B has length n. Since $1 \le m \le n$, we can map the elements of A to elements of B preserving the position in the lists. We realize the resulting bijection between X and Y using the list operations sub and pos with escape values $a \in A$ and $b \in B$ (§19.7):

$$\begin{aligned} fx &:= \mathsf{sub}\, b\, B\ (\mathsf{pos}\, A\, x) \\ gy &:= \mathsf{sub}\, a\, A\ (\mathsf{pos}\, B\, y) \end{aligned}$$

Recall that pos yields the position of a value in a list, and that sub yields the value at a position of a list. ■

An embedding of a finite type into a finite type of the same cardinality is in fact a bijection since in this case the second roundtrip property does hold.

Fact 21.4.2 $\mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y \to \mathsf{inv}_{XY}\, gf \to \mathsf{inv}\, fg$.

Proof We show $f(gy) = y$ for arbitrary y. We choose a covering A of X and know by Fact 21.1.3 that $f@A$ is a covering of Y. Hence $fx = y$ for some x. We now have $f(gy) = f(g(fx)) = fx = y$. ■

Next we show that all finite types of the same size are in bijection.

Corollary 21.4.3 (Existence) $\mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y \to \mathcal{B}XY$.

Proof For $n = 0$ the claim follows with Facts 21.2.2 and 10.6.8. For $n > 0$ the claim follows with Facts 21.4.1 and 21.4.2. ■

Corollary 21.4.4 (Existence) $\mathcal{I}XY \to \mathcal{I}YX \to \mathsf{fin}\, X \to \mathcal{B}XY$.

Proof Facts 21.2.1 (2), 21.3.3, and 21.4.3 ■

Corollary 21.4.5 (Listless Characterization) $\mathsf{fin}_n\, X \Leftrightarrow \mathcal{B}X(\mathcal{O}^n\bot)$.

Proof Direction $\to$ follows with Facts 21.2.4 and 21.4.3. Direction $\leftarrow$ follows with Facts 21.2.4 and 21.3.4. ■

Corollary 21.4.6 $\mathsf{fin}\, X \to \Sigma n.\, \mathcal{B}X(\mathcal{O}^n\bot)$

Proof Facts 21.2.1 and 21.4.5. ■

Corollary 21.4.7 (EWOs) Finite types have EWOs.

Proof Follows with Fact 21.4.5 since numeral types have EWOs (Fact 18.1.4) and injections transport EWOs (Fact 18.1.5). ■

Fact 21.4.8 (Existence) Every nonempty finite type can be embedded into N: $\mathsf{fin}_{\mathsf{S}n}\, X \to \mathcal{I}\, X\mathsf{N}$.

Proof Let A be a listing of X. We realize the injection of X into N using the list operations pos and sub with an escape values $a \in A$ (§19.7):

$$
\begin{aligned}
fx &:= \mathsf{pos}\, A\, x \\
gn &:= \mathsf{sub}\, a\, A\, n
\end{aligned}
$$
■

Exercise 21.4.9 Show the following facts:

a) $\mathsf{fin}_m\, \mathsf{F}_n \to m = n$.

b) $\mathcal{B}\, \mathsf{F}_m \mathsf{F}_n \to m = n$.

21.5 Upgrade Theorem

Fact 21.5.1 (Injectivity-surjectivity agreement)
Functions between finite types of the same cardinality are injective if and only if they are surjective: $\mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y \to (\mathsf{injective}\, XYf \longleftrightarrow \mathsf{surjective}\, XYf)$.

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 21.1.3.

Let f be injective. Then $f@A$ is nonrepeating by Exercise 19.5.7 (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 19.5.7 (b). ■

Fact 21.5.2 (Upgrade) Let $f^{X \to Y}$ be a surjective or injective function between finite types of the same cardinality. Then one can obtain a function g such that f and g constitute a bijection between X and Y.

Proof Let $\mathsf{fin}_n X$ and $\mathsf{fin}_n Y$. The both types have equality deciders and EWOs (Fact 21.4.7). By Fact 21.5.1 we can assume a surjective function $f^{X \to Y}$. Fact 18.1.6 gives us a function g such that $\mathsf{inv}\, f g$. Fact 21.4.2 gives us $\mathsf{inv}\, g f$ as claimed. ■

Exercise 21.5.3
Prove $\mathsf{fin}_m\, X \to \mathsf{fin}_n\, Y \to m > 0 \to (\forall f^{X \to Y}.\ \mathsf{injective}\, f \longleftrightarrow \mathsf{surjective}\, f) \to m = n$.

Exercise 21.5.4 Show that all inverse functions of an injective function between finite types of the same cardinality agree.

21.6 Listless Development

Fact 21.4.5 characterizes finite types with numeral types and bijections not using lists. In fact, in set theory finite sets are usually obtained as sets that are in bijection with canonical finite sets. Moreover, a number n is represented as a particular set with exactly n elements. Following the development in set theory, one may study finite types in type theory not using lists. Important results of such a development would be the following theorems:

1. $\mathcal{B}\, \mathsf{F}_m \mathsf{F}_n \to m = n$
2. $\forall f g^{\mathsf{F}_n \to \mathsf{F}_n}.\ \mathsf{inv}\, g f \to \mathsf{inv}\, f g$
3. $\mathcal{I}\, \mathsf{F}_n \mathsf{N}$
4. $\mathcal{I}\, \mathsf{N}\, \mathsf{F}_n \to \bot$

In the proofs of these results induction over numbers (i.e., the cardinality n) will replace induction over lists.

We will give a few list-free proofs for numeral types since they are interesting from a technical perspective. They require a different mindset and sometimes require tricky techniques for option types.

The main tool for proving properties of numeral types F_n is induction on the cardinality n. An important insight we will use is that we can lower an embedding of $\mathcal{O}^2(X)$ into $\mathcal{O}^2(Y)$ into an embedding of $\mathcal{O}(X)$ into $\mathcal{O}(Y)$. We realize the idea with a **lowering operator** as follows:

$$L :\ \forall XY.\ (\mathcal{O}(X) \to \mathcal{O}^2(Y)) \to X \to \mathcal{O}(Y)$$
$$LXYfx \ := \ \text{MATCH}\ f(°x)\ [\ °b \Rightarrow b \mid \emptyset \Rightarrow \text{MATCH}\ f\emptyset\ [\ °b \Rightarrow b \mid \emptyset \Rightarrow \emptyset\ ]\ ]$$

The idea behind L is simple: Given x, Lf checks whether f maps $°x$ to $°b$. If so, Lf maps x to b. Otherwise, Lf checks whether f maps $\emptyset$ to $°b$. If so, Lf maps x to b. If not, Lf maps x to $\emptyset$.

Lemma 21.6.1 (Lowering) Let $f : \mathcal{O}^2(X) \to \mathcal{O}^2(Y)$ and $g : \mathcal{O}^2(Y) \to \mathcal{O}^2(X)$. Then $\mathsf{inv}\, g f \to \mathsf{inv}\, (Lg)(Lf)$.

Proof Let $\mathsf{inv}\, g f$. We show $(Lg)(Lfa) = a$ by brute force case analysis following the matches of Lf and Lg. There are 2^4 cases that all follow with equational reasoning as provided by the ongruence tactic. ∎

We can now show that a self-injection of a numeral type is always a bijection.

Theorem 21.6.2 (Self-injection)
Let f and g be functions $\mathsf{F}_n \to \mathsf{F}_n$. Then $\mathsf{inv}\, gf \to \mathsf{inv}\, fg$.

Proof We prove the claim by induction on n. For $n = 0$ and $n = 1$ the proofs are straightforward.

Let $f, g : \mathcal{O}^{\mathsf{SS}n}(\bot) \to \mathcal{O}^{\mathsf{SS}n}(\bot)$ and $\mathsf{inv}\, gf$. By Lemma 21.6.1 and the inductive hypothesis we have $\mathsf{inv}\,(Lf)(Lg)$. We consider 2 cases:

1. $f(g\emptyset) = \emptyset$. We show $f(g{^\circ}b) = {^\circ}b$. We have $(Lf)(Lgb) = b$. The claim now follows by case analysis and linear equational reasoning following the definitions of Lf and Lg (7 cases are needed).
2. $f(g\emptyset) = {^\circ}b$. We derive a contradiction.
   a) $f\emptyset = {^\circ}b'$ We have $(Lf)(Lgb') = b'$. A contradiction follows by case analysis and linear equational reasoning following the definitions of Lf and Lg (4 cases are needed).
   b) $f\emptyset = \emptyset$. Contradictory since $\mathsf{inv}\, gf$. ■

The above proof requires the verification of 12 cases by linear equational reasoning as realized by Coq's congruence tactic. The cascaded case analysis of the proof is cleverly chosen as to minimize the cases that need to be considered. The need for cascaded case analysis of function applications so that linear equational reasoning can finish the current branch of the proof appeared before with Kaminski's equation (§6.1).

We remark that the lowering operator is related to the certifying lowering operator established by Lemma 10.8.1. However, there are essential differences. The lowering operator uses a default value while Lemma 10.8.1 exploits an assumption and computational falsity elimination to avoid the need for a default value. In fact, the default value is not available in the setting of Lemma 10.8.1, and the assumption is not available in the setting of the lowering operator.

Theorem 21.6.2 stands out in that its proof requires the verification of more cases than one feels comfortable with on paper. Here the accompanying verification with a proof assistant gives confidence beyond intuition and common belief.

We now generalize self-injection to general finite types.

Corollary 21.6.3 $\mathcal{B}X\mathsf{F}_n \to \mathcal{B}Y\mathsf{F}_n \to \mathsf{inv}_{XY}\, gf \to \mathsf{inv}\, fg$.

Proof Let f_1, g_1 form a bijection $\mathcal{B}X\mathsf{F}_n$ and f_2, g_2 form a bijection $\mathcal{B}X\mathsf{F}_n$. We have

$$\mathsf{inv}\,(\lambda a.f1(g(g_2 a)))\,(\lambda a.f2(f(g_1 a)))$$

by equational reasoning (congruence tactic in Coq). Theorem 21.6.2 gives us

$$\mathsf{inv}\,(\lambda a.f2(f(g_1 a)))\,(\lambda a.f1(g(g_2 a)))$$

This gives us $\mathsf{inv}\, fg$ by equational reasoning (congruence tactic in Coq). ∎

Using the lowering lemma, we can also prove a cardinality result for numeral types.

Theorem 21.6.4 (Cardinality)
$\mathcal{I}(\mathsf{F}_m)(\mathsf{F}_n) \to m \leq n$.

Proof Let $f : \mathsf{F}_m \to \mathsf{F}_n$ and $\mathsf{inv}\, gf$. If $m = 0$ or $n = 0$ the claim is straightforward. Otherwise we have $f : \mathcal{O}^{\mathsf{S}m}(\bot) \to \mathcal{O}^{\mathsf{S}n}(\bot)$ and $\mathsf{inv}\, gf$. We prove $m \leq n$ by induction on m with n, f, and g quantified. For $m = 0$ the claim is trivial. In the successor case, we need to show $\mathsf{S}m \leq n$. If $n = 0$, we have $f : \mathcal{O}^{\mathsf{SS}m}(\bot) \to \mathcal{O}(\bot)$ contradicting $\mathsf{inv}\, gf$. If $n > 0$, the claim follows by Lemma 21.6.1 and the inductive hypothesis. ∎

Exercise 21.6.5 Show $\mathcal{B}\, \mathsf{F}_m \mathsf{F}_n \to m = n$ using induction and the bijection theorem for option types (10.8.2).

Exercise 21.6.6 Try to do the proof of Theorem 21.6.2 without looking at the details of the given proof. This will make you appreciate the cleverness of the case analysis of the given proof. It took a few iterations to arrive at this proof. Acknowledgements go to Andrej Dudenhefner.

Exercise 21.6.7 (Pigeonhole)
Prove $\forall f^{\mathsf{F}_{\mathsf{S}n} \to \mathsf{F}_n}.\ \Sigma ab.\ a \neq b \wedge fa = fb$ not using lists.
Hint: A proof similar to the proof of Theorem 21.6.4 works, but the situation is simpler. The decision function from Fact 21.2.6 (c) is essential.

Exercise 21.6.8 (Double negation shift)
Prove $\forall n\, \forall p^{\mathsf{F}_n \to \mathbb{P}}.\ (\forall x.\ \neg\neg px) \to \neg\neg\forall x.\ px$ not using lists.

Exercise 21.6.9 (Embedding numeral types into the type of numbers)
Numeral types can be embedded into the numbers by interpreting the constructor $\emptyset$ as 0 and the constructor $^\circ$ as successor.

a) Define an encoding function $E : \forall n.\ \mathsf{F}_n \to \mathsf{N}$.
b) Define a decoding function $D : \mathsf{N} \to \forall n.\ \mathsf{F}_{\mathsf{S}n}$.
c) Prove $Ena < n$.
d) Prove $D(E(\mathsf{S}n)a)n = a$.
e) Prove $k \leq n \to E(\mathsf{S}n)(Dkn) = k$.

Hint: The definition of E needs computational falsity elimination.

Exercise 21.6.10 (Decidability)
Let p be a decidable predicate on F_n. Then:

1. $\mathcal{D}(\forall x.px)$
2. $\mathcal{D}(\exists x.px)$
3. $(\Sigma x.px) + (\forall x.\neg px)$

Exercise 21.6.11 (Finite Choice)
We define the *choice property* for two types X and Y as follows:

$$\mathsf{choice}\,XY \;:=\; \forall p^{X\to Y\to\mathbb{P}}.\,(\forall x\exists y.\,pxy) \to \exists f\,\forall x.\,px(fx)$$

The property says that every total relation from X to Y contains a function from X to Y. Prove $\mathsf{choice}\,XY$ for all finite types X:

a) $\mathsf{choice}\,\bot\,Y$
b) $\mathsf{choice}\,X\,Y \to \mathsf{choice}\,(\mathcal{O}X)\,Y$
c) $\mathsf{choice}\,(\mathsf{F}_n)\,Y$
d) $\mathcal{B}\,X\,(\mathsf{F}_n) \to \mathsf{choice}\,XY$

The proposition $\mathsf{choice}\,\mathsf{N}\,Y$ is known as *countable choice* for Y. The computational type theory we are considering cannot prove countable choice for B.

21.7 Notes

We have chosen to define finite types using lists. This is a natural and convenient definition given that lists are a basic computational data structure. On the other hand, we could define finite types more abstractly as types that are in bijection with numeral types obtained with option types and recursion. This definition is backed up by two bijection theorems (21.6.4 and 21.6.2).

22 Countable Types

Countable types include finite types and N, and are closed under retracts, sums, cartesian products, and list types. All countable types have equality deciders, enumerators, and EWOs. Infinite countable types are in bijection with N. Typical examples for infinite countable types are inductive types for syntactic objects (e.g., expressions and formulas). As it comes to characterizations of countable types, a type X is countable iff X has an enumerator and an equality decider, or iff $\mathcal{O}X$ is a retract of N, or iff X is a retract of N and also inhabited.

22.1 Enumerable Types

An **enumerator** of a type X is a function $f^{\mathsf{N}\to\mathcal{O}(X)}$ that reaches all elements of X. Ofmally, we define the **type of enumerators** of a type X as follows:

$$\begin{aligned} \mathsf{enum}'\,X\,f^{\mathsf{N}\to\mathcal{O}X} &:= \forall x\,\exists n.\ fn = {}^{\circ}x \\ \mathsf{enum}\,X &:= \Sigma f.\,\mathsf{enum}'\,X f \end{aligned}$$

We say that a type is **enumerable** if it has an enumerator.

Fact 22.1.1 (Enumerable types)

1. $\bot$ and N are enumerable types: $\mathsf{enum}\,\bot$ and $\mathsf{enum}\,\mathsf{N}$.
2. Enumerable types are closed under retracts, $\mathcal{O}$, $+$, and $\times$:
   a) $\mathcal{I}XY \to \mathsf{enum}\,Y \to \mathsf{enum}\,X$
   b) $\mathsf{enum}\,X \Leftrightarrow \mathsf{enum}(\mathcal{O}\,X)$
   c) $\mathsf{enum}\,X \times \mathsf{enum}\,Y \Leftrightarrow \mathsf{enum}(X+Y)$
   d) $\mathsf{enum}\,X \to \mathsf{enum}\,Y \to \mathsf{enum}(X\times Y)$
3. Finite types are enumerable: $\mathsf{fin}\,X \to \mathsf{enum}\,X$.

Proof Straightforward. Closure under $+$ and $\times$ follows with $\mathcal{I}(\mathsf{N}\times\mathsf{N})\mathsf{N}$ (Chapter 7). (3) follows since $\mathsf{fin}\,X \to \Sigma n.\ \mathcal{I}X(\mathcal{O}^n\bot)$ (Fact 21.4.6). ∎

Given a function $f^{\mathsf{N}\to\mathcal{O}X}$, we call a function $g^{X\to\mathsf{N}}$ such that $\forall x.\ f(gx) = {}^{\circ}x$ a **co-enumerator** for f.

Fact 22.1.2 (Co-enumerator) If $f^{\mathsf{N}\to\mathcal{O}X}$ has a co-enumerator g, then f is an enumerator of X, g is injective, and X has an equality decider.

Fact 22.1.3 (Co-enumerator) Enumerators of discrete types have co-enumerators: $\mathcal{E}X \to \mathsf{enum}'Xf \to \Sigma g.\,\forall x.\, f(gx) = {}^\circ x$.

Proof It suffices to show $\forall x.\Sigma n.\, fn = {}^\circ x$. Follows with an EWO of N since $\mathcal{O}X$ is discrete. ■

22.2 Countable Types

A **countable** type is a type coming with an equality decider and an enumerator:

$$\mathsf{cty}\,X \;:=\; \mathcal{E}(X) \times \mathsf{enum}\,X$$

Fact 22.2.1 (Countable types)

1. $\bot$ and N are countable types: $\mathsf{cty}\,\bot$ and $\mathsf{cty}\,\mathsf{N}$.
2. Countable types are closed under retracts, $\mathcal{O}$, $+$, and $\times$:
   a) $\mathcal{I}XY \to \mathsf{cty}\,Y \to \mathsf{cty}\,X$
   b) $\mathsf{cty}\,X \Leftrightarrow \mathsf{cty}(\mathcal{O}\,X)$
   c) $\mathsf{cty}\,X \times \mathsf{cty}\,Y \Leftrightarrow \mathsf{cty}(X+Y)$
   d) $\mathsf{cty}\,X \to \mathsf{cty}\,Y \to \mathsf{cty}(X\times Y)$
3. Finite types are countable: $\mathsf{fin}\,X \to \mathsf{cty}\,X$.

Proof Follows with Fact 22.1.1 and the concomitant closure facts for discrete types. ■

Fact 22.2.2 (Co-enumerator characterization)
A type X is countable if and only if there are functions $f^{\mathsf{N}\to\mathcal{O}X}$ and $g^{X\to\mathsf{N}}$ such that g is a co-enumerator of f.

Proof Facts 22.1.3 and 22.1.2. ■

It turns out that a type X is countable if and only if $\mathcal{O}X$ is a retract of N.

Fact 22.2.3 (Retract characterization) $\mathsf{cty}\,X \Leftrightarrow \mathcal{I}(\mathcal{O}X)\mathsf{N}$.

Proof Suppose $\mathsf{cty}\,X$. Fact 22.2.2 gives us f and g such that $\forall x.\ f(gx) = {}^\circ x$. We obtain an injection $\mathcal{I}(\mathcal{O}X)\mathsf{N}$ as follows:

$$\begin{aligned} g'a &:= \text{MATCH}\ a\ [\,{}^\circ x \Rightarrow \mathsf{S}(gx) \mid \emptyset \Rightarrow 0\,] \\ f'n &:= \text{MATCH}\ n\ [\,0 \Rightarrow \emptyset \mid \mathsf{S}n \Rightarrow fn\,] \end{aligned}$$

The other direction follows with the transport lemmas for equality deciders 10.6.3 and 10.7.1 and the observation that the inverse function of the injection is an enumerator of X. ∎

Fact 22.2.4 (EWOs) Countable types have EWOs.

Proof Follows with an EWO for N (Fact 18.3.1) and Facts 22.2.3, 18.1.5, and 18.1.3. ∎

Fact 22.2.5 An injection $\mathcal{I}X\mathsf{N}$ can be raised into an injection $\mathcal{I}(\mathcal{O}X)\mathsf{N}$.

Proof Let f and g be the functions of $\mathcal{I}X\mathsf{N}$. Then

$$\begin{aligned} f'a &:= \text{MATCH}\ a\ [\,{}^\circ x \Rightarrow \mathsf{S}(fx) \mid \emptyset \Rightarrow 0\,] \\ g'n &:= \text{MATCH}\ n\ [\,0 \Rightarrow \emptyset \mid \mathsf{S}n \Rightarrow {}^\circ gn\,] \end{aligned}$$

yield an injection $\mathcal{I}(\mathcal{O}X)\mathsf{N}$. ∎

Fact 22.2.6
An injection $\mathcal{I}(\mathcal{O}X)Y$ with X inhabited can be lowered into an injection $\mathcal{I}XY$: $\mathcal{I}(\mathcal{O}X)Y \to X \to \mathcal{I}XY$.

Proof Let f and g be the functions of $\mathcal{I}(\mathcal{O}X)Y$ and x_0 an element of X. Then

$$\begin{aligned} f'x &:= f({}^\circ x) \\ g'y &:= \text{MATCH}\ gy\ [\,{}^\circ x \Rightarrow x \mid \emptyset \Rightarrow x_0\,] \end{aligned}$$

yield an injection $\mathcal{I}XY$. ∎

Fact 22.2.7 Nonempty countable types are exactly the retracts of N: $X \to (\mathsf{cty}\,X \Leftrightarrow \mathcal{I}X\mathsf{N})$.

Proof Follows with Facts 22.2.3, 22.2.5, and 22.2.6. ∎

Fact 22.2.8 (Uncountable type) The function type $\mathsf{N} \to \mathsf{B}$ is not countable.

Proof Suppose $\mathsf{N} \to \mathsf{B}$ is countable. Then Fact 22.2.7 give us an injection $\mathcal{I}(\mathsf{N} \to \mathsf{B})\mathsf{N}$ and thus a surjective function $\mathsf{N} \to (\mathsf{N} \to \mathsf{B})$. Contradiction with Cantor's theorem (Fact 8.3.4). ∎

22.3 Injection into N via List Enumeration

Infinite countable types appear frequently in computational settings. Typical examples are the syntactic types for expressions and for formulas appearing in Chapters 20 and 23. For these types constructing equality deciders is routine, but so far we don't have a method for constructing enumerators. In fact, constructing an enumerating function $\mathsf{N} \to \mathcal{O}(X)$ directly is not feasible since we need recursion on N but don't have the necessary termination arguments. We are now providing a method obtaining enumerators by constructing injections $\mathcal{I}X\mathsf{N}$.

The key idea is to have an infinite sequence $L_1, L_2, L_3, \dots$ of lists over X such that L_n is a prefix of $L_{\mathsf{S}n}$ and every x appears eventually is some L_n. Because of the prefix property, the first position of x in L_n does not dependent on the n as long as $x \in L_n$. For the retract, we now map x to its first position in some list L_n with $x \in L_n$, and n to the element at position n of a sufficiently long list L_m.

We define prefixes as follows: $\mathsf{prefix}\, A\, B \;:=\; \exists C.\ A + C = B$.

Fact 22.3.1 $\mathsf{prefix}\, A\, B \to k < \mathsf{len}\, A \to \mathsf{sub}\, A\, k = \mathsf{sub}\, B\, k$.

Proof $\forall k.\ k < \mathsf{len}\, A \to \mathsf{sub}\, A\, k = \mathsf{sub}\,(A + C)\, k$ follows by induction on A. ■

Theorem 22.3.2 (List enumeration)
An injection $\mathcal{I}X\mathsf{N}$ can be obtained from an equality decider for X and two functions $L^{\mathsf{N}\to\mathcal{L}(X)}$ and $\beta^{X\to\mathsf{N}}$ such that:

1. $\forall n.\ \ \mathsf{prefix}\, L_n\, L_{\mathsf{S}n} \wedge \mathsf{len}\, L_n < \mathsf{len}\, L_{\mathsf{S}n}$
2. $\forall x.\ \ x \in L_{\beta x}$

Proof Let X be a discrete type and L and β be functions as specified. Since L_1 is not empty by (1), we have a default value $x_0 : X$. We define functions $f^{X\to\mathsf{N}}$ and $g^{\mathsf{N}\to X}$

$$
\begin{aligned}
fx \;&:=\; \mathsf{pos}\, L_{\beta x}\, x \\
gn \;&:=\; \mathsf{sub}\, L_{\mathsf{S}n}\, n
\end{aligned}
$$

and show $\mathsf{inv}\, g f$. We prepare the proof with the following lemmas:

3. $m \le n \to \mathsf{prefix}\, L_m\, L_n$
4. $\mathsf{prefix}\, L_m\, L_n \vee \mathsf{prefix}\, L_n\, L_m$
5. $k < \mathsf{len}\, L_m \to k < \mathsf{len}\, L_n \to \mathsf{sub}\, L_m\, k = \mathsf{sub}\, L_n\, k$
6. $n \le \mathsf{len}\, L_n$

(3) follows by induction on n. (4) is a straightforward consequence of (3). (5) follows with Fact 22.3.1 and (4). (6) follows by induction on n and (1).

To show $\mathsf{inv}\, g f$, we fix x and set $k := \mathsf{pos}\, L_{\beta x}\, x$ and show $\mathsf{sub}\, L_{\mathsf{S}k}\, k = x$. By Fact 19.7.1(1) it suffices to show $\mathsf{sub}\, L_{\mathsf{S}k}\, k = \mathsf{sub}\, L_{\beta x}\, k$, which follows by (5) since $k < \mathsf{len}\, L_{\mathsf{S}k}$ by (6) and $k < \mathsf{len}\, L_{\beta x}$ by (2) and Fact 19.7.1(2). ■

Given a type X, we call functions L and β as specified by Theorem 22.3.2 a **list enumeration** of X. Note that types are infinite if they have a list enumeration.

22.4 More Countable Types

Using a list enumeration, we now show that the recursive inductive type

$$\mathsf{tree} ::= \mathsf{A}\,(\mathsf{N}) \mid \mathsf{T}\,(\mathsf{tree}, \mathsf{tree})$$

closing N under pairing is countable.

Fact 22.4.1 $\mathcal{E}(\mathsf{tree})$.

Proof Routine. By induction and case analysis on trees. ∎

To define a list enumeration for tree, we need a function that given a list A of trees returns a list containing all trees $\mathsf{T}st$ with $s, t \in A$.

Lemma 22.4.2 (List product)
$\forall AB^{\mathcal{L}(\mathsf{tree})}.\ \Sigma C.\ \ \forall u.\ u \in C \longleftrightarrow \exists s{\in}A\,\exists t{\in}B.\ u = \mathsf{T}st.$

Proof We fix B and prove the claim by induction on A following the scheme

$$\begin{aligned} [] \cdot B &= [] \\ (s :: A) \cdot B &= (\mathsf{T}s)@B + A \cdot B \end{aligned}$$

∎

Fact 22.4.3 $\mathcal{I}\,\mathsf{tree}\,\mathsf{N}$.

Proof By Theorem 22.3.2 and Fact 22.4.1 it suffices to construct a list enumeration for tree. We define:

$$\begin{aligned} L(0) &:= [] \\ L(\mathsf{S}n) &:= L(n) + \mathsf{A}\,n + \{\,\mathsf{T}(s,t) \mid s,t \in \mathsf{L}(n)\,\} \\ \beta(\mathsf{A}\,n) &:= \mathsf{S}n \\ \beta(\mathsf{T}\,st) &:= \mathsf{S}(\beta s + \beta t) \end{aligned}$$

where $\{\,\mathsf{T}(s,t) \mid s,t \in \mathsf{L}(n)\,\}$ is notation for the application of the list product function from Lemma 22.4.2 to $\mathsf{L}(n)$ and $\mathsf{L}(n)$. We show $\forall t.\ t \in L(\beta t)$ by induction on t, the rest is obvious.

For A, we show $\mathsf{A}\,n \in L(\mathsf{S}n)$, which is straightforward.

For T, we show $\mathsf{T}\,st \in L(\mathsf{S}(\beta s + \beta t))$. By induction we have $s \in L(\beta s)$ and $t \in L(\beta t)$. It suffices to show $s,t \in L(\beta s + \beta t)$. Follows with the monotonicity property

$$\forall mn.\ m \le n \to Lm \subseteq Ln$$

which in turn follows by induction on n. ∎

Fact 22.4.4 cty tree.

Proof Follows with Facts 22.4.3, 22.2.1(2a), and 22.2.1(1). ■

We can now show countability of a type by embedding it into tree. This is in many cases straightforward.

Fact 22.4.5 $\mathsf{cty}\,(\mathsf{N} \times \mathsf{N})$.

Proof By Facts 22.4.4 and 22.2.1(2a) it suffices to embed $\mathsf{N} \times \mathsf{N}$ into tree, which can be done as follows:

$$\begin{aligned} f(x,y) &:= \mathsf{T}(\mathsf{A}\,x)(\mathsf{A}\,y) \\ g(\mathsf{T}(\mathsf{A}\,x)(\mathsf{A}\,y)) &:= (x,y) \\ g\,\_ &:= (0,0) \end{aligned}$$

Now $g(f(x,y)) = (x,y)$ holds by computational equality. ■

Note that Fact 22.4.5 can also be obtained with Cantor pairing (Fact 10.6.9). The point is that Fact 22.4.5 obtains countability of $\mathsf{N} \times \mathsf{N}$ with a routine method rather than the ingenuous Cantor pairing.

Fact 22.4.6 $\mathsf{cty}\,(\mathcal{L}\,\mathsf{N})$.

Proof By Facts 22.4.4 and 22.2.1(2a) it suffices to embed $\mathcal{L}\,\mathsf{N}$ into tree, which can be done as follows:

$$\begin{aligned} f\,[] &:= \mathsf{A}\,0 \\ f\,(x :: A) &:= \mathsf{T}(\mathsf{A}\,x)(f A) \\ g(\mathsf{T}(\mathsf{A}\,x)\,t) &:= x :: g t \\ g\,\_ &:= [] \end{aligned}$$

Now $g(fA) = A$ follows by induction on A. ■

Fact 22.4.7 Countable types are closed under taking list types: $\mathsf{cty}\,X \to \mathsf{cty}(\mathcal{L}\,X)$.

Proof Let $\mathsf{cty}\,X$. The retract characterization (Fact 22.2.3) gives us functions $F^{\mathcal{O}X \to \mathsf{N}}$ and $G^{\mathsf{N} \to \mathcal{O}X}$ such that $\mathsf{inv}\,GF$. By Facts 22.4.6 and 22.2.1(2a) it suffices to embed $\mathcal{L}\,X$ into $\mathcal{L}\,\mathsf{N}$, which can be done as follows:

$$\begin{aligned} f\,A &:= (\lambda x.\, F(^\circ x))\,@\,A \\ g\,[] &:= [] \\ g\,(n :: A) &:= \textsc{match}\; Gn\; [\,^\circ x \Rightarrow x :: gA \mid \emptyset \Rightarrow []\,] \end{aligned}$$

Now $\forall A.\ g(fA) = A$ follows by induction on A. ■

Exercise 22.4.8 Prove $\mathcal{I}X\mathsf{N} \to \mathcal{I}(\mathcal{L}X)(\mathcal{L}\mathsf{N})$.

Exercise 22.4.9 Prove that the type of expressions (§ 20.1) and the type of formulas (§ 23.1) are countable by embedding them into tree. No recursion is needed for the embedding.

Exercise 22.4.10 (General list product) Assume a function $f^{X \to Y \to Z}$. Construct a function $\forall AB.\ \Sigma C.\ \forall z.\ z \in C \longleftrightarrow \exists x{\in}A\,\exists y{\in}B.\ z = fxy$.

22.5 Alignments

We start with an informal presentation of ideas we are going to formalize. An *alignment* of a type X is a nonrepeating sequence $x_0, x_1, x_2, \ldots$ listing all values of X. The sequence may be finite or infinite depending on the cardinality of X. Countable types have alignments since they have enumerators. If a type has an infinite alignment, it is in bijection with N. Moreover, a type is finite if and only if it has a finite alignment.

Formally, we define alignments as follows:

$$\begin{aligned} \mathsf{hit}_X\ f^{X\to\mathsf{N}}\ n &:= \exists x.\ fx = n \\ \mathsf{serial}_X\ f^{X\to\mathsf{N}} &:= \forall nk.\ \mathsf{hit}\ fn \to k \le n \to \mathsf{hit}\ fk \\ \mathsf{alignment}_X\ f^{X\to\mathsf{N}} &:= \mathsf{serial}\ f \wedge \mathsf{injective}\ f \end{aligned}$$

We call an alignment *infinite* if it is surjective. We call an alignment *finite* if it it has a cutoff n such that it hits exactly the numbers $k < n$. Formally, we define cutoffs as follows:

$$\mathsf{cutoff}_X\ f^{X\to\mathsf{N}}\ n \ := \ \forall k.\ \mathsf{hit}\ fk \longleftrightarrow k < n$$

Fact 22.5.1 Cutoffs are unique. That is, all cutoffs of a function $X \to \mathsf{N}$ agree.

Fact 22.5.2 The surjective alignments are exactly the bijective functions $X \to \mathsf{N}$.

Fact 22.5.3 (Bijection)
Countable types with surjective alignments are in bijection with N:
$\mathsf{cty}\,X \to \mathsf{alignment}_X\,f \to \mathsf{surjective}\,f \to \mathcal{B}X\mathsf{N}$.

Proof Follows with Fact 18.1.7 since countable types have EWOs (Fact 22.2.4). ∎

Fact 22.5.4 (Witnesses of Hits)
Witnesses of hits over countable types are computable:
$\mathsf{cty}\,X \to \mathsf{hit}_X\,fn \to \Sigma x.\ fx = n$.

Proof Immediate since countable types have EWOs and equality on N is decidable. ∎

Lemma 22.5.5 (Segment)
Let f be an alignment of a countable type. Then for every n hit by f one can obtain a nonrepeating list A of length $\mathsf{S}n$ such that $f@A$ is a segment.

Proof Let f be an alignment of a countable type X. Let $\mathsf{hit}\,fn$. By induction on n we construct a non-repeating list B of length $\mathsf{S}n$ such that $f@B$ is a segment.

Base case $n = 0$. Since countable types have EWOs, we can obtain x such that $fx = 0$. Then $A = [x]$ satisfies the claim.

Successor case. We assume $\mathsf{hit}\,f(\mathsf{S}n)$ and construct a nonrepeating list A of length $\mathsf{SS}n$ such that $f@A$ is a segment. Since f is serial, we have $\mathsf{hit}\,fn$. The inductive hypothesis gives us a nonrepeating list A of length $\mathsf{S}n$ such that $f@A$ is a segment. Since countable types have EWOs, we can obtain x such that $fx = \mathsf{S}n$. Now $x :: A$ satisfies the claim. We show $x \notin A$, the rest is obvious. Suppose $x \in A$. Then $fx < \mathsf{S}n$ contradicting $fx = \mathsf{S}n$. ∎

22.6 Alignment Construction

We will now show that countable types have alignments. The construction proceeds in two steps and starts from the enumerator of a countable type. The first step obtains a nonrepeating enumerator by removing repetitions.

An enumerator $f^{\mathsf{N}\to\mathcal{O}X}$ is **nonrepeating** if $\forall mn.\ fm = fn \neq \emptyset \to m = n$.

Fact 22.6.1 (Nonrepeating enumerator)
Every countable type has a nonrepeating enumerator:
$\mathsf{cty}\,X \to \Sigma f^{\mathsf{N}\to\mathcal{O}X}.\ \mathsf{enum}'\,f \wedge \mathsf{nonrepeating}\,f$.

Proof Let f be an enumerator of a countable type X. We obtain a nonrepeating enumerator of X by keeping for all x only the first n such that $fn = {}^\circ x$:

$$f'n := \begin{cases} \emptyset & \text{if } fn = \emptyset \\ {}^\circ x & \text{if } fn = {}^\circ x \wedge \mathsf{least}\,(\lambda n.fn = {}^\circ x)n \\ \emptyset & \text{if } fn = {}^\circ x \wedge \neg\mathsf{least}\,(\lambda n.fn = {}^\circ x)n \end{cases}$$

The definition of f' is admissible since $\mathcal{D}(\mathsf{least}\,(\lambda n.fn = {}^\circ x)n)$ since X is discrete and least preserves decidability (Fact 16.2.8). That f' enumerates X and is nonrepeating follows with Facts 16.2.7 and 16.1.1 for least. ∎

There is a second characterization of seriality that is useful for the second step of the alignment construction.

Fact 22.6.2 A function $f^{X\to\mathsf{N}}$ is serial if and only if $\forall n.\ \mathsf{hit}\,f(\mathsf{S}n) \to \mathsf{hit}\,fn$.

Proof One direction is trivial. For the other direction we assume $\forall n.\ \mathsf{hit}\,f(\mathsf{S}n) \to \mathsf{hit}\,fn$ and prove $\forall nk.\ \mathsf{hit}\,fn \to k \le n \to \mathsf{hit}\,fk$ by induction on n. ∎

Theorem 22.6.3 (Alignment)
Every countable type has an alignment:
$\forall X.\ \mathsf{cty}\, X \to \Sigma g^{X \to \mathsf{N}}.\ \mathsf{alignment}\, g.$

Proof Let X be a countable type. By Facts 22.6.1 and 22.1.3 X has a nonrepeating enumerator $f^{\mathsf{N} \to \mathcal{O}X}$ with a co-enumerator $g^{X \to \mathsf{N}}$. We define a function $h^{\mathsf{N} \to \mathsf{N}}$ such that hn is the number of hits f has for $k < n$:

$$h(0) := 0$$
$$h(\mathsf{S}n) := \text{IF } fn \neq \emptyset \text{ THEN } \mathsf{S}(hn) \text{ ELSE } hn$$

By induction on n we show two intuitively obvious facts about h:

1. $\forall kn.\ hn = \mathsf{S}k \to \Sigma my.\ m < n \wedge fm = {}^{\circ}y \wedge hm = k.$
2. $\forall mxn.\ fm = {}^{\circ}x \to m < n \to hm < hn.$

We now show that $g'x := h(gx)$ is an alignment of X.
That g' is serial follows with (1) and Fact 22.6.2.
That g' is injective follows with

3. $\forall xy.\ h(gx) = h(gy) \to gx = gy$

since the co-enumerator g is injective. To see (3), assume $h(gx) = h(gy)$. Then f hits both gx and gy. By (2) we have that $gx < gy$ and $gy < gx$ are both contradictory. Hence $gx = gy$. ∎

22.7 Bijection Theorem

We now show that two countable types are in bijection if they are retracts of each other: $\mathsf{cty}\, X \to \mathsf{cty}\, Y \to \mathcal{I}XY \to \mathcal{I}YX \to \mathcal{B}XY$.

Lemma 22.7.1 (Transport)
Let X and Y be countable types with alignments f and g and $\mathcal{I}XY$. Then:
$\forall x.\ \Sigma y.\ gy = fx.$

Proof Let $F^{X \to Y}$ and $G^{Y \to X}$ be such that $\forall x.\ G(Fx) = x$. We fix x and show $\Sigma y.\ gy = fx$. By Fact 22.5.4 it suffices to show that g hits fx. With the Segment Lemma 22.5.5 we obtain a nonrepeating list $A : \mathcal{L}(X)$ of length $\mathsf{S}(fx)$. Now $g@(F@A)$ has length $\mathsf{S}(fx)$ and is nonrepeating since g and F are injective. The Large Element Lemma 19.6.6 gives $k \in g@(F@A)$ such that $k \geq fx$. Thus g hits k. Since g is serial and $fx \leq k$, g hits fx. ∎

Theorem 22.7.2 (Bijection)
Countable types are in bijection if they are retracts of each other:
$\mathsf{cty}\,X \to \mathsf{cty}\,Y \to \mathcal{I}XY \to \mathcal{I}YX \to \mathcal{B}XY$.

Proof X and Y have alignments f and g by Fact 22.6.3. With the Transport Lemma 22.7.1 we obtain functions $F : \forall x.\, \Sigma y.\, gy = fx$ and $G : \forall y. \Sigma x. fx = gy$. Using the injectivity of f and g one verifies that $\lambda x.\pi_1(Fx)$ and $\lambda y.\pi_1(Gy)$ form a bijection. ■

Corollary 22.7.3 Infinite countable types are in bijection with N:
$\mathcal{I}\mathsf{N}X \to \mathsf{cty}\,X \to \mathcal{B}X\mathsf{N}$.

Proof Let $\mathcal{I}\mathsf{N}X$ and $\mathsf{cty}\,X$. Then X is inhabited and thus $\mathcal{I}X\mathsf{N}$ by Fact 22.2.7. Now the bijection theorem 22.7.2 gives us a bijection $\mathcal{B}X\mathsf{N}$ since $\mathsf{cty}\,\mathsf{N}$. ■

22.8 Alignments of Finite Types

We will show that a countable type X is finite with cardinality n if and only if n is the cutoff of some alignment of X.

Fact 22.8.1 Let n be the cutoff of an alignment of a countable type X. Then $\mathsf{fin}_n\,X$.

Proof Let X be a countable type, f be an alignment of X, and n be the cutoff of f. It suffices to show that X has a listing of length n.

If $n = 0$, then X is empty and $[]$ is a listing as required.

Now assume $n > 0$. Then f hits $n - 1$. The Segment Lemma 22.5.5 gives us a nonrepeating list A of length n such that $f@A$ contains exactly the numbers $k < n$. We assume $x : X$ and show that $x \in A$. Since n is a cutoff of f, we have $fx < n$. Hence $fx \in f@A$. Since f is injective, we have $x \in A$. ■

Fact 22.8.2 If $\mathsf{fin}_n\,X$, then n is a cutoff of every alignment of X.

Proof Let $\mathsf{fin}_n\,X$ and f be an alignment of X. Let A be a listing of X. Then $f@A$ is a nonrepeating list of length n containing exactly the numbers hit by f. Thus it suffices to show that $f@A$ contains exactly the numbers $k < n$. Follows with Fact 19.6.7 since $f@A$ is serial and nonrepeating. ■

Corollary 22.8.3 Alignments of finite types are not surjective.

Proof Let $\mathsf{fin}_n\,X$ and f be a alignment of X. It suffices to show that f doesn't hit n. By Fact 22.8.2 we have a cutoff n of f. Thus $n < n$ if f hits n. ■

22.9 Finite or Infinite

Assuming the law of excluded middle, we show that a serial function either has a cutoff or is surjective. Since there is no algorithm deciding this property, assuming excluded middle seems necessary.

Fact 22.9.1 $\mathsf{XM} \to \mathsf{serial}\, f \to \mathsf{ex}(\mathsf{cutoff}\, f) \vee \mathsf{surjective}\, f$.

Proof Assume XM and let f be serial. Using XM, we assume that f is not surjective and show that f has a cutoff. Using XM, we obtain a k not hit by f. Using XM and Fact 16.4.3, we obtain the least n not hit by f. Since f is serial, n is the cutoff of f. ∎

We now have that under XM a countable type is either finite or in bijection with N.

Fact 22.9.2 $\mathsf{XM} \to \mathsf{cty}\, X \to \Box\, (\mathsf{fin}\, X + \mathsf{bijection}\, X\, \mathsf{N})$.

Proof Follows with Facts 22.6.3, 22.9.1, 22.8.1, and 22.5.3. ∎

The truncation $\Box$ in Fact 22.9.2 is needed so that the disjunctive assumption XM can be harvested. Truncations are introduced in § 10.9.

Fact 22.9.3 $\mathsf{XM} \to \mathsf{cty}\, X \to \Box\, \mathcal{I}\, X \mathsf{N} \vee (X \to \bot)$.

Proof Follows with Facts 22.9.2, 21.2.2 and 21.4.8. ∎

22.10 Discussion

We have shown that a countable type is either a finite type or an infinite type that is in bijection with N. In fact, most results in this chapter generalize results we have already seen for finite types. The generalized results include the construction of alignments for countable types and the construction of bijections for countable types of the same cardinality.

Cantor pairing is an ingenuous construction establishing $\mathsf{N} \times \mathsf{N}$ as an infinite countable type. We gave a more general construction (list enumeration) that works for $\mathsf{N} \times \mathsf{N}$ and for syntactic types in general.

We did not give a formal definition of infinite types since there are several possibilities: We may call a type X infinite if X is not a finite type, if there is an injection $\mathcal{I}\mathsf{N}X$, or if there is a new element generator, among other possibilities.

As it comes to cardinalities of countable types, we may say that the cardinality of a countable type is a value of $\mathcal{O}(\mathsf{N})$, where $^\circ n$ represents the finite cardinality n and $\emptyset$ represents the single infinite cardinality.

Part III

Case Studies

23 Propositional Deduction

In this chapter we study propositional deduction systems. Propositional deduction systems can be elegantly formalized with indexed inductive type families. The chapter is designed such that it can serve as an introduction to propositional deduction systems and to indexed inductive type definitions at the same time. No previous knowledge of indexed inductive type definitions is assumed.

We present ND systems and Hilbert systems for intuitionistic provability and for classical provability. We show the equivalence of the respective systems and that classical provability reduces to intuitionistic provability (Glivenko's theorem). We consider a three-valued Heyting interpretation and the two-valued boolean interpretation and show that certain formulas are unprovable in the systems (e.g., the double negation law in intuitionistic systems).

We characterize classical provability with a refutation system based on boolean formula decomposition. The refutation system provides the basis for a certifying solver, from wich we obtain that classical provability is decidable and agrees with boolean entailment. We construct the certifying solver using size recursion.

The chapter can serve as an introduction to deduction systems in general, preparing the study of deduction systems for programming languages (e.g., type systems, operational semantics). More specifically, in this chapter we learn how inductive types elegantly represent abstract syntax, judgments, and derivation rules.

23.1 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 *judgments* $A \vdash s$ consisting of a list of formulas A serving as *assumptions* and a single formula s serving as *conclusion*. That a judgment $A \vdash s$ 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 judgments we cannot derive?
- *Weakening property:* Given a derivation of $A \vdash s$ and a list B containing A, can we always obtain a derivation of $B \vdash s$?
- *Cut property:* Given derivations of $A \vdash s$ and $s :: A \vdash t$, can we always obtain a derivation of $A \vdash t$?
- *Decidability:* Is it decidable whether a judgment $A \vdash s$ is derivable?

We will consider the following type of **formulas**:

$$s, t, u, v : \mathsf{For} \;:=\; x \mid \bot \mid s \to t \mid s \wedge t \mid s \vee t \qquad (x : \mathsf{N})$$

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 familiar notation

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

to express *negated formulas*.

Exercise 23.1.1 (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 a certifying equality decider for formulas.

23.2 Intuitionistic ND System

The deduction rules of the intuitionistic ND system we will consider are given in Figure 23.1 using several notational gadgets:

- *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 $A \vdash t$ from derivations of $A \vdash (s \to t)$ and $A \vdash s$. We say that the rule has two *premises* and one *conclusion*.

[1] The use of abstract syntax is discussed more carefully in Chapter 20.

$$\text{A}\ \frac{s \in A}{A \vdash s} \qquad \text{E}_\bot\ \frac{A \vdash \bot}{A \vdash s} \qquad \text{I}_\to\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \text{E}_\to\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

$$\text{I}_\wedge\ \frac{A \vdash s \qquad A \vdash t}{A \vdash s \wedge t} \qquad \text{E}_\wedge\ \frac{A \vdash s \wedge t \qquad A, s, t \vdash u}{A \vdash u}$$

$$\text{I}^1_\vee\ \frac{A \vdash s}{A \vdash s \vee t} \qquad \text{I}^2_\vee\ \frac{A \vdash t}{A \vdash s \vee t} \qquad \text{E}_\vee\ \frac{A \vdash s \vee t \qquad A, s \vdash u \qquad A, t \vdash u}{A \vdash u}$$

Figure 23.1: Deduction rules of the intuitinistic ND system

All rules in Figure 23.1 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 23.1 can derive exactly those judgments $A \vdash s$ 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 23.1 can derive the judgment $A \vdash s$ (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 23.1.

The rules in Figure 23.1 with a *logical constant* (i.e., $\bot$, $\to$, $\wedge$, $\vee$) in the conclusion are called **introduction rules**, and the rules with a logical constant in the leftmost premise are called **elimination rules**. The first rule in Figure 23.1 is known as **assumption rule**. Note that every rule but the assumption rule is an introduction or an elimination rule for some logical constant. Also note that there is no introduction rule for $\bot$, and that there are two introduction rules for $\vee$. The elimination rule for $\bot$ is also known as **explosion rule**.

Note that no deduction rule contains more than one logical constant. This results in an important modularity property. If we want to omit a logical constant, for instance $\wedge$, we just omit all rules containing this constant. Note that every system with $\bot$ and $\to$ can express negation. When trying to understand the structural properties of the system, it is usually a good idea to just consider $\bot$ and $\to$. Note that the assumption rule cannot be omitted since it is the only rule not taking a derivation as premise.

Here are common conveniences for the turnstile notation we will use in the following:

$$\begin{aligned} s \vdash u \quad &\rightsquigarrow \quad [s] \vdash u \\ s, t \vdash u \quad &\rightsquigarrow \quad [s, t] \vdash u \\ \vdash u \quad &\rightsquigarrow \quad [\,] \vdash u \end{aligned}$$

Example 23.2.1 Below is a **derivation** for $s \vdash \neg\neg s$ depicted as a **derivation tree**:

$$\dfrac{\dfrac{\dfrac{}{s, \neg s \vdash \neg s}\,\mathsf{A} \qquad \dfrac{}{s, \neg s \vdash s}\,\mathsf{A}}{s, \neg s \vdash \bot}\,\mathsf{E}_{\rightarrow}}{s \vdash \neg\neg s}\,\mathsf{I}_{\rightarrow}$$

The labels A, $\mathsf{E}_{\rightarrow}$, and $\mathsf{I}_{\rightarrow}$ at the right of the lines are the names for the rules used (assumption, elimination, and introduction).

Constructing ND derivations

Generations of students have been trained to construct ND derivations. In fact, constructing derivations in the intuitionistic ND system is pleasant if one follows the following recipe:

1. Construct a proof diagram as if the formulas were propositions.
2. Translate the proof diagram into a derivation (using the proof assistant).

Step 1 is the more difficult one, but you already well-trained as it comes to constructing intuitionistic proof diagrams. Once the proof assistant is used, constructing derivations becomes fun. Using the proof assistant becomes possible once the relevant ND system is realized as an inductive type.

The proof assistant comes with a decision procedure for intuitionistically provable quantifier-free propositions. If in doubt whether a certain derivation can be constructed in the intuitionistic ND system, the decision procedure of the proof assistant can readily decide the question.

Exercise 23.2.2 Give derivation trees for $A \vdash (s \rightarrow s)$ and $\neg\neg\bot \vdash \bot$.

Exercise 23.2.3 If you are eager to construct more derivations, Exercise 23.3.3 will provide you with interesting examples.

23.3 Formalisation with Indexed Inductive Type Family

It turns out that propositional deduction systems like the one in Figure 23.2 can be formalized elegantly and directly with inductive type definitions accommodating deduction rules as value constructors of derivation types $A \vdash s$.

Let us explain this fundamental idea. We may see the deduction rules in Figure 23.1 as functions that given derivations for the judgments in the premises yield a derivation for the judgment appearing as conclusion. The introduction rule for conjunctions, for instance, may be seen as a function that given derivations for $A \vdash s$ and $A \vdash t$ yields a derivation for $A \vdash s \wedge t$. We now go one step further

$$
\begin{array}{rl}
s \in A \;\to\; A \vdash s & \mathsf{A} \\
A \vdash \bot \;\to\; A \vdash s & \mathsf{E}_\bot \\
A, s \vdash t \;\to\; A \vdash (s \to t) & \mathsf{I}_\to \\
A \vdash (s \to t) \;\to\; A \vdash s \;\to\; A \vdash t & \mathsf{E}_\to \\
A \vdash s \;\to\; A \vdash t \;\to\; A \vdash (s \wedge t) & \mathsf{I}_\wedge \\
A \vdash (s \wedge t) \;\to\; A, s, t \vdash u \;\to\; A \vdash u & \mathsf{E}_\wedge \\
A \vdash s \;\to\; A \vdash (s \vee t) & \mathsf{I}^1_\vee \\
A \vdash t \;\to\; A \vdash (s \vee t) & \mathsf{I}^2_\vee \\
A \vdash (s \vee t) \;\to\; A, s \vdash u \;\to\; A, t \vdash u \;\to\; A \vdash u & \mathsf{E}_\vee
\end{array}
$$

Prefixes for A, s, t, u omitted, constructor names given at the right

Figure 23.2: Value constructors for derivation types $A \vdash s$

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 judgment $A \vdash s$ 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 23.2. Note that the types of the constructors follow exactly the patterns of the deduction rules in Figure 23.1.

When we look at the target types of the constructors in Figure 23.2, it becomes clear that the argument s of the type constructor $A \vdash s$ is *not* a parameter since it is instantiated by the constructors for the introduction rules ($\mathsf{I}_\to$, $\mathsf{I}_\wedge$, $\mathsf{I}^1_\vee$, $\mathsf{I}^2_\vee$). Such nonparametric arguments of type constructors are called **indices**. In contrast, the argument A of the type constructor $A \vdash s$ is a parameter since it is not instantiated in the target types of the constructors. More precisely, the argument A is a *nonuniform* parameter of the type constructor $A \vdash s$ since it is instantiated in some argument types of some of the constructors ($\mathsf{I}_\to$, $\mathsf{E}_\wedge$, and $\mathsf{E}_\vee$).

We call inductive type definitions where the type constructor has indices **indexed inductive definitions**. Indexed inductive definitions can also introduce **indexed inductive predicates**. In fact, we alternatively could introduce $\vdash$ as an indexed inductive predicate and this way demote derivations from computational objects to proofs.

The suggestive BNF-style notation we have used so far to write inductive type definitions does not generalize to indexed inductive type definitions. So we will use an explicit format giving the type constructor together with the list its value constructors. Often, the format used in Figure 23.2 will be convenient.

Fact 23.3.1 (Double negation)

1. $\neg\neg\bot \vdash \bot$
2. $s \vdash \neg\neg s$
3. $(A \vdash \neg\neg\bot) \Leftrightarrow (A \vdash \bot)$

Proof See Example 23.2.1 and the remarks there after. ■

In §23.9 we will show that $\neg\neg s \vdash s$ is not derivable for some formulas s. In particular, $\neg\neg s \vdash s$ is not derivable if s is a variable. However, as the above proof shows, $\neg\neg s \vdash s$ is derivable for $s = \bot$. This fact will play an important role.

Fact 23.3.2 (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.

Exercise 23.3.3 Construct derivations as follows:

a) $A \vdash \neg\neg\bot \rightarrow \bot$
b) $A \vdash s \rightarrow \neg\neg s$
c) $A \vdash (\neg s \rightarrow \neg\neg\bot) \rightarrow \neg\neg s$
d) $A \vdash (s \rightarrow \neg\neg t) \rightarrow \neg\neg(s \rightarrow t)$
e) $A \vdash \neg\neg(s \rightarrow t) \rightarrow \neg\neg s \rightarrow \neg\neg t$
f) $A \vdash \neg\neg\neg s \rightarrow \neg s$
g) $A \vdash \neg s \rightarrow \neg\neg\neg s$

Exercise 23.3.4 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) $A, s, \neg t \vdash \bot \ \rightarrow\ A \vdash \neg\neg(s \rightarrow t)$

Hint: (c) is routine if you first show $A \vdash (\neg t \rightarrow \neg s) \rightarrow \neg\neg(s \rightarrow t)$.

Exercise 23.3.5 Prove the implicative facts (1)–(6) appearing in Exercise 23.11.6.

23.4 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 23.3. 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.

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 23.3, the critical argument of $\mathsf{E}_{\vdash}$ that needs to be written as '_' in the defining equations is s in the target type $\forall As.\ A \vdash s \to pAs$ of $\mathsf{E}_{\vdash}$.

23.5 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.

$$
\begin{aligned}
\mathsf{E}_\vdash :\ & \forall p^{\mathcal{L}(\mathsf{For})\to\mathsf{For}\to\mathbb{T}}.\\
& (\forall As.\ s \in A \to pAs) \to\\
& (\forall As.\ pA\bot \to pAs) \to\\
& (\forall Ast.\ p(s :: A)t \to pA(s \to t)) \to\\
& (\forall Ast.\ pA(s \to t) \to pAs \to pAt) \to\\
& (\forall Ast.\ pAs \to pAt \to pA(s \wedge t)) \to\\
& (\forall Astu.\ pA(s \wedge t) \to p(s :: t :: A)u \to pAu) \to\\
& (\forall Ast.\ pAs \to pA(s \vee t)) \to\\
& (\forall Ast.\ pAt \to pA(s \vee t)) \to\\
& (\forall Astu.\ pA(s \vee t) \to p(s :: A)u \to p(t :: A)u \to pAu) \to\\
& \forall As.\ A \vdash s \to pAs
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{E}_\vdash\, pe_1 \dots e_9\, A\, \_\, (\mathsf{A}\, sh) &:= e_1 Ash\\
(\mathsf{E}_\bot\, sd) &:= e_2 As(\mathsf{E}_\vdash \dots A\bot d)\\
(\mathsf{I}_\to\, std) &:= e_3 Ast(\mathsf{E}_\vdash \dots (s :: A)td)\\
(\mathsf{E}_\to\, std_1d_2) &:= e_4 Ast(\mathsf{E}_\vdash \dots A(s \to t)d_1)(\mathsf{E}_\vdash \dots Asd_2)\\
(\mathsf{I}_\wedge\, std_1d_2) &:= e_5 Ast(\mathsf{E}_\vdash \dots Asd_1)(\mathsf{E}_\vdash \dots Atd_2)\\
(\mathsf{E}_\wedge\, stud_1d_2) &:= e_6 Astu(\mathsf{E}_\vdash \dots A(s \wedge t)d_1)(\mathsf{E}_\vdash \dots (s :: t :: A)ud_2)\\
(\mathsf{I}_\vee^1\, std) &:= e_7 Ast(\mathsf{E}_\vdash \dots Asd)\\
(\mathsf{I}_\vee^2\, std) &:= e_8 Ast(\mathsf{E}_\vdash \dots Atd)\\
(\mathsf{E}_\vee\, stud_1d_2d_3) &:= e_9 Astu(\mathsf{E}_\vdash \dots A(s \vee t)d_1)\\
&\qquad (\mathsf{E}_\vdash \dots (s :: A)ud_2)\\
&\qquad (\mathsf{E}_\vdash \dots (t :: A)ud_3)
\end{aligned}
$$

Figure 23.3: Eliminator for $A \vdash s$

Fact 23.5.1 (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 23.5.2 (Implication) $A \vdash (s \to t) \iff A, s \vdash t$.

Proof Direction $\Leftarrow$ holds by $\mathsf{I}_{\to}$. For direction $\Rightarrow$ we assume $A \vdash (s \to t)$ and obtain $A, s \vdash (s \to t)$ with weakening. Now A and $\mathsf{E}_{\to}$ yield $A, s \vdash t$. ∎

As a consequence, we can represent all assumptions of a derivation $A \vdash s$ as premises of implications at the right-hand side. To this purpose, we define a *reversion function* $A \cdot s$ with $[] \cdot t := t$ and $(s :: A) \cdot t := A \cdot (s \to t)$. For instance, we have $[s_1, s_2, s_3] \cdot t = (s_3 \to s_2 \to s_1 \to t)$. To ease our notation, we will write $\vdash s$ for $[] \vdash s$.

Fact 23.5.3 (Reversion) $A \vdash s \iff \vdash A \cdot s$.

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 23.5.4 (Ground Prover) $\forall s.\ \mathsf{ground}\, s \to (\vdash s) + (\vdash \neg s)$.

Proof By induction on s using weakening. ∎

Exercise 23.5.5 Prove $\forall s.\ \mathsf{ground}\, s \to\ \vdash (s \vee \neg s)$.

Exercise 23.5.6 Prove $\forall A s.\ \mathsf{ground}\, s \to A, s \vdash t \ \to\ A, \neg s \vdash t \ \to\ A \vdash t$.

Exercise 23.5.7 Prove the deduction laws for conjunctions and disjunctions:

a) $A \vdash (s \wedge t) \iff A \vdash s \times A \vdash t$

b) $A \vdash (s \vee t) \iff \forall u.\ A, s \vdash u \ \to\ A, t \vdash u \ \to\ A \vdash u$

Exercise 23.5.8 Construct derivations for the following judgments:

a) $\vdash (t \to \neg s) \to \neg(s \wedge t)$

b) $\vdash \neg\neg s \to \neg\neg t \to \neg\neg(s \wedge t)$

c) $\vdash \neg s \to \neg t \to \neg(s \vee t)$

d) $\vdash (\neg t \to \neg\neg s) \to \neg\neg(s \vee t)$

e) $\vdash \neg\neg s \to \neg t \to \neg(s \to t)$

f) $\vdash (\neg t \to \neg s) \to \neg\neg(s \to t)$

Exercise 23.5.9 (Order-preserving reversion)
We define a reversion function $A \cdot s$ preserving the order of assumptions:

$$[] \cdot s \;:=\; s$$
$$(t :: A) \cdot s \;:=\; t \to (A \cdot s)$$

Prove $A \vdash s \Leftrightarrow\; \vdash A \cdot s$.
Hint: Prove the generalization $\forall B.\ B +\!\!+ A \vdash s \Leftrightarrow B \vdash A \cdot s$ by induction on A.

23.6 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 23.6.1 (Double Negation) $A \dot\vdash (\neg\neg s \to s)$.

Proof Straightforward using the contradiction rule. ∎

Fact 23.6.2 (Cut) $A \dot\vdash s \;\to\; A, s \dot\vdash t \;\to\; A \dot\vdash t$.

Proof Same as for the intuitionistic system. ∎

Fact 23.6.3 (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 23.6.4 (Explosion) $A \dot{\vdash} \bot \;\to\; A \dot{\vdash} s$.

Proof By contradiction and weakening. ∎

Fact 23.6.5 (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 23.6.6 (Implication) $A, s \dot{\vdash} t \;\Leftrightarrow\; A \dot{\vdash} (s \to t)$.

Proof Same proof as for the intuitionistic system. ∎

Fact 23.6.7 (Reversion) $A \dot{\vdash} s \Leftrightarrow\; \vdash A \cdot s$.

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 23.6.8 (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 23.6.9 Show $(A \vdash s \to t \to u) \iff (A \vdash t \to s \to u)$.

Exercise 23.6.10 Show $\dot{\vdash} s \vee \neg s$ and $\dot{\vdash} ((s \to t) \to s) \to s$.

Exercise 23.6.11 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 23.6.12 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.

23.7 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 23.7.1 $A \dot{\vdash} s \rightarrow A \vdash \neg\neg s$.

Proof By induction on $A \dot{\vdash} s$. This yields the following proof obligations (the obligations for conjunctions and disjunctions are omitted).

- $s \in A \rightarrow A \vdash \neg\neg s$
- $A, \neg s \vdash \neg\neg\bot \rightarrow A \vdash \neg\neg s$.
- $A, s \vdash \neg\neg t \rightarrow A \vdash \neg\neg(s \rightarrow t)$
- $A \vdash \neg\neg(s \rightarrow t) \rightarrow A \vdash \neg\neg s \rightarrow A \vdash \neg\neg t$

Using rule $\mathsf{E}_{\rightarrow}$ of the intuitionistic system, the obligations can be strengthened to:

- $\vdash s \rightarrow \neg\neg s$
- $\vdash (\neg s \rightarrow \neg\neg\bot) \rightarrow \neg\neg s$
- $\vdash (s \rightarrow \neg\neg t) \rightarrow \neg\neg(s \rightarrow t)$
- $\vdash \neg\neg(s \rightarrow t) \rightarrow \neg\neg s \rightarrow \neg\neg t$.

The proofs of the strengthened obligations are routine (Exercise 23.3.3). ■

Theorem 23.7.2 (Glivenko) $A \dot{\vdash} s \Leftrightarrow A \vdash \neg\neg s$.

Proof Direction $\Rightarrow$ follows with Lemma 23.7.1. Direction $\Leftarrow$ follows with translation (23.6.5) and double negation (23.6.1). ■

Corollary 23.7.3 (Agreement on negated formulas) $A \dot{\vdash} \neg s \Leftrightarrow A \vdash \neg s$.

Corollary 23.7.4 (Refutation agreement)
Intuitionistic and classical refutation agree: $A \vdash \bot \Leftrightarrow A \dot{\vdash} \bot$.

Proof Glivenko's theorem and the bottom law 23.3.1. ■

Corollary 23.7.5 (Equiconsistency)
Intuitionistic ND is consistent if and only if classical ND is consistent:
$((\vdash \bot) \rightarrow \bot) \iff ((\dot{\vdash} \bot) \rightarrow \bot)$.

Proof Immediate consequence of Corollary 23.7.4. ■

Exercise 23.7.6 We call a formula s **stable** if $\neg\neg s \vdash s$. Prove the following:

a) $\bot$ is stable.

b) If t is stable, then $s \rightarrow t$ is stable.

c) If s is stable, then $A \dot{\vdash} s \Leftrightarrow A \vdash s$.

23.8 Intuitionistic Hilbert System

Hilbert systems are deduction systems predating ND systems.[2] They are simpler than ND systems in that they come without assumption management. While it is virtually impossible for humans to write proofs in Hilbert systems, one can construct compilers translating derivations in ND systems into derivations in Hilbert systems.

To ease our presentation, we restrict ourselves in this section to formulas not containing conjunctions and disjunctions. Since implications are the primary connective in Hilbert systems and conjunctions and disjunctions appear as extensions, adding conjunctions and disjunctions will be an easy exercise.

We consider an intuitionistic Hilbert system formalized with an inductive type constructor $\mathcal{H} : \mathsf{For} \to \mathbb{T}$ and the derivation rules

$$\mathsf{H_{MP}}\ \frac{\mathcal{H}(s \to t) \qquad \mathcal{H}(s)}{\mathcal{H}(t)} \qquad\qquad \mathsf{H_K}\ \frac{}{\mathcal{H}(s \to t \to s)}$$

$$\mathsf{H_S}\ \frac{}{\mathcal{H}((s \to t \to u) \to (s \to t) \to s \to u)} \qquad\qquad \mathsf{H_\bot}\ \frac{}{\mathcal{H}(\bot \to s)}$$

There are a single two-premise rule called **modus ponens** and three premise-free rules called **axiomatic rules**. So all the action comes with modus ponens, which puts implication into the primary position. Note that the single argument of the type constructor $\mathcal{H}$ comes out as an index.

A Hilbert system internalizes the assumption list of the ND system using implication. It keeps the elimination rule for implications (now called modus ponens) but reformulates all other rules as axiomatic rules using implication. Surprisingly, only two rules ($\mathsf{H_K}$ and $\mathsf{H_S}$) suffice to simulate the assumption management and the introduction rule for implication. The axiomatic rules for conjunction and disjunction follow the ND rules and the translation scheme we see in the falsity elimination rule $\mathsf{H_\bot}$ and come with the following conclusions:

$$\begin{gathered} s \to t \to s \wedge t \\ s \wedge t \to (s \to t \to u) \to u \\ s \to s \vee t \\ t \to s \vee t \\ s \vee t \to (s \to u) \to (t \to u) \to u \end{gathered}$$

[2]Hilbert systems are also known as axiomatic systems. They originated with Gottlieb Frege before they were popularized by David Hilbert.

$$\mathsf{H}^{\Vdash}_{\mathsf{A}}\ \frac{s \in A}{A \Vdash s} \qquad \mathsf{H}^{\Vdash}_{\mathsf{MP}}\ \frac{A \Vdash s \to t \qquad A \Vdash s}{A \Vdash t} \qquad \mathsf{H}^{\Vdash}_{\mathsf{K}}\ \frac{}{A \Vdash s \to t \to s}$$

$$\mathsf{H}^{\Vdash}_{\mathsf{S}}\ \frac{}{A \Vdash (s \to t \to u) \to (s \to t) \to s \to u} \qquad \mathsf{H}^{\Vdash}_{\bot}\ \frac{}{A \Vdash \bot \to s}$$

Figure 23.4: Generalized Hilbert system $\Vdash : \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{T}$

We will prove that $\mathcal{H}$ derives exactly the formulas intuitionistic ND derives in the empty context (that is, $\mathcal{H}s \Leftrightarrow \vdash s$). One direction of the proof is straightforward.

Fact 23.8.1 (Soundness for ND) $\mathcal{H}(s) \to ([] \vdash s)$.

Proof By induction on the derivation of $\mathcal{H}(s)$. The modus ponens rule can be simulated with $\mathsf{E}_{\to}$, and the conclusions of the axiomatic rules are all easily derivable in the intuitionistic system. ∎

The other direction of the equivalence proof (completeness for ND) is challenging since it has to internalize the assumption management of the ND system. We will see that this can be done with the axiomatic rules H_{K} and H_{S}. We remark that the conclusions of H_{K} and H_{S} may be seen as types for the functions $\lambda xy.x$ and $\lambda fgx.(fx)(gx)$.

The completeness proof uses the generalized Hilbert system $\Vdash$ shown in Figure 23.4 as an intermediate system. Similar to the ND system, the generalized Hilbert system maintains a context, but this time no rule modifies the context. The assumption rule $\mathsf{H}^{\Vdash}_{\mathsf{A}}$ is the only rule reading the context. The context can thus be accommodated as a uniform parameter of the type constructor $\Vdash$.

Fact 23.8.2 (Agreement) $\mathcal{H}(s) \longleftrightarrow [] \Vdash s$.

Proof Both directions are straightforward inductions. ∎

It remains to construct a function translating ND derivations $A \vdash s$ into Hilbert derivations $A \Vdash s$. For this we use a simulation function for every rule of the ND system (Figure 23.1). The simulation functions are obvious for all rules of the ND system but for $\mathsf{I}_{\to}$.

Fact 23.8.3 (Basic simulation functions)

1. $\forall As.\ s \in A \to A \Vdash s$.
2. $\forall Ast.\ (A \Vdash s \to t) \to (A \Vdash s) \to (A \Vdash t)$.
3. $\forall As.\ (A \Vdash \bot) \to (A \Vdash s)$.

Proof Functions (1) and (2) are exactly $H_A^{\Vdash}$ and $H_{MP}^{\Vdash}$. Function (3) can be obtained with $H_\bot^{\Vdash}$ and $H_{MP}^{\Vdash}$. ■

The translation function for $I_\to$ needs several auxiliary functions.

Fact 23.8.4 (Operational versions of K and S)

1. $\forall Asu.\ (A \Vdash u) \to (A \Vdash s \to u)$.
2. $\forall Astu.\ (A \Vdash s \to t \to u) \to (A \Vdash s \to t) \to (A \Vdash s \to u)$.

Proof (1) follows with $H_K^{\Vdash}$ and $H_{MP}^{\Vdash}$. (2) follows with $H_S^{\Vdash}$ and $H_{MP}^{\Vdash}$. ■

Fact 23.8.5 (Identity) $\forall As.\ A \Vdash s \to s$.

Proof Follows with the operational version of S (with $s := s$, $t := s \to s$, and $u := s$) using $H_K^{\Vdash}$ for both premises. ■

The next fact is the heart of the translation of ND derivations into Hilbert derivations. It is well-known in the literature under the name *deduction theorem*.

Fact 23.8.6 (Simulation function for $I_\to$) $\forall Ast.\ (A, s \Vdash t) \to (A \Vdash s \to t)$.

Proof By induction on the derivation $A, s \Vdash t$ (the context argument of $\Vdash$ is a uniform parameter).

- $H_A^{\Vdash}$. If $s = t$, the claim follows with Fact 23.8.5. If $t \in A$, the claim follows with $H_A^{\Vdash}$ and the operational version of K (Fact 23.8.4(1)). The case distinction is possible since equality of formulas is decidable.
- $H_{MP}^{\Vdash}$. Follows with the operational version of S (Fact 23.8.4(2)) and the inductive hypotheses.
- $H_K^{\Vdash}$, $H_S^{\Vdash}$, $H_\bot^{\Vdash}$. The axiomatic cases follow with the operational version of K (Fact 23.8.4(1)) and $H_K^{\Vdash}$, $H_S^{\Vdash}$, $H_\bot^{\Vdash}$, rspectively. ■

Fact 23.8.7 (Completeness for ND) $(A \vdash s) \to (A \Vdash s)$.

Proof By induction on the derivation of $A \vdash s$ using Facts 23.8.3 and 23.8.6. ■

Theorem 23.8.8 (Agreement) $\mathcal{H}(s) \Leftrightarrow \vdash s$.

Proof Follows with Facts 23.8.1, 23.8.7, and 23.8.2. ■

Exercise 23.8.9 Show $(A \Vdash s) \Leftrightarrow (A \vdash s)$.

Exercise 23.8.10 Extend the development of this section to formulas with conjunctions and disjunctions. Add the axiomatic rules shown at the beginning of §23.8.

Exercise 23.8.11 Define a classical Hilbert system and show its equivalence with the classical ND system. Do this by replacing the axiomatic rule for $\bot$ with an axiomatic rule providing the double negation law $\neg\neg s \to s$.

23.9 Heyting Evaluation

The proof techniques we have seen 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 falsity, atomic formulas, and the double negation law for atomic formulas are not intuitionistically derivable.

A powerful technique for showing negative results is evaluation of formulas into a finite and ordered domain of so-called *truth values*. Things are arranged such that all derivable formulas evaluate under all assignments to the largest truth value.[3] A formula can then be established as underivable by presenting an assignment under which the formula evaluates to a different truth value.

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 (i.e., 0). 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:

$$\mathsf{imp}\ ab \;:=\; \text{IF } a \leq b \text{ THEN } 2 \text{ ELSE } b$$

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.

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 $a \leq b$. As a matter of convenience, we 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:

$$\mathcal{E} : (\mathsf{N} \to \mathsf{V}) \to \mathsf{For} \to \mathsf{V}$$

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= 0 \\
\mathcal{E}\alpha(s \to t) &:= \text{IF } \mathcal{E}\alpha s \leq \mathcal{E}\alpha t \text{ THEN } 2 \text{ ELSE } \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \wedge t) &:= \text{IF } \mathcal{E}\alpha s \leq \mathcal{E}\alpha t \text{ THEN } \mathcal{E}\alpha s \text{ ELSE } \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \vee t) &:= \text{IF } \mathcal{E}\alpha s \leq \mathcal{E}\alpha t \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathcal{E}\alpha s
\end{aligned}$$

[3] An assignment assigns a truth value to every atomic formula.

Note that conjunction is interpreted as minimum, disjunction is interpreted as maximum, and implications is interpreted as described above.

We will show that all formulas derivable in the Hilbert system $\mathcal{H}$ defined in §23.8 evaluate under all assignments to the largest truth value 2:

$$\forall \alpha s.\ \ \mathcal{H}(s) \to \mathcal{E}\alpha s = 2$$

For the proof we fix an assignment α and say that a formula s is true if $\mathcal{E}\alpha s = 2$. Next we verify that the conclusions of all axiomatic rules (see §23.8) are true, which follows by case analysis on the truth values $\mathcal{E}\alpha s$, $\mathcal{E}\alpha t$, and $\mathcal{E}\alpha u$. It remains to show that modus ponens derives true formulas from true formulas, which again follows by case analysis on the truth values $\mathcal{E}\alpha s$ and $\mathcal{E}\alpha t$.

Fact 23.9.1 (Soundness) $\forall \alpha s.\ \mathcal{H}(s) \to \mathcal{E}\alpha s = 2$.

Proof By induction on the derivation $\mathcal{H}(s)$. The cases for the axiomatic rules follow by case analysis on the truth values $\mathcal{E}\alpha s$, $\mathcal{E}\alpha t$, and $\mathcal{E}\alpha u$. The case for modus ponens follows by the inductive hypotheses and case analysis on the truth values $\mathcal{E}\alpha s$ and $\mathcal{E}\alpha t$. ∎

Corollary 23.9.2 (Soundness) $\vdash s \to \mathcal{E}\alpha s = 2$.

Proof Fact 23.9.1 and Theorem 23.8.8. ∎

With our definitions we have the computational equalities

$$\begin{aligned} \mathcal{E}(\lambda\_.1)\bot &= 0 \\ \mathcal{E}(\lambda\_.1)x &= 1 \\ \mathcal{E}(\lambda\_.1)(\neg x) &= 0 \\ \mathcal{E}(\lambda\_.1)(\neg\neg x) &= 2 \\ \mathcal{E}(\lambda\_.1)(\neg\neg x \to x) &= 1 \end{aligned}$$

Thus, with soundness, we can now disprove $\vdash \bot$, $\vdash x$, and $\vdash (\neg\neg x \to x)$.

A formula s is **independent in** $\vdash$ if one can prove both $(\vdash s) \to \bot$ and $(\vdash \neg s) \to \bot$.

Corollary 23.9.3 (Independence) x, $\neg\neg x \to x$ and $x \vee \neg x$ are independent in $\vdash$.

Proof Follows with Corollary 23.9.2 and the assignment $\lambda\_.1$. ∎

Corollary 23.9.4 (Consistency) $\nvdash \bot$ and $\not\dot\vdash \bot$.

Proof Intuitionistic consistency follows with Corollary 23.9.2 and the assignment $\lambda\_.1$. Classic consistency follows with equiconsistency (Corollary 23.7.5). ∎

Exercise 23.9.5 Show that x, $\neg x$, and $(x \to y) \to x) \to x$ are independent in $\vdash$.

Exercise 23.9.6 Show $\neg\forall s.\, ((\vdash (\neg\neg s \to s)) \to \bot)$.

Exercise 23.9.7 Show that classical ND is not sound for the Heyting interpretation: $\neg(\forall \alpha s.\, \dot{\vdash} s \to \mathcal{E}\alpha s = 2)$.

Exercise 23.9.8 Disprove $\dot{\vdash} x$ and $\dot{\vdash} \neg x$.

Exercise 23.9.9 Disprove $\dot{\vdash}(s \vee t) \Leftrightarrow \dot{\vdash} s \vee A \dot{\vdash} t$.

Exercise 23.9.10 (Heyting interpretation for ND system) One can define evaluation of contexts such that $(A \vdash s) \to \mathcal{E}\alpha A \leq \mathcal{E}\alpha s$ and $\mathcal{E}\alpha[] = 2$.

a) Define evaluation of contexts as specified.

b) Show $\mathcal{E}\alpha A \leq \mathcal{E}\alpha s \to A = [] \to \mathcal{E}\alpha s = 2$.

c) Prove $(A \vdash s) \to \mathcal{E}\alpha A \leq \mathcal{E}\alpha s$ by induction on $A \vdash s$.

Hint: Define evaluation of contexts such that contexts may be seen as conjunctions of formulas.

Exercise 23.9.11 (Diamond Heyting interpretation) The formulas

$$\neg x \vee \neg\neg x$$
$$(x \to y) \vee (y \to x)$$

evaluate in our Heyting interpretation to 2 but are unprovable intuitionistically. They can be shown unprovable with a 4-valued diamond-ordered

$$\bot < a, b < \top$$

Heyting interpretation as follows:

- $x \wedge y$ is the infimum of x and y.
- $x \vee y$ is the supremum of x and y.
- $x \to y$ is the maximal z such that $x \wedge z \leq y$.

a) Verify $(\neg a \vee \neg\neg a) = \bot$

b) Verify $((a \to b) \vee (b \to a)) = \bot$.

c) Prove $\mathcal{H}(s) \to \mathcal{E}\alpha s = \top$.

To know more, google *Heyting algebras*.

23.10 Boolean Evaluation

We define **boolean evaluation** of formulas following familiar ideas:

$$\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathsf{B}$$

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= \mathbf{F} \\
\mathcal{E}\alpha(s \to t) &:= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathbf{T} \\
\mathcal{E}\alpha(s \wedge t) &:= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathcal{E}\alpha t \text{ ELSE } \mathbf{F} \\
\mathcal{E}\alpha(s \vee t) &:= \text{IF } \mathcal{E}\alpha s \text{ THEN } \mathbf{T} \text{ ELSE } \mathcal{E}\alpha t
\end{aligned}$$

We call functions $\alpha : \mathsf{N} \to \mathsf{B}$ **boolean assignments**.

Boolean evaluation may be seen as a special Heyting evaluation with only two truth values $\mathbf{F} < \mathbf{T}$.

We define

$\mathsf{sat}\,\alpha\,s$	$:=$	$\mathcal{E}\alpha s$	**α satisfies s**
$\mathsf{sat}\,\alpha\,A$	$:=$	$\forall s \in A.\ \mathsf{sat}\,\alpha s$	**α satisfies A**
$\mathsf{sat}\,A$	$:=$	$\Sigma\alpha.\ \mathsf{sat}\,\alpha\,A$	**A is satisfiable**

It will be convenient to use the word **clause** for lists of formulas. It is well known that boolean satisfiability of clauses is decidable. There exist various practical tools for deciding boolean satisfiability. We will develop a certifying decider $\forall A.\, \mathcal{D}(\mathsf{sat}\,A)$ for satisfiability and refine it into a certifying decider $\forall As.\, \mathcal{D}(A \dot{\vdash} s)$ for classical ND.

23.11 Boolean Formula Decomposition

Our decider $\forall A.\, \mathcal{D}(\mathsf{sat}\,A)$ for boolean satisfiability will be based on boolean formula decomposition. We describe **boolean formula decomposition** with the **decomposition table** in Figure 23.5. One way to read the table is saying that a boolean assignment satisfies the formula on the left if and only if it satisfies both or one of

$\neg\bot$	nothing
$s \wedge t$	s and t
$\neg(s \wedge t)$	$\neg s$ or $\neg t$
$s \vee t$	s or t
$\neg(s \vee t)$	$\neg s$ and $\neg t$
$s \to t$	$\neg s$ or t
$\neg(s \to t)$	s and $\neg t$

Figure 23.5: Boolean decomposition table

the possibly negated subformulas on the right. Formally we have the equivalences

$$\begin{aligned}
\mathcal{E}\alpha(s \wedge t) = \mathbf{T} &\longleftrightarrow \mathcal{E}\alpha(s) = \mathbf{T} \;\wedge\; \mathcal{E}\alpha(t) = \mathbf{T} \\
\mathcal{E}\alpha(\neg(s \wedge t)) = \mathbf{T} &\longleftrightarrow \mathcal{E}\alpha(\neg s) = \mathbf{T} \;\vee\; \mathcal{E}\alpha(\neg t) = \mathbf{T} \\
\mathcal{E}\alpha(s \vee t) = \mathbf{T} &\longleftrightarrow \mathcal{E}\alpha(s) = \mathbf{T} \;\vee \mathcal{E}\alpha(t) = \mathbf{T} \\
\mathcal{E}\alpha(\neg(s \vee t)) = \mathbf{T} &\longleftrightarrow \mathcal{E}\alpha(\neg s) = \mathbf{T} \;\wedge\; \mathcal{E}\alpha(\neg t) = \mathbf{T} \\
\mathcal{E}\alpha(s \to t) = \mathbf{T} &\longleftrightarrow \mathcal{E}\alpha(\neg s) = \mathbf{T} \;\vee \mathcal{E}\alpha(t) = \mathbf{T} \\
\mathcal{E}\alpha(\neg(s \to t)) = \mathbf{T} &\longleftrightarrow \mathcal{E}\alpha(s) = \mathbf{T} \;\wedge\; \mathcal{E}\alpha(\neg t) = \mathbf{T} \\
\mathcal{E}\alpha(\neg\bot) = \mathbf{T} &\longleftrightarrow \top
\end{aligned}$$

for all boolean assignment α. The equivalences follow with the de Morgan laws

$$\begin{aligned}
\mathcal{E}\alpha(\neg(s \wedge t)) &= \mathcal{E}\alpha(\neg s \vee \neg t) \\
\mathcal{E}\alpha(\neg(s \vee t)) &= \mathcal{E}\alpha(\neg s \wedge \neg t)
\end{aligned}$$

and the implication and double negation laws:

$$\begin{aligned}
\mathcal{E}\alpha(s \to t) &= \mathcal{E}\alpha(\neg s \vee t) \\
\mathcal{E}\alpha(\neg\neg s) &= \mathcal{E}\alpha(s)
\end{aligned}$$

The decomposition table suggests an algorithm that given a list of formulas replaces decomposable formulas with smaller formulas. This way we obtain from an initial list A one or several *decomposed lists* $A_1, \dots, A_n$ containing only formulas of the forms

$$x, \quad \neg x, \quad \bot$$

such that an assignment satisfies the initial list A if and only if it satisfies one of the decomposed lists $A_1, \dots, A_n$. We may get more than one decomposed list since the decomposition rules for $\neg(s \wedge t)$, $s \vee t$ and $s \to t$ are *branching* (see Figure 23.5). For a decomposed list, we can either construct an assignment satisfying all its formulas, or prove that no such satisfying assignment exists. Put together, this will give us a certifying decider $\forall A.\ \mathcal{D}(\mathsf{sat}\,A)$.

We are now facing the challenge to give a formal account of boolean formula decomposition. We do this with two derivation systems $\sigma(A)$ and $\rho(A)$ for clauses shown in Figures 23.6 and 23.7. Like the derivation systems we have seen before, $\sigma(A)$ and $\rho(A)$ can be formalized as inductive type families $\mathcal{L}(\mathsf{For}) \to \mathbb{T}$. We will see that σ derives all satisfiable clauses and ρ derives all unsatisfiable clauses. We define the application conditions for the terminal rules as follows:

- $\mathsf{solved}\,A \;:=\; \forall s \in A.\ \Sigma x.\ (s = x \wedge \neg x \notin A) + (s = (\neg x) \wedge x \notin A)$
  Every formula in A is either a variable or a negated variable and there is no clash $x \in A \wedge \neg x \in A$.
- $\mathsf{clashed}\,A \;:=\; \bot \in A \;+\; \Sigma s.\ s \in A \wedge (\neg s) \in A$
  A contains either $\bot$ or a clash $s \in A \wedge \neg s \in A$.

Fact 23.11.1 (Solved and clashed clauses)
Solved clauses are satisfiable, and clashed clauses are unsatisfiable and refutable:

1. $\mathsf{solved}\,A \to \mathsf{sat}\,A$
2. $\mathsf{clashed}\,A \to \mathsf{sat}\,A \to \bot$
3. $\mathsf{clashed}\,A \to (A \dot{\vdash} \bot)$

Proof Straightforward. Exercise. ∎

Except for the terminal rule and the second so-called rotation rule, the derivation rules of both systems correspond to the decomposition schemes in Figure 23.5. The relationship with the decomposition rules becomes clear if one reads the derivation rules backwards from the conclusion to the premises. If a scheme decomposes with an "or", this translates for σ to two rules and for ρ to one rule with two premises. The rotation rule (second rule in both systems) makes it possible to move a decomposable formula into head position, as required by the decomposition rules.

The informal design rational for the rules of ρ is as follows: An assignment satisfies the conclusion of the rule if and only if it satisfies one premise of the rule.

Fact 23.11.2 (Boolean soundness)
$\sigma A \to \mathsf{sat}\,A$.

Proof By induction on the derivation of σA exploiting that solved clauses are satisfiable, and that for every recursive rule assignments satisfying the premise satisfy the conclusion. ∎

$$\frac{\textsf{solved}\,A}{\sigma(A)} \qquad \frac{\sigma(A +\!\!\!+ [s])}{\sigma(s :: A)} \qquad \frac{\sigma(A)}{\sigma(\neg\bot :: A)}$$

$$\frac{\sigma(s :: t :: A)}{\sigma(s \wedge t :: A)} \qquad \frac{\sigma(\neg s :: A)}{\sigma(\neg(s \wedge t) :: A)} \qquad \frac{\sigma(\neg t :: A)}{\sigma(\neg(s \wedge t) :: A)}$$

$$\frac{\sigma(s :: A)}{\sigma(s \vee t :: A)} \qquad \frac{\sigma(t :: A)}{\sigma(s \vee t :: A)} \qquad \frac{\sigma(\neg s :: \neg t :: A)}{\sigma(\neg(s \vee t) :: A))}$$

$$\frac{\sigma(\neg s :: A)}{\sigma(s \to t :: A)} \qquad \frac{\sigma(t :: A)}{\sigma(s \to t :: A)} \qquad \frac{\sigma(s :: \neg t :: A)}{\sigma(\neg(s \to t) :: A))}$$

Figure 23.6: Corefutation system $\sigma(A)$

$$\frac{\textsf{clashed}\,A}{\rho(A)} \qquad \frac{\rho(A +\!\!\!+ [s])}{\rho(s :: A)} \qquad \frac{\rho(A)}{\rho(\neg\bot :: A)}$$

$$\frac{\rho(s :: t :: A)}{\rho(s \wedge t :: A)} \qquad \frac{\rho(\neg s :: A) \qquad \rho(\neg t :: A)}{\rho(\neg(s \wedge t) :: A)}$$

$$\frac{\rho(s :: A) \qquad \rho(t :: A)}{\rho(s \vee t :: A)} \qquad \frac{\rho(\neg s :: \neg t :: A)}{\rho(\neg(s \vee t) :: A)}$$

$$\frac{\rho(\neg s :: A) \qquad \rho(t :: A)}{\rho(s \to t :: A)} \qquad \frac{\rho(s :: \neg t :: A)}{\rho(\neg(s \to t) :: A)}$$

Figure 23.7: Refutation system $\rho(A)$

Fact 23.11.3 (Boolean soundness)
$\rho A \to \mathsf{sat}\, A \to \bot$.

Proof By induction on the derivation of ρA. As a representative example, we consider the proof obligation for the positive implication rule:

$$(\mathsf{sat}(\neg s :: A) \to \bot) \to (\mathsf{sat}(t :: A) \to \bot) \to \mathsf{sat}((s \to t) :: A) \to \bot$$

By assumption we have an assignment α satisfying A and $s \to t$. Thus α satisfies either $\neg s$ or t. Hence α satisfies either $\neg s :: A$ or $t :: A$. Both cases are contradictory with the assumptions. ∎

We now observe that ρ is also sound for ND refutation.

Fact 23.11.4 (ND soundness)
$\rho A \to (A \dot{\vdash} \bot)$.

Proof By induction on the derivation of ρA. As a representative example, we consider the proof obligation for the positive implication rule:

$$(\neg s :: A \dot{\vdash} \bot) \to (t :: A \dot{\vdash} \bot) \to (s \to t :: A \dot{\vdash} \bot)$$

By the implication lemma (Fact 23.5.2). it suffices to show $\dot{\vdash} \neg\neg s \to \neg t \to \neg(s \to t)$, which is routine. ∎

Exercise 23.11.5 Define the derivation systems σ and ρ as inductive type families and say whether the arguments are parameters or an indices.

Exercise 23.11.6 Verify the proof of ND soundness (Lemma 23.11.4) in detail. The proof is modular in that there is a separate proof obligation for every rule of the refutation systems (Figure 23.7). The obligation for the rotation rule

$$(A +\!\!\!+ [s] \vdash \bot) \to (s :: A \vdash \bot)$$

follows with weakening, and the obligations for the terminal rules are obvious. The obligations for the decomposition rules follow with the implication lemma (Fact 23.5.2) and the derivability of the ND judgments from Exercise 23.5.8.

23.12 Certifying Boolean Solvers

We now construct a certifying solver $\forall A.\ \sigma(A) + \rho(A)$. Given the soundness theorems for σ and ρ, this solver yields certifying solvers $\forall A.\ \mathcal{D}(\mathsf{sat}\, A)$ and $\forall A.\ \mathsf{sat}\, A + (A \dot{\vdash} \bot)$. The main issue in constructing the basic solver $\forall A.\ \sigma(A) + \rho(A)$ is finding a terminating strategy for formula decomposition.

We start with a **presolver** $\forall A.\ \mathsf{decomposable}(A) + \mathsf{solved}(A) + \mathsf{clashed}(A)$ that given a clause A either exhibits a decomposable formula in A or established A as solved or clashed. A formula is **decomposable** if it has the form $s \wedge t$, $\neg(s \wedge t)$, $s \vee t$, $\neg(s \vee t)$, $s \to t$ with $t \neq \bot$, or $\neg(s \to t)$. A clause is **decomposable** if it contains a decomposable formula:

$$\mathsf{decomposable}(A) \ := \ \Sigma BsC.\ \ A = B +\!\!\!+ s :: C \ \wedge \ \mathsf{decomposable}(s)$$

Lemma 23.12.1 (Presolver)
$\forall A.\ \ \mathsf{decomposable}(A) + \mathsf{solved}(A) + \mathsf{clashed}(A)$.

Proof By induction on A. Straightforward. ∎

To establish the termination of our decomposition strategy, we employ a *size function*

$$\gamma : \mathcal{L}(\mathsf{For}) \to \mathsf{N}$$

counting the constructors in the formulas in the list but omitting top-level negations. For instance,

$$\gamma\,[(x \to \neg x),\ \neg(\neg x \wedge x),\ \neg x] = 11$$

Note that $\neg x$ counts 1 if appearing at the top level, but 3 if not appearing at the top level (since $\neg x$ abbreviates $x \to \bot$). We observe that every scheme in the decomposition table (Figure 23.5) reduces the size of a clause as obtained with γ.

Next we obtain a certifying function rotating a given formula in a clause to the front of the clause such that derivability with σ and τ is propagated and the size of the clause is preserved.

Lemma 23.12.2 (Rotator)

$$\begin{aligned}\forall AsB.\ \Sigma C.\ \ &(\sigma(s :: C) \to \sigma(A +\!\!\!+ s :: B)) \ \times \\ &(\rho(s :: C) \to \rho(A +\!\!\!+ s :: B)) \ \times \\ &(\gamma(s :: C) = \gamma(A +\!\!\!+ s :: B)).\end{aligned}$$

Proof By induction on A using the rotation rules of σ and ρ. ∎

Lemma 23.12.3 (Basic certifying solver)
$\forall A.\ \sigma(A) + \rho(A)$.

Proof By size recursion on A. We first apply the presolver to A. If the presolver yields the claim using the terminal rules, we are done. Otherwise, we use the rotator to move the decomposable formula found by the presolver into head position. We now recurse following the unique decomposition scheme applying. ∎

We now come to the theorem we were aiming at in this and the previous section.

Theorem 23.12.4 (ND solver) $\forall A.\ \mathsf{sat}\, A + (A \dot{\vdash} \bot)$.

Proof Lemma 23.12.3 and soundness theorems. ■

Exercise 23.12.5 (Decidability of satisfiability)
Prove the following facts.

a) $\forall A.\ \sigma(A) \Leftrightarrow \mathsf{sat}(A)$

b) $\forall A.\ \rho(A) \Leftrightarrow (\mathsf{sat}(A) \to \bot)$

c) $\forall A.\ \mathcal{D}(\sigma(A))$

d) $\forall A.\ \mathcal{D}(\rho(A))$

e) $\forall A.\ \mathcal{D}(\mathsf{sat}(A))$.

Exercise 23.12.6 From our development it is clear that a solver $\forall A.\ \mathsf{sat}\, A + (A \dot{\vdash} \bot)$ can be constructed without making the derivation systems σ and ρ and the accompanying soundness lemmas explicit. Try to rewrite the existing Coq development accordingly. This will lead to a shorter (as it comes to lines of code) but less transparent proof.

23.13 Boolean Entailment

We define **boolean entailment** as follows:

$$A \dot{\vDash} s \;:=\; \forall \alpha.\ \mathsf{sat}\, \alpha A \to \mathsf{sat}\, \alpha s$$

Boolean entailment describes the boolean consequence relation commonly used in mathematics. We will show that classical ND agrees with boolean entailment.

Fact 23.13.1 (ND soundness) $(A \dot{\vdash} s) \to (A \dot{\vDash} s)$.

Proof By induction on the derivation $A \dot{\vdash} s$. ■

Fact 23.13.2 (ND completeness) $(A \dot{\vDash} s) \to (A \dot{\vdash} s)$.

Proof We assume $A \dot{\vDash} s$. Using the ND solver (Theorem 23.12.4), we have $\mathsf{sat}(\neg s :: A) + (\neg s :: A \dot{\vdash} \bot)$. If $\neg s :: A \dot{\vdash} \bot$, the claim follows with the contradiction rule. If $\mathsf{sat}(\neg s :: A)$, we have a contradiction with $A \dot{\vDash} s$. ■

Corollary 23.13.3 Boolean entailment $A \dot{\vDash} s$ and classical ND $A \dot{\vdash} s$ agree.

Note that the proofs of the two directions of the agreement $(A \dot{\vdash} s) \Leftrightarrow (A \dot{\vDash} s)$ are independent, and that only the completeness direction requires the ND solver.

Next we observe that boolean entailment reduces to unsatisfiability.

Fact 23.13.4 (Reduction to unsatisfiability)
$(A \dot{\vDash} s) \Leftrightarrow (\mathsf{sat}(\neg s :: A) \to \bot)$.

Proof Direction $\to$ is easy. For the other direction we assume $\mathsf{sat}(\neg s :: A) \to \bot$ and $\mathsf{sat}\,\alpha A$ and show $\mathcal{E}\alpha s = \mathbf{T}$. We now assume $\mathcal{E}\alpha s = \mathbf{F}$ and obtain a contradiction from our assumptions since $\mathcal{E}\alpha(\neg s) = \mathbf{T}$. ∎

Fact 23.13.5 (ND decidability) $\forall A s.\ \mathcal{D}(A \dot{\vdash} s)$.

Proof With the ND solver (Theorem 23.12.4) we obtain $\mathsf{sat}(\neg s :: A) + (\neg s :: A \dot{\vdash} \bot)$. If $\neg s :: A \dot{\vdash} \bot$, we have $A \dot{\vdash} s$ by the contradiction rule. Otherwise, we assume $\mathsf{sat}(\neg s :: A)$ and $A \dot{\vdash} s$ and obtain a contradiction with soundness (Fact 23.13.1) and Fact 23.13.4. ∎

Exercise 23.13.6 Note that the results in this section did not use results from the previous two sections except for the ND solver (Theorem 23.12.4). Prove the following facts using the results from this section and possibly the ND solver.

a) $\forall A s.\ \mathcal{D}(A \dot{\vDash} s)$
b) $\forall A.\ \mathcal{D}(\mathsf{sat}\,A)$
c) $\forall A.\ \mathsf{sat}\,A \Leftrightarrow ((A \dot{\vdash} \bot) \to \bot)$

Exercise 23.13.7 Give a consistency proof for classical ND that does not make use of intuitionistic ND.

Exercise 23.13.8 Show that x and $\neg x$ are independent in $\dot{\vdash}$.

Exercise 23.13.9 Show that $\neg\neg\neg x$ is independent in $\dot{\vdash}$.

Exercise 23.13.10 Show $(\forall s t.\ \dot{\vdash}(s \vee t) \to (\dot{\vdash} s) \vee (\dot{\vdash} t)) \to \bot$.

23.14 Cumulative Refutation System

Refutation systems based on formula decomposition exist in many variations in the literature, where they often appear under the names *tableaux systems* and *Gentzen systems*. They also exist for intuitionistic provability and modal logic. See Troelstra's and Schwichtenberg's textbook [25] to know more.

Figure 23.8 shows a refutation system γ modifying our refutation system ρ so that the formula to be decomposed can be at any position of the list and is not deleted when it is decomposed. Hence no rotation rule is needed.

We speak of the *cumulative refutation system*. When realized with an inductive type family, the argument A of the type constructor γ comes out as a nonuniform

$$\frac{\bot \in A}{\gamma(A)} \qquad \frac{s \in A \qquad \neg s \in A}{\gamma(A)}$$

$$\frac{(s \wedge t) \in A \qquad \gamma(s :: t :: A)}{\gamma(A)} \qquad \frac{\neg(s \wedge t) \in A \qquad \gamma(\neg s :: A) \qquad \gamma(\neg t :: A)}{\gamma(A)}$$

$$\frac{(s \vee t) \in A \qquad \gamma(s :: A) \qquad \gamma(t :: A)}{\gamma(A)} \qquad \frac{\neg(s \vee t) \in A \qquad \gamma(\neg s :: \neg t :: A)}{\gamma(A)}$$

$$\frac{(s \to t) \in A \qquad \gamma(\neg s :: A) \qquad \gamma(t :: A)}{\gamma(A)} \qquad \frac{\neg(s \to t) \in A \qquad \gamma(s :: \neg t :: A)}{\gamma(A)}$$

Figure 23.8: Cumulative refutation system

parameter. So, in contrast to the derivation systems we considered before, the inductive type family $\gamma(A)$ has no index argument and thus belongs to the BNF class of inductive types.

Fact 23.14.1 (Boolean soundness)
$\gamma(A) \to \exists s \in A.\ \mathcal{E}\alpha s = \mathbf{F}$.

Proof By induction on the derivation $\gamma(A)$. Similar to the proof of Fact 23.11.3. ∎

Fact 23.14.2 (Weakening)
$\gamma(A) \to A \subseteq B \to \gamma(B)$.

Proof By induction on $\gamma(A)$ with B quantified. ∎

Fact 23.14.3 (Completeness)
$\rho(A) \to \gamma(A)$.

Proof Straightforward using weakening. ∎

Fact 23.14.4 (Agreement)
$\rho(A) \Leftrightarrow \gamma(A)$.

Proof Completeness (Fact 23.14.3), certifying boolean solver for ρ (Theorem 23.12.3), and boolean soundness (Fact 23.14.1). ∎

The rules of the cumulative refutation system yield a method for refuting formulas working well with pen and paper. We demonstrate the method at the example of the unsatisfiable formula $\neg(((s \to t) \to s) \to s)$.

	$\neg(((s \to t) \to s) \to s)$	negated implication
	$(s \to t) \to s$	positive implication
	$\neg s$	
1	$\neg(s \to t)$	negative implication
	s	clash with $\neg s$
	$\neg t$	
2	s	clash with $\neg s$

Exercise 23.14.5 Refute the negations of the following formulas with the cumulative refutation system drawing a diagram as in the example above.

a) $s \vee \neg s$

b) $s \to \neg\neg s$

c) $\vdash \neg\neg s \to \neg t \to \neg(s \to t)$

d) $\vdash (\neg t \to \neg s) \to \neg\neg(s \to t)$

e) $\vdash (t \to \neg s) \to \neg(s \wedge t)$

f) $\vdash \neg\neg s \to \neg\neg t \to \neg\neg(s \wedge t)$

g) $\vdash \neg s \to \neg t \to \neg(s \vee t)$

h) $\vdash (\neg t \to \neg\neg s) \to \neg\neg(s \vee t)$

Exercise 23.14.6 (Saturated lists) A list A is *saturated* if the decomposition rules of the cumulative refutation system do not add new formulas:

1. If $(s \wedge t) \in A$, then $s \in A$ and $t \in A$.
2. If $\neg(s \wedge t) \in A$, then $\neg s \in A$ or $\neg t \in A$.
3. If $(s \vee t) \in A$, then $s \in A$ or $t \in A$.
4. If $\neg(s \vee t) \in A$, then $\neg s \in A$ and $\neg t \in A$.
5. If $(s \to t) \in A$, then $\neg s \in A$ or $t \in A$.
6. If $\neg(s \to t) \in A$, then $s \in A$ and $\neg t \in A$.

Prove that an assignment α satisfies a saturated list A not containing $\bot$ if it satisfies all *atomic formulas* (x and $\neg x$) in A.
Hint: Prove

$$\forall s.\ (s \in A \to \mathcal{E}\alpha s = \mathbf{T}) \wedge (\neg s \in A \to \mathcal{E}\alpha(\neg s) = \mathbf{T})$$

by induction on s.

23.15 Substitution

In the deduction systems we consider in this chapter, atomic formulas act as variables for formulas. We will now show that derivability of formulas is preserved if one instantiates atomic formulas. To ease our language, we call atomic formulas **propositional variables** in this section.

A **substitution** is a function $\theta : \mathsf{N} \to \mathsf{For}$ mapping every number to a formula. Recall that propositional variables are represented as numbers. 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 23.15.1 $s \in A \to \theta s \in \theta A$.

Proof By induction on A. ■

Fact 23.15.2 (Substitutivity) $A \vdash s \to \theta A \vdash \theta s$ 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 23.15.1 for the assumption rule. ■

Exercise 23.15.3 Prove that substitution preserves derivability in the intuitionistic Hilbert system $\mathcal{H}$. Note that the proof obligation for the axiomatic rules all follow with the same technique. Now use the equivalence with the ND system and Glivenko to show substitutivity for the other three systems.

23.16 Entailment Relations

An **entailment relation** is a predicate[4]

$$\Vdash : \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{P}$$

satisfying the properties listed in Figure 23.9. Note that the first five 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.

[4]We are reusing the turnstile $\Vdash$ previously used for Hilbert systems.

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 23.9: Requirements for entailment relations

Fact 23.16.1 Intuitionistic provability ($A \vdash s$) and classical provability ($A \dot{\vdash} s$) are entailment relations.

Proof Follows with the results shown so far. ■

Fact 23.16.2 Boolean entailment $A \dot{\vDash} s$ is an entailment relation.

Proof Follows with Fact 23.16.1 since boolean entailment agrees with classical ND (Fact 23.13.3). ■

It turns out that every entailment relation is sandwiched between intuitionistic provability at the bottom and classic provability at the top. Let $\Vdash$ be an entailment relation in the following.

Fact 23.16.3 (Modus Ponens) $A \Vdash (s \to t) \;\to\; A \Vdash s \;\to\; A \Vdash t$.

Proof By implication and cut. ■

Fact 23.16.4 (Least entailment relation)
Intuitionistic provability is a least entailment relation: $A \vdash s \;\to\; A \Vdash s$.

Proof By induction on $A \vdash s$ using modus ponens. ■

Fact 23.16.5 $\Vdash s \;\to\; \Vdash \neg s \;\to\; \bot$.

Proof Let $\Vdash s$ and $\Vdash \neg s$. By Fact 23.16.3 we have $\Vdash \bot$. By consistency and explosion we obtain a contradiction. ■

Fact 23.16.6 (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 23.16.7 (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 23.16.3 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 23.16.8 $\Vdash s \to \dot{\vDash} s$.

Proof Let $\Vdash s$. 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 23.16.5. ∎

Fact 23.16.9 (Greatest entailment relation)
Boolean entailment is a greatest entailment relation: $A \Vdash s \to A \dot{\vDash} s$.

Proof Follows with reversion (Facts 23.16.6 and 23.16.2) and Lemma 23.16.8. ∎

Theorem 23.16.10 (Sandwich) Every entailment relation $\Vdash$ satisfies $\vdash \subseteq \Vdash \subseteq \dot{\vdash}$.

Proof Facts 23.16.4, 23.16.9, and 23.13.3. ∎

Exercise 23.16.11 Let $\Vdash$ be an entailment relation. 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 23.16.12 Tebbi's lemma provides for a particularly elegant proof of Lemma 23.16.8. Verify that Lemma 23.16.8 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 23.16.4 and 23.16.5.

23.17 Notes

The study of natural deduction originated in the 1930's with the work of Gerhard Gentzen [12, 13] and Stanisław Jaśkowski [17]. The standard text on natural deduction and proof theory is Troelstra and Schwichtenberg [25].

Decidability of intuitionistic ND One can show that intuitionistic ND is decidable. This can be done with a formula decomposition 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 relations 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 In the classical systems, falsity and implication can express conjunction and disjunction. On the other hand, one can prove using Heyting structures that in intuitionistic systems the logical constants are independent.

Certifying Functions The construction of the certifying solvers and their auxiliary functions in this chapter are convincing examples for the efficiency and power of certifying functions. Imagine you would have to carry out these constructions in a functional programming language with simply typed functions defined with equations based on informal specifications.

24 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).

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

$$\mathbf{T} \,\&\, b := b \qquad \mathbf{T} \mid b := \mathbf{T} \qquad !\,\mathbf{T} := \mathbf{F}$$
$$\mathbf{F} \,\&\, b := \mathbf{F} \qquad \mathbf{F} \mid b := b \qquad !\,\mathbf{F} := \mathbf{T}$$

With these definitions, boolean identities like

$$a \,\&\, b = b \,\&\, a \qquad a \mid b = b \mid a \qquad !\,!\,b = b$$

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*

$$!(a \,\&\, b) = \,!a \mid !b \qquad !(a \mid b) = \,!a \,\&\, !b \qquad !\,!\,a = a$$

and conjunctions can be pushed below disjunctions with the *distribution law*

$$a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, b)$$

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 $\mathbf{T}$.

24.2 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 24.1. 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

$$\alpha \,\mathsf{sats}\, s := \mathcal{E}\alpha s = \mathbf{T}$$

$$\begin{aligned} \mathcal{E}\alpha x &:= \alpha x \\ \mathcal{E}\alpha\bot &:= \mathbf{F} \\ \mathcal{E}\alpha(s \to t) &:= \,!\mathcal{E}\alpha s \mid \mathcal{E}\alpha t \\ \mathcal{E}\alpha(s \wedge t) &:= \mathcal{E}\alpha s \,\&\, \mathcal{E}\alpha t \\ \mathcal{E}\alpha(s \vee t) &:= \mathcal{E}\alpha s \mid \mathcal{E}\alpha t \end{aligned}$$

Figure 24.1: 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 $\alpha \mathsf{sat} s$ abbreviates the boolean equation $\mathcal{E}\alpha s = \mathbf{T}$. Reasoning with boolean equations will be the main workhorse in our proofs.

Exercise 24.2.1 Prove that $s \to t$ and $\neg s \vee t$ are equivalent.

Exercise 24.2.2 Convince yourself that the predicate $\alpha \mathsf{sat} s$ is decidable.

Exercise 24.2.3 Verify the following reflection laws for formulas:

$$\begin{aligned} \alpha\mathsf{sat}(s \wedge t) &\longleftrightarrow \alpha\mathsf{sat}s \wedge \alpha\mathsf{sat}t \\ \alpha\mathsf{sat}(s \vee t) &\longleftrightarrow \alpha\mathsf{sat}s \vee \alpha\mathsf{sat}t \\ \alpha\mathsf{sat}\neg s &\longleftrightarrow \neg(\alpha\mathsf{sat}s) \end{aligned}$$

Exercise 24.2.4 (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 24.2.5 (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:

$$\forall \alpha.\ \ \alpha\mathsf{sat}\gamma A \ \longleftrightarrow\ \forall (s,t) \in A.\ \ \mathcal{E}\alpha s = \mathcal{E}\alpha t$$

Exercise 24.2.6 (Valid formulas) We say that a formula is **valid** if it is satisfied by all assignments: $\mathsf{val}\, s := \forall \alpha.\ \alpha\mathsf{sat}s$. 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 24.2.7 Write an evaluator $f : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathbb{P}$ such that $f\alpha s \longleftrightarrow \alpha \mathsf{sat} s$ and $f\alpha(s \vee t) \approx f\alpha s \vee f\alpha t$ for all formulas s, t.
Hint: Recall the reflection laws from §24.1.

24.3 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 $s_1 \vee \cdots \vee s_n$ 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 $x \vee \neg x$ and $y \vee \neg y$ 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:

$$S, T : \mathsf{SFor} ::= s^+ \mid s^- \qquad \textbf{signed formula}$$
$$C, D : \mathsf{Cla} := \mathcal{L}(\mathsf{SFor}) \qquad \textbf{clause}$$

Clauses represent conjunctions. We define evaluation of signed formulas and clauses as follows:

$$\mathcal{E}\alpha(s^+) := \mathcal{E}\alpha s \qquad \mathcal{E}\alpha\,[] := \mathbf{T}$$
$$\mathcal{E}\alpha(s^-) := \,!\mathcal{E}\alpha s \qquad \mathcal{E}\alpha(S :: C) := \mathcal{E}\alpha S \,\&\, \mathcal{E}\alpha C$$

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 $\alpha \mathsf{sat} S$, $\alpha \mathsf{sat} C$, $\alpha \mathsf{sat}\Delta$, and sat C. Since formulas, signed formulas, clauses, and lists of clauses all come with the notion of satisfying assignments, we can speak about **equivalence** between these objects although they belong to different types. For instance, s, s^+, $[s^+]$, and $[[s^+]]$, are all equivalent since they are satisfied by the same assignments.

A **solved clause** is a clause consisting of signed variables (i.e., x^+ and x^-) such that no variable appears positively and negatively. Note that a solved clause C is satisfied by every assignment that maps the positive variables in C to $\mathbf{T}$ and the negative variables in C to $\mathbf{F}$.

Fact 24.3.1 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 24.3.2 A clausal DNF is satisfiable if and only if it is nonempty.

Exercise 24.3.3 Prove $\mathcal{E}\alpha(C + D) = \mathcal{E}\alpha C \;\&\; \mathcal{E}\alpha D$ and $\mathcal{E}\alpha(\Delta + \Delta') = \mathcal{E}\alpha\Delta \mid \mathcal{E}\alpha\Delta'$.

Exercise 24.3.4 Write a function that maps lists of clauses to equivalent formulas.

Exercise 24.3.5 Our formal proof of Fact 24.3.1 is unexpectedly tedious in that it requires two inductive lemmas:

1. $\alpha \,\mathsf{sat}\, C \;\longleftrightarrow\; \forall S \in C.\ \alpha \,\mathsf{sat}\, S$.
2. $\mathsf{solved}\ C \;\to\; S \in C \;\to\; \exists x.\ (S = x^+ \land x^- \notin C) \lor (S = x^- \land x^+ \notin C)$.

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 24.3.1. Can you do better?

24.4 DNF Solver

We would like to construct a function computing clausal DNFs for formulas. Formally, we specify the function with the informative type

$$\forall s\, \Sigma \Delta.\ \ \mathsf{DNF}\,\Delta \land s \equiv \Delta$$

where

$$\begin{aligned} s \equiv \Delta \;&:=\; \forall \alpha.\ \alpha \,\mathsf{sat}\, s \longleftrightarrow \alpha \,\mathsf{sat}\, \Delta \\ \mathsf{DNF}\,\Delta \;&:=\; \forall C \in \Delta.\ \mathsf{solved}\, C \end{aligned}$$

To define the function, we will generalize the type to

$$\forall CD.\ \mathsf{solved}\, C \to \Sigma \Delta.\ \ \mathsf{DNF}\,\Delta \land C + D \equiv \Delta$$

$$
\begin{aligned}
\mathsf{dnf}\ C\ []\ &=\ [C]\\
\mathsf{dnf}\ C\ (x^+ :: D)\ &=\ \text{IF } \ulcorner x^- \in C \urcorner \text{ THEN } []\ \text{ELSE } \mathsf{dnf}\ (x^+ :: C)\ D\\
\mathsf{dnf}\ C\ (x^- :: D)\ &=\ \text{IF } \ulcorner x^+ \in C \urcorner \text{ THEN } []\ \text{ELSE } \mathsf{dnf}\ (x^- :: C)\ D\\
\mathsf{dnf}\ C\ (\bot^+ :: D)\ &=\ []\\
\mathsf{dnf}\ C\ (\bot^- :: D)\ &=\ \mathsf{dnf}\ C\ D\\
\mathsf{dnf}\ C\ ((s \to t)^+ :: D)\ &=\ \mathsf{dnf}\ C\ (s^- :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^+ :: D)\\
\mathsf{dnf}\ C\ ((s \to t)^- :: D)\ &=\ \mathsf{dnf}\ C\ (s^+ :: t^- :: D)\\
\mathsf{dnf}\ C\ ((s \wedge t)^+ :: D)\ &=\ \mathsf{dnf}\ C\ (s^+ :: t^+ :: D)\\
\mathsf{dnf}\ C\ ((s \wedge t)^- :: D)\ &=\ \mathsf{dnf}\ C\ (s^- :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^- :: D)\\
\mathsf{dnf}\ C\ ((s \vee t)^+ :: D)\ &=\ \mathsf{dnf}\ C\ (s^+ :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^+ :: D)\\
\mathsf{dnf}\ C\ ((s \vee t)^- :: D)\ &=\ \mathsf{dnf}\ C\ (s^- :: t^- :: D)
\end{aligned}
$$

Figure 24.2: Specification of a procedure $\mathsf{dnf} : \mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$

where $C \equiv \Delta := \forall \alpha.\ \alpha \mathsf{sat} C \longleftrightarrow \alpha \mathsf{sat} \Delta$. 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 24.2. 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:

$$
\begin{aligned}
\sigma[]\ &:=\ 0 & \sigma x\ &:=\ 1\\
\sigma(s^+ :: C)\ &:=\ \sigma s + \sigma C & \sigma\bot\ &:=\ 1\\
\sigma(s^- :: C)\ &:=\ \sigma s + \sigma C & \sigma(s \circ t)\ &:=\ 1 + \sigma s + \sigma t
\end{aligned}
$$

Note that the equations specifying the procedure in Figure 24.2 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 §24.1.

Lemma 24.4.1 $\forall CD.\ \mathsf{solved}\ C \to \Sigma\Delta.\ \ \mathsf{DNF}\,\Delta \wedge C +\!\!\!+ D \equiv \Delta.$

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 24.4.2 (DNF solver) $\forall C \Sigma\Delta.\ \mathsf{DNF}\,\Delta \wedge C \equiv \Delta.$

Proof Immediate from Lemma 24.4.1. ∎

Corollary 24.4.3 $\forall s \Sigma\Delta.\ \mathsf{DNF}\,\Delta \wedge s \equiv \Delta.$

Corollary 24.4.4 There is a solver $\forall C.\ (\Sigma\alpha.\ \alpha \mathsf{sat} C) + \neg \mathsf{sat}\ C.$

Corollary 24.4.5 There is a solver $\forall s.\ (\Sigma\alpha.\ \alpha \mathsf{sat} s) + \neg \mathsf{sat}\ s.$

Corollary 24.4.6 Satisfiability of clauses and formulas is decidable.

Exercise 24.4.7 Convince yourself that the predicate $S \in C$ is decidable.

Exercise 24.4.8 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 24.4.9 Recall the definition of valid formulas from Exercise 24.2.6. 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 24.4.10 If you are already familiar with well-founded recursion in computational type theory (Chapter 31), define a function $\mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$ satisfying the equations specifying the procedure dnf in Figure 24.2.

24.5 DNF Recursion

From the equations for the DNF procedure (Figure 24.2) and the construction of the basic DNF solver (Lemma 24.4.1) one can abstract out the recursion scheme shown in Figure 24.3. We refer to this recursion scheme as **DNF recursion**. DNF recursion has one clause for every equation of the DNF procedure in Figure 24.2 where the recursive calls appear as inductive hypotheses. DNF recursion simplifies the proof of Lemma 24.4.1. However, DNF recursion can also be used for other constructions (our main example is a completeness lemma (24.6.5) 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 24.3: DNF recursion scheme

Lemma 24.5.1 (DNF recursion)
The DNF recursion scheme shown in Figure 24.3 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 24.5.2 Use DNF recursion to construct a certifying boolean solver for clauses: $\forall C.\ (\Sigma\alpha.\ \alpha\mathsf{sat}C) + (\neg\mathsf{sat}(C))$.

$$\frac{\mathsf{tab}(S :: C +\!\!\!+ D)}{\mathsf{tab}(C +\!\!\!+ S :: D)} \qquad \frac{}{\mathsf{tab}(x^+ :: x^- :: C)} \qquad \frac{}{\mathsf{tab}(\bot^+ :: C)}$$

$$\frac{\mathsf{tab}(s^- :: C) \qquad \mathsf{tab}(t^+ :: C)}{\mathsf{tab}((s \to t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^+ :: t^- :: C)}{\mathsf{tab}((s \to t)^- :: C)}$$

$$\frac{\mathsf{tab}(s^+ :: t^+ :: C)}{\mathsf{tab}((s \wedge t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^- :: C) \qquad \mathsf{tab}(t^- :: C)}{\mathsf{tab}((s \wedge t)^- :: C)}$$

$$\frac{\mathsf{tab}(s^+ :: C) \qquad \mathsf{tab}(t^+ :: C)}{\mathsf{tab}((s \vee t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^- :: t^- :: C)}{\mathsf{tab}((s \vee t)^- :: C)}$$

Figure 24.4: Inductive type family $\mathsf{tab} : \mathsf{Cla} \to \mathbb{T}$

24.6 Tableau Refutations

Figure 24.4 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 24.4 as a simplification of the equations specifying the DNF procedure in Figure 24.2. 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 24.6.1 (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 24.6.2 (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 24.6.3 (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 24.6.4 (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 24.6.5 (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 24.6.6
A clause is tableau refutable if and only if it is unsatisfiable:
$\mathsf{tab}(C) \Leftrightarrow \neg\mathsf{sat}(C)$.

Proof Follows with Fact 24.6.1 and Lemma 24.6.5. ∎

Corollary 24.6.7 $\forall C.\ \mathsf{tab}(C) + (\mathsf{tab}(C) \to \bot)$.

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 24.6.8 Prove $\mathsf{tab}(C + S :: D + T :: E) \longleftrightarrow \mathsf{tab}(C + T :: D + S :: E)$.

Exercise 24.6.9 Give an inductive type family deriving exactly the satisfiable clauses. Start with an inductive family deriving exactly the solved clauses.

24.7 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 23).

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 $\alpha\,\mathsf{sat}A$ 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 24.5: Rules for abstract refutation systems $\rho : \mathcal{L}(\mathsf{For}) \to \mathbb{P}$

Fact 24.7.1 $\mathcal{E}\alpha\hat{C} = \mathcal{E}\alpha C$, $\mathcal{E}\alpha A^+ = \mathcal{E}\alpha A$, and $\widehat{A^+} = A$.

Fact 24.7.2 (Decidability) Satisfiability of unsigned clauses is decidable.

Proof Follows with Corollary 24.4.6 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 24.5. 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 24.7.3 Let ρ be a refutation system. Then $\mathsf{tab}\ C \to \rho\hat{C}$.

Proof Straightforward by induction on $\mathsf{tab}\ C$. ■

Fact 24.7.4 (Completeness)
Every refutation system derives all unsatisfiable unsigned clauses.

Proof Follows with Theorem 24.6.6 and Lemma 24.7.3. ■

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 24.7.5 A sound refutation system is decidable and derives exactly the unsatisfiable unsigned clauses.

Proof Facts 24.7.4 and 24.7.2. ■

Theorem 24.7.6 The minimal refutation system inductively defined with the rules for abstract refutation systems derives exactly the unsatisfiable unsigned clauses.

Proof Follows with Fact 24.7.4 and a soundness lemma similar to Fact 24.6.1. ■

Exercise 24.7.7 (Certifying Solver) Construct a function $\forall A.\ (\Sigma\alpha.\ \alpha \mathsf{sat} A) + \mathsf{tab}\, A$.

Exercise 24.7.8 Show that boolean entailment

$$A \dot{\models} s \ := \ \forall\alpha.\ \alpha \mathsf{sat} A \ \rightarrow \ \alpha \mathsf{sat} s$$

is decidable.

Exercise 24.7.9 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.

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 10.4.2).

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 18):

$$\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{sig}(\mathsf{dec}\,p) \to \mathsf{ex}\,p \to \mathsf{sig}\,p$$

We also need *arithmetic pairing functions* (Chapter 7)

$$\langle\_,\_\rangle : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$
$$\pi_1 : \mathsf{N} \to \mathsf{N}$$
$$\pi_2 : \mathsf{N} \to \mathsf{N}$$

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 characterization)
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 characterization)
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 $\to$. 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:

$$\mathcal{S} : \mathbb{P} \to \mathbb{T}$$
$$\mathcal{S}(P) := \Sigma f^{\mathsf{N}\to\mathsf{B}}.\ P \longleftrightarrow \mathsf{tsat}\, f$$

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) $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ (P \longleftrightarrow Q) \to S(P) \to S(Q)$.

Fact 25.3.3 $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ S(P) \to S(Q) \to S(P \land Q)$.

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 $P \land Q$. Note the use of the pairing functions π_1 and π_2. ∎

Fact 25.3.4 $\forall P^{\mathbb{P}} Q^{\mathbb{P}}.\ S(P) \to S(Q) \to S(P \lor Q)$.

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 $P \lor Q$. ∎

Fact 25.3.5 $\forall P Q^{\mathbb{P}}.\ \ S(P) \to S(Q) \to (P \lor Q) \to (P + Q)$.

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 $P \lor Q$. Since we have $P \lor Q$, 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 $S(\mathsf{tsat}\, f)$.

Proof Trivial. ∎

Fact 25.3.7 (MP characterization)
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 $S(\mathsf{tsat}\, f)$. Trivial. ∎

A **certifying semi-decider** for a predicate $p^{X \to \mathbb{P}}$ is a function $\forall x^X.\, S(px)$. From a certifying semi-decider for p we can obtain a semi-decider for p by forgetting the proofs. Vice versa, we can construct from a semi-decider and its correctness proof a certifying semi-decider.

Fact 25.3.8
We can translate between semi-deciders and certifying semi-deciders:
$\forall X^{\mathbb{T}} p^{X \to \mathbb{P}}.\ \mathsf{sig}(\mathsf{sdec}\, p) \Leftrightarrow \forall x. S(px)$.

Proof Direction $\Rightarrow$. We assume $\forall x.\ px \longleftrightarrow \exists n. fnx = \mathbf{T}$ and x^X and obtain $S(px)$ with fx as test.

Direction $\Leftarrow$. We assume $g^{\forall x. S(px)}$ 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 characterization 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 Characterization) 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 countable types 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 countable type is a type that has an enumerator and an equality decider.

Corollary 25.5.3 One can translate between enumerators and semi-deciders for predicates on countable 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 characterization)
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 characterization)
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 Characterizations

We have established many different equivalent characterizations of Markov's principle making connections between tests, deciders, semi-deciders, enumerators, and semi-decisions. Most of the characterizations use the notion of stability. The following fact collects prominent characterizations 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 characterizations of Markov's principle stated in Fact 25.7.1. Start by writing down formally the characterizations stated informally.

Notes

The chapter originated with Forster et al. [11]. Andrej Dudenhefner and Yannick Forster contributed nice facts about semi-deciders. Forster et al. [10] certify a reduction from the halting problem for Turing machines (HTM) to the Post correspondence problem (PCP) (§28.6) 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 countable 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 Abstract Reduction Systems

Warning: This chapter is under construction.

26.1 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 26.1.1 (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:

$$\begin{aligned} E :\ & \forall p^{\forall xy.\ \mathsf{path}\,xy \to \mathbb{T}}. \\ & (\forall x.\ pxx(\mathsf{P}_1\,x)) \to \\ & (\forall xyz\,\forall r^{Rxy}\,\forall a^{\mathsf{path}\,yz}.\ pxz(\mathsf{P}_2\,xyz\,ra)) \to \\ & \forall xya.\ pxya \end{aligned}$$

$$\begin{aligned} E\,pe_1e_2\,x\,\_\,,(\mathsf{P}_1\,\_) &:= e_1x \\ E\,pe_1e_2\,x\,\_\,(\mathsf{P}_2\,\_\,x'\,y\,r\,a) &:= e_2\,xx'\,y\,r(E\,pe_1e_2\,x'\,y\,a) \end{aligned}$$

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 26.1.2 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 26.1.3 Define the step function asserted by Fact 26.1.1 with a term.

Exercise 26.1.4 (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 26.1.5 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 26.1.6 (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 26.1.4). Direction $\Leftarrow$ follows with $\forall k.\ \mathsf{path}_R\, x(k + x)$, which follows by induction on k with x quantified.

26.2 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 26.1.1.

Fact 26.2.1 (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 26.1.4). 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 26.2.1 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 26.2.2 R^* and $(R^*)^*$ agree.

Proof We have $R^* \subseteq (R^*)^*$ by Fact 26.1.1. For the other direction $(R^*)^* \subseteq R^*$ we use star recursion (Fact 26.2.1). 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 26.2.3 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 26.2.2 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 26.2.4 (Functional characterization)
Prove $R^*xy \Leftrightarrow \forall p^{X \to X \to \mathbb{T}}.\ \mathsf{refl}\,p \to \mathsf{trans}\,p \to R \subseteq p \to pxy$.

Part IV

Indexed Inductive Types

27 Numeral Types as Indexed Inductive Types

This chapter is our first encounter with indexed inductive types. The value constructors of an indexed inductive type constructor can freely instantiate the so-called index arguments of the type constructor, which provides for the definition of fine-grained type families. As lead example we consider an indexed family of numeral types $\mathcal{N}(n)$. A numeral type $\mathcal{N}(n)$ has n elements obtained with a zero constructor $\forall n.\ \mathcal{N}(\mathsf{S}n)$ and a successor constructor $\forall n.\ \mathcal{N}(n) \to \mathcal{N}(\mathsf{S}n)$. Indexed numerals $\mathcal{N}(n)$ are in bijection with the recursive numerals $\mathcal{O}^n(\bot)$. The difference between the two families is that indexed numerals are obtained with a single inductive type definition while recursive numerals are obtained with recursion on numbers and the inductive type definitions for option types and falsity.

Indexed inductive types come with the technical challenge that the format for indexed discrimination is severely restricted. Thus, intuitively obvious discriminations must often be realized with elaborate encodings relying on nontrivial conversions.

Indexed inductive types have interesting applications and provide expressivity not available otherwise. This cannot be seen from the indexed numeral types we are considering here. Indexed numerals are still a good starting example for indexed inductive types since they provide a fine setting for explaining the new techniques.

27.1 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 don't write 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:

$$\begin{aligned}E\,pe_1e_2\,\_\,(\mathsf{Z}\,n) &:= e_1 n && : p(\mathsf{S}n)(\mathsf{Z}\,n)\\E\,pe_1e_2\,\_\,(\mathsf{U}\,na) &:= e_2na(E\,pe_1e_2\,na) && : p(\mathsf{S}n)(\mathsf{U}\,na)\end{aligned}$$

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).

27.2 Index Condition and Predecessors

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 appearing in index positions of discriminating types as **index variables**.

The index condition is a severe restriction often disallowing intuitively natural definitions. However, there are routine techniques to work around the index condition. We will demonstrate the issue with the definition of a predecessor function

$$P : \forall n.\ \mathcal{N}(\mathsf{S}n) \to \mathcal{O}(\mathcal{N}(n))$$

satisfying the computational equalities

$$\begin{aligned} Pn(\mathsf{U}na) &\approx {}^\circ a \\ Pn(\mathsf{Z}n) &\approx \emptyset \end{aligned}$$

Clearly, the index condition disallows the intuitively appealing definition taking the specifying equations as defining equations since this would discriminate on the type $\mathcal{N}(\mathsf{S}n)$ where the index argument $\mathsf{S}n$ is not a variable. To avoid the problem, we define a more general predecessor function discriminating on an unconstrained numeral type:

$$\begin{aligned} P' &: \forall n.\ \mathcal{N}(n) \to \text{MATCH } n\ [\, 0 \Rightarrow \bot \mid \mathsf{S}\,n' \Rightarrow \mathcal{O}(\mathcal{N}(n'))\,] \\ P'\,\_\,(\mathsf{Z}n) &:= \emptyset \qquad : \mathcal{O}(\mathcal{N}(n)) \\ P'\,\_\,(\mathsf{U}na) &:= {}^\circ a \qquad : \mathcal{O}(\mathcal{N}(n)) \end{aligned}$$

We now obtain a predecessor function as specified as follows:

$$\begin{aligned} P &: \forall n.\ \mathcal{N}(\mathsf{S}n) \to \mathcal{O}(\mathcal{N}(n)) \\ P\,na &:= P'(\mathsf{S}n)a \end{aligned}$$

Using P', we can also define a predecessor function

$$\hat{P} : \forall n.\ \mathcal{N}(\mathsf{SS}n) \to \mathcal{N}(\mathsf{S}n)$$

satisfying the computational equalities

$$\begin{aligned} \hat{P}\,n(\mathsf{U}(\mathsf{S}n)a) &\approx a \\ \hat{P}\,n\,(\mathsf{Z}(\mathsf{S}n)) &\approx \mathsf{Z}n \end{aligned}$$

Showing the constructor laws for numeral types is now routine using the predecessor function P. *Constructor disjointness*

$$\forall n\, \forall a^{\mathcal{N}(n)}.\ \mathsf{Z}\,n \neq \mathsf{U}\,na$$

follows with lemma `feq` (Figure 5.1) and P, which reduce to claim to $\emptyset \neq {}^\circ a$, one of the constructor laws for options. Injectivity of the value constructor U

$$\forall n\, \forall ab^{\mathcal{N}(n)}.\ \mathsf{U}\,na = \mathsf{U}\,nb \to a = b$$

follows again with `feq` and P, which reduce to claim to ${}^\circ a = {}^\circ b \to a = b$, the other constructor law for options.

Exercise 27.2.1 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 27.2.2 Prove $\mathcal{N}(0) \to \bot$. Hint: Use P'.

Exercise 27.2.3 (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: Define the listing function using the map function for lists. Use induction on n and the fact that nonrepeating lists are mapped to nonrepeating lists if the element function is injective (Exercise 19.5.7 (a)).

27.3 Inversion Operator

Intuition tells us that

$$\forall n\, \forall a^{\mathcal{N}(\mathsf{S}n)}.\ (a = \mathsf{Z}\, n) + (\Sigma a'.\ a = \mathsf{U}\, n a') \tag{27.1}$$

holds for numerals. To prove this fact, we define a more general function we call **inversion operator**. The type of the inversion operator

$$\begin{aligned} \mathsf{inv} :\ \forall n\, \forall a^{\mathcal{N}(n)}.\ & \text{MATCH } n \text{ RETURN } \mathcal{N}(n) \to \mathbb{T} \\ & [\, 0 \Rightarrow \lambda a.\ \bot \\ & \mid Sn \Rightarrow \lambda a.\ (a = \mathsf{Z}\, n) + (\Sigma a'.\ a = \mathsf{U}\, n a') \\ &]\, a \end{aligned}$$

is such that we can discriminate on the numeral argument. If we instantiate the type of the inversion operator with $n = 0$, we obtain $\mathcal{N}(0) \to \bot$ up to conversion, and if we instantiate the type with $n = \mathsf{S}n$, we obtain the type (27.1) 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

$$\begin{aligned} (\mathsf{Z}\, n = \mathsf{Z}\, n) + (\Sigma a'.\ \mathsf{Z}\, n = \mathsf{U}\, n a') \\ (\mathsf{U}\, n a = \mathsf{Z}\, n) + (\Sigma a'.\ \mathsf{U}\, n a = \mathsf{U}\, n a') \end{aligned}$$

The type of the inversion operator is written with what we call a **reloading match**.[1] The reloading of the numeral argument is necessary so that the branches of the match do type check.

[1] Chlipala [7] speaks of the *convoy pattern*.

The inversion operator described satisfies two computational equalities:

$$\begin{aligned} \mathsf{inv}\,(\mathsf{S}n)\,(\mathsf{Z}\,n) &\approx \mathsf{L}\,\mathsf{Q} \\ \mathsf{inv}\,(\mathsf{S}n)\,(\mathsf{U}\,a) &\approx \mathsf{R}\,(a,\mathsf{Q}) \end{aligned}$$

Using the inversion operator we can define an equality decider for numerals.

Fact 27.3.1 (Equality decider) $\forall n\,\forall ab^{\mathcal{N}(n)}.\ \mathcal{D}(a=b)$.

Proof By induction on a (using the eliminator) with b quantified, followed by inversion of b (using the inversion operator). The 4 cases follow with the constructor laws. ■

Exercise 27.3.2 Prove the following facts using the inversion operator:

a) $\mathcal{N}(0) \to \bot$

b) $\forall n\,\forall a^{\mathcal{N}(\mathsf{S}n)}.\ (a = \mathsf{Z}n) + \Sigma a'.\ a = \mathsf{U}na'$

c) $\forall a^{\mathcal{N}(1)}.\ a = \mathsf{Z}0$

d) $\forall a^{\mathcal{N}(2)}.\ (a = \mathsf{Z}1) + (a = \mathsf{U}(\mathsf{Z}0))$

Check your proof with the proof assistant. 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 27.3.3 Define the inversion operator in two ways: (1) with defining equations, and (2) using the universal eliminator. Convince yourself that the reloading match cannot be avoided. Write the inversion operator I such that it satisfies the following equations by computational equality (L and R are the constructors for sums; Q is the constructor for identity proofs (arguments are omitted):

$$\begin{aligned} I(\mathsf{S}n)(\mathsf{Z}n) &= \mathsf{L}\,\mathsf{Q} \\ I(\mathsf{S}n)(\mathsf{U}a) &= \mathsf{R}\,(a,\mathsf{Q}) \end{aligned}$$

Use the inversion operator to define a predecessor function P satisfying the equations specified in §27.2 by computational equality.

27.4 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

$$\forall n\, \forall a^{\mathcal{N}(n)}.\ \ Nna < n \tag{27.2}$$

which follows by induction on a (using the universal eliminator for numerals).

Next we show that N is injective:

$$\forall nab.\ Nna = Nnb \to a = b \tag{27.3}$$

This follows with a routine proof following the pattern used for the construction of the equality decider: Induction on a with b quantified followed by inversion of b using the inversion operator.

We now 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:

$$\forall n\, \forall a^{\mathcal{N}(\mathsf{S}n)}.\ \ B(N(\mathsf{S}n)a)n = a \tag{27.4}$$

$$\forall kn.\ \ k \le n \to N(\mathsf{S}n)(Bkn) = k \tag{27.5}$$

Note that (27.4) yields the injectivity of N. Moreover, (27.5) together with (27.2) yields the surjectivity of $N(\mathsf{S}n)$ for $\{0, \dots, n\}$.

The second roundtrip property (27.5) follows by a straightforward induction on k.

The first roundtrip property (27.4) 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.

Exercise 27.4.1 (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 $L\,5\,(\mathsf{U}(\mathsf{U}(\mathsf{Z}\,2))) \approx \mathsf{U}(\mathsf{U}(\mathsf{Z}\,3))$.

a) Prove $N(\mathsf{S}n)(Lna) = Nna$.

b) Prove that L is injective using the injectivity of N.

27.5 Recursive Numeral Types

We define **recursive numeral types** as follows:

$$\begin{aligned} &\mathcal{F} : \mathsf{N} \to \mathbb{T} \\ &\mathcal{F}(0) := \bot \\ &\mathcal{F}(\mathsf{S}n) := \mathcal{O}(\mathcal{F}(n)) \end{aligned}$$

We may think of recursive numerals $\mathcal{F}(n)$ as iterated option types $\mathcal{O}^n(\bot)$. In fact, we have $\mathcal{F}(n) \approx \mathcal{O}^n(\bot)$ if $\mathcal{O}^n(\bot)$ is obtained with an iteration operator as in §1.9.

Intuitively, it is clear that indexed numerals $\mathcal{N}(n)$ are in bijection with recursive numerals $\mathcal{F}(n)$. The constructors Z and U for indexed numerals correspond to the constructors $\emptyset$ and $^\circ$ for options. In fact, we have $\emptyset : \mathcal{F}(n)$ and $^\circ : \mathcal{F}(n) \to \mathcal{F}(\mathsf{S}n)$ due to the conversion rule. As one would expect, the elimination operator and the inversion operator for indexed numerals carry over to recursive numerals, with obvious routine constructions on the recursive side. We don't have a transport in the other direction since recursive numerals are a derived type family without a native elimination operation.

Exercise 27.5.1 Define and verify 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.

Exercise 27.5.2 Prove the following types:

a) $\mathcal{F}(0) \to \bot$

b) $\forall n\, \forall a^{\mathcal{F}(\mathsf{S}n)}.\ a \neq \mathsf{Z}n \to \Sigma a'.\ a = \mathsf{U}na'$

Note that the proofs are straightforward discrimination proofs.

Exercise 27.5.3 Define an operator for recursive numerals simulating the eliminator for indexed numerals:

$$\begin{aligned} &\forall p^{\forall n.\ \mathcal{F}(n) \to \mathbb{T}}. \\ &(\forall n.\ p(\mathsf{S}n)(\emptyset)) \to \\ &(\forall na.\ pna \to p(\mathsf{S}n)(^\circ a)) \to \\ &\forall na.\ pna \end{aligned}$$

Hint: Recurse on n and then discriminate on a.

Exercise 27.5.4 Define an operator for recursive numerals simulating the inversion operator for indexed numerals:

$$
\begin{aligned}
\forall n\, \forall a^{\mathcal{F}(n)}.\ & \textsc{match}\ n\ \textsc{return}\ \mathcal{F}(n) \to \mathbb{T} \\
& [\,0 \Rightarrow \lambda a.\ \bot \\
& \mid Sn \Rightarrow \lambda a.\ (a = \emptyset) + (\Sigma a'.\ a = {}^{\circ}a') \\
&]\, a
\end{aligned}
$$

28 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.

28.1 Binary 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:* Is $x < y$ provable if $x \mathrel{\dot{<}} y$ is derivable?
- *Completeness:* Is $x \mathrel{\dot{<}} y$ derivable if $x < y$ is provable?

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 $3 \mathrel{\dot<} 6$:

$$\dfrac{\dfrac{}{3 \mathrel{\dot<} 4} \qquad \dfrac{\dfrac{}{4 \mathrel{\dot<} 5} \quad \dfrac{}{5 \mathrel{\dot<} 6}}{4 \mathrel{\dot<} 6}}{3 \mathrel{\dot<} 6} \qquad\qquad \dfrac{\dfrac{\dfrac{}{3 \mathrel{\dot<} 4} \quad \dfrac{}{4 \mathrel{\dot<} 5}}{3 \mathrel{\dot<} 5} \qquad \dfrac{}{5 \mathrel{\dot<} 6}}{3 \mathrel{\dot<} 6}$$

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 inductive 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 inductive 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 L_1 and L_2, we see that the second argument of L is an index that is instantiated in the target type of L_1. On the other hand, the first argument of L is a parameter since it not instantiated in the target types of L_1 and L_2. We express the fact that the first argument of L is a parameter as usual in the declaration of the inductive type constructor L:

$$\begin{aligned} &\mathsf{L}\, (x : \mathsf{N}) : \; \mathsf{N} \to \mathbb{T} \; := \\ &\mid \mathsf{L}_1 : \; \mathsf{L}\, x (\mathsf{S} x) \\ &\mid \mathsf{L}_2 : \; \forall yz.\, \mathsf{L}\, xy \to \mathsf{L}\, yz \to \mathsf{L}\, xz \end{aligned}$$

Recall that the parameter-index distinction matters since indices must not be instantiated in the types of discriminations.

Note that the first argument of L is instantiated in the fourth argument type of L_2. We acknowledge this fact by saying that the first argument of L is a **nonuniform parameter**. So far we have only seen **uniform parameters**, which are instantiated neither in the argument types nor the target types of value constructors. The difference between the two kinds of parameters shows in the targets and clauses of eliminators, where nonuniform parameters are quantified locally like indices. In contrast, uniform parameters are quantified only once in the prefix of the eliminator.

We remark that the proof assistant Coq realizes the parameter-index distinction by declaration, making it possible to declare parameters as indices (but not vice versa).

We can now do formal proofs concerning the derivation system L. We first prove **completeness**:

$$\forall xy.\ x < y \to \mathsf{L}xy \tag{28.1}$$

The proof succeeds by induction on y with x fixed. The base case follows by computational falsity elimination. 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 $\mathsf{L1}$ 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**

$$\forall xy.\ \mathsf{L}xy \to x < y \tag{28.2}$$

by induction on the derivation of $\mathsf{L}xy$. This give us the proof obligations

$$\begin{gathered}
x < \mathsf{S}x\\
x < y \to y < z \to x < z
\end{gathered}$$

which are both obvious. Note that the obligations are obtained from the derivations rules by 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.

We can define an **inversion function** for L as follows:

$$\begin{aligned}
\mathsf{inv}:\ & \forall xy.\ \mathsf{L}xy \to (y = \mathsf{S}x) + \Sigma z.\ \mathsf{L}xz \times \mathsf{L}zy \\
& \mathsf{inv}\,x(\mathsf{S}x)(\mathsf{L}_1 x) \approx \mathsf{L\,Q} \\
& \mathsf{inv}\,xz(\mathsf{L}_2 xyzab) \approx \mathsf{R}\,(y,(a,b))
\end{aligned}$$

With this inversion function it is easy to prove **constructor disjointness**:

$$\mathsf{L}_1 x = \mathsf{L}_2 xy(\mathsf{S}x)ab \to \bot$$

We can also prove injectivity of L_2

$$\forall xyz\,\forall aba'b'.\ \mathsf{L}_2 xzyab = \mathsf{L}_2 xzya'b' \to (a,b) = (a',b')$$

if we assume *dependent pair injectivity for numbers*:

$$\forall p^{\mathsf{N}\to\mathbb{T}}\,\forall x\,\forall ab^{px}.\ (x,a)_p = (x,b)_p \to a = b$$

Dependent pair injectivity for numbers will be shown in §29.3 once we have introduced inductive equality.

Exercise 28.1.1 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.

Exercise 28.1.2 Do the following with the proof assistant:

a) Define the inversion function specified above and check that it satisfies the computational equalities specified.
b) Prove $\mathsf{L}_1 x = \mathsf{L}_2 xy(\mathsf{S}x)ab \to \bot$.
c) Prove injectivity of L_2 assuming dependent pair injectivity for numbers. You find a detailed discussion of this proof in §29.5.

28.2 Linear Derivation System for Comparisons

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 needed for completeness, since every defining rule adds a clause to the eliminator for the system (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 new 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 of the derivation system

$$\forall xy.\, \mathsf{L}xy \to x < y$$

follows as before by induction on the derivation of $\mathsf{L}xy$, requiring soundness of each of the two rules. **Completeness** of the system

$$\forall xy.\, x < y \to \mathsf{L}xy$$

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**

$$\begin{aligned}
\mathsf{inv} : \ & \forall xy\ \forall a^{\mathsf{L}xy}.\ \text{MATCH } x, y \text{ RETURN } \mathsf{L}xy \to \mathbb{T} \\
& \quad \mid 0, \mathsf{S}y \ \Rightarrow\ \lambda a.\ a = \mathsf{L}_1 y \\
& \quad \mid \mathsf{S}x, \mathsf{S}y \ \Rightarrow\ \lambda a.\ \Sigma a'.\ a = \mathsf{L}_2 xya' \\
& \quad \mid _, _ \ \Rightarrow\ \lambda a. \bot \\
& \quad]\, a \\
\mathsf{inv}\, 0\, (\mathsf{S}y)(\mathsf{L}_1 y) & \approx \mathsf{Q} \\
\mathsf{inv}\, (\mathsf{S}x)\, (\mathsf{S}y)(\mathsf{L}_2 xya) & \approx (a, \mathsf{Q})
\end{aligned}$$

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 28.2.1 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 28.2.2 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 28.2.3 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 28.2.4 Prove that the constructor L_2 is injective in its third argument. Hint: Use derivation uniqueness.

Exercise 28.2.5 Given an equality decider for the derivation types $\mathsf{L}xy$. Hint: Use derivation uniqueness.

Exercise 28.2.6 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 §29.4 in Chapter 29 on inductive equality. Using an inversion operator with a type cast we will prove derivation uniqueness in §29.4.

Exercise 28.2.7 (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 $\mathsf{E}(n)$.

28.3 Derivation Systems for GCDs

Recall that a **gcd relation** (Definition 17.3.1) 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. $pxyz \to pyxz$ *symmetry rule*
3. $x \leq y \to px(y-x)z \to pxyz$ *subtraction rule*

We will prove the following results for gcd relations not using the results previously shown for gcd relations:

1. There is an inductively defined gcd relation G.
2. G is contained in every gcd relation.
3. There is a function respecting G (and hence every gcd relation).
4. All functional gcd relations agree with *G*.
5. G is functional.

We define the indexed inductive predicate $\mathsf{G} : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathbb{P}$ with three derivation rules mimicking the conditions for gcd relations:

$$\mathsf{G}_1\ \frac{}{\mathsf{G}\,0yy} \qquad \mathsf{G}_2\ \frac{\mathsf{G}\,xyz}{\mathsf{G}\,yxz} \qquad \mathsf{G}_3\ \frac{x \le y \qquad \mathsf{G}\,x(y-x)z}{\mathsf{G}\,xyz}$$

The rules yield an indexed type family with three indices.[1] We have defined G as a predicate rather than a type function so that G directly qualifies as a gcd relation. It is also be possible to define G as a type function and show that its truncation is a gcd predicate.

First we show that G is the least gcd relation (up to equivalence).

Fact 28.3.1 (Containment) G is a gcd relation contained in every 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 conditions for gcd relations instantiated for p. ■

Next we show that there is a function respecting G.

Fact 28.3.2 (Totality) $\forall xy\, \Sigma z.\, \mathsf{G}\,xyz$.

Proof By size recursion on $x + y$. For $x = 0$ or $y = 0$ the claim follows with G_1 and G_2. Otherwise, we have $x \le y$ without loss of generality (because of G_2). The inductive hypothesis yields z such that $\mathsf{G}\,x(y-x)z$. The claim follows with G_3. ■

Corollary 28.3.3 (Agreement) G agrees with every functional gcd relation.

Proof Follows with Facts 28.3.1 and 28.3.2. ■

[1] There is the possibility to declare the second or third argument of G as a nonuniform parameter, but we prefer the variant with three indices.

It remains to show that G is functional. The functionality of *G* can be obtained straightforwardly from the existence of some functional gcd relation. We give two constructions of functional gcd relations in Chapter 17 (concrete gcd relation and step indexing). We will not use these results here and prove that G is functional just relying on methods for indexed inductive families. Our proof is based on a deterministic variant G′ of G defined with the following rules:

$$G'_1 \; \frac{}{G'\,0yy} \qquad\qquad G'_2 \; \frac{}{G'\,(Sx)0(Sx)}$$

$$G'_3 \; \frac{x \le y \qquad G'\,(Sx)(y-x)z}{G'\,(Sx)(Sy)z} \qquad\qquad G'_4 \; \frac{y < x \qquad G'\,(x-y)(Sy)z}{G'\,(Sx)(Sy)z}$$

We will show that G′ is a functional gcd relation. We may see G′ as a relational reformulation of the procedural specification of GCDs (Definition 17.3.3).

Fact 28.3.4 (Symmetry) $\forall xyz.\ G'\,xyz \to G'\,yxz$.

Proof By induction on the derivation of $G'\,xyz$. The interesting case is G'_3. We distinguish between $x < y$ and $x = y$. Case $x < y$ follows with G'_4 and the inductive hypothesis. For $x = y$ we have to show $G'(Sx)(x-x)z \to G'(Sx)(Sx)z$, which follows by G'_3. ∎

Fact 28.3.5 G′ is a gcd relation.

Proof The first condition is the first rule fo G′. The second condition is Fact 28.3.4. The third condition follows by case analysis on x and y and the third rule for G′. ∎

To show functionality of G′, we shall use an inversion operator with the type:

$$\begin{aligned}\forall xyz\,\forall a^{G'xyz}.\ &\text{MATCH } x, y\\ &[\,0,\,y \Rightarrow z = y\\ &\mid Sx,\,0 \Rightarrow z = Sx\\ &\mid Sx,\,Sy \Rightarrow \text{IF } \ulcorner x \le y \urcorner \text{ THEN } G'\,(Sx)(y-x)z \text{ ELSE } G'\,(x-y)(Sy)z\\ &]\end{aligned}$$

Defining such an operator is routine.

Fact 28.3.6 G′ is functional.

Proof We show $\forall xyzz'.\ G'\,xyz \to G'\,xyz' \to z = z'$ by induction on the derivation of $G'\,xyz$ and inversion of $G'\,xyz'$. All four obligations are straightforward. ∎

Corollary 28.3.7 G and G′ agree. Hence G′ is functional.

Proof Follows with Corollary 28.3.3 and Facts 28.3.5 and 28.3.6. ∎

Exercise 28.3.8 Prove $\mathsf{G}\,x\,x\,x$ and $\mathsf{G}\,1\,y\,1$.

Exercise 28.3.9 (Inductive method for procedural specifications)
The development of this section suggests a method for constructing functions satisfying procedural specifications using indexed inductive types:

1. Translate the procedural specification Γ into an indexed inductive predicate γ.
2. Construct a function g respecting γ using size recursion.
3. Show that γ is functional by induction on and inversion of derivations.
4. Show g satisfies Γ.

Execute the method for the procedural specification of a GCD function given by Definition 17.3.3. Hint: The proof for (4) is similar to the proof of Fact 17.3.8.

28.4 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 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:

$$s \subseteq t \;:=\; \forall A.\; A \vdash s \to A \vdash t$$
$$s \equiv t \;:=\; \forall A.\; A \vdash s \Leftrightarrow A \vdash t$$

An easy to show inclusion is

$$s \subseteq s^* \tag{28.3}$$

(only constructor applications and rewriting with $A +\!\!+ [] = A$ are needed). More challenging is the inclusion

$$s^* \cdot s^* \subseteq s^* \tag{28.4}$$

We need an inversion function

$$A \vdash s \cdot t \to \Sigma A_1 A_2.\, (A = A_1 +\!\!+ A_2) \times (A_1 \vdash s) \times (A_2 \vdash t) \tag{28.5}$$

and a lemma

$$A \vdash s^* \to B \vdash s^* \to A +\!\!+ B \vdash s^* \tag{28.6}$$

The inversion function can be obtained as an instance of a more general **inversion operator**

$$\begin{aligned}
\forall As.\; A \vdash s \;\to\; & \text{MATCH } s \\
& [\, x \Rightarrow A = [x] \\
& \mid \mathbf{0} \Rightarrow \bot \\
& \mid \mathbf{1} \Rightarrow A = [] \\
& \mid u + v \Rightarrow (A \vdash u) + (A \vdash v) \\
& \mid u \cdot v \Rightarrow \Sigma A_1 A_2.\; (A = A_1 +\!\!+ A_2) \times (A_1 \vdash u) \times (A_2 \vdash v) \\
& \mid u^* \Rightarrow (A = []) + \Sigma A_1 A_2.\; (A = A_1 +\!\!+ A_2) \times (A_1 \vdash u) \times (A_2 \vdash u^*) \\
&]
\end{aligned}$$

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 (28.6). 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:

$$\forall As.\, A \vdash s \to \text{MATCH } s\; [\, s^* \Rightarrow B \vdash s^* \to A +\!\!+ B \vdash s^* \mid \_ \Rightarrow \top \,]$$

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 $\forall As.\ A \vdash s \to \mathbb{T}$ is routine. We remark that a weaker eliminator with a type function $\mathcal{L}(\mathsf{N}) \to \mathtt{exp} \to \mathbb{T}$ suffices.

We now have (28.4). A straightforward consequence is

$$s^* \cdot s^* \equiv s^*$$

A less obvious consequence is the equivalence

$$(s^*)^* \equiv s^* \tag{28.7}$$

saying that the star operation is idempotent. Given (28.3), it suffices to show

$$A \vdash (s^*)^* \to A \vdash s^* \tag{28.8}$$

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 $A_1 + A_2 \vdash s^*$. The inductive hypothesis gives us $A_2 \vdash s^*$, which gives us the claim using (28.6).

The above proof is informal since the inductive premise $A \vdash (s^*)^*$ is index constrained. A formal proof succeeds with the reformulation

$$\forall As.\ A \vdash s \ \to\ \text{MATCH}\ s\ [\ (s^*)^* \Rightarrow A \vdash s^* \mid \_ \Rightarrow \top\ ]$$

Exercise 28.4.1 (Certifying solver)
Define a certifying solver $\forall s.\ (\Sigma A.\ A \vdash s) + (\forall A.\ A \vdash s \to \bot)$.

Exercise 28.4.2 (Restrictive star rule) The second derivation rule for star expressions can be replaced 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 $A \vdash s$: $\forall As.\ A \dot{\vdash} s \Leftrightarrow A \vdash s$.

Exercise 28.4.3 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 28.4.4 (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:

$$\begin{aligned}
\mathcal{R}\, x\, A &:= (A = [x]) \\
\mathcal{R}\, \mathbf{0}\, A &:= \bot \\
\mathcal{R}\, \mathbf{1}\, A &:= (A = []) \\
\mathcal{R}\, (s + t)\, A &:= \mathcal{R} s A + \mathcal{R} t A \\
\mathcal{R}\, (s \cdot t)\, A &:= \Sigma A_1 A_2.\, (A = A_1 +\!\!\!+ A_2) \times \mathcal{R} s A_1 \times \mathcal{R} t A_2 \\
\mathcal{R}\, (s^*)\, A &:= \Sigma n.\, \mathcal{P}\, (\mathcal{R} s) n A \\
\\
\mathcal{P}\, \varphi\, 0\, A &:= (A = []) \\
\mathcal{P}\, \varphi\, (\mathsf{S} n)\, A &:= \Sigma A_1 A_2.\, (A = A_1 +\!\!\!+ A_2) \times \varphi A_1 \times \mathcal{P}\, \varphi\, n\, A
\end{aligned}$$

a) Prove $\mathcal{R}\, s A \Leftrightarrow A \vdash s$.

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.

28.5 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 [5].

A function $D : \mathsf{N} \to \mathsf{exp} \to \mathsf{exp}$ is a **derivation function** if

$$\forall x A s.\;\; x :: A \vdash s \Leftrightarrow A \vdash D x s$$

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 28.5.1 $\forall s.\, \mathcal{D}([] \vdash s)$.

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 28.5.2 $\forall As.\ \mathcal{D}(A \vdash s)$ provided we have a derivation function.

Proof By recursion on A using Fact 28.5.1 in the base case and the derivation function in the cons case. ∎

We define a derivation function D as follows:

$$
\begin{aligned}
&D : \mathsf{N} \to \mathsf{exp} \to \mathsf{exp} \\
&\quad Dxy := \text{IF } \ulcorner x = y \urcorner \text{ THEN } \mathbf{1} \text{ ELSE } \mathbf{0} \\
&\quad Dx\,\mathbf{0} := \mathbf{0} \\
&\quad Dx\,\mathbf{1} := \mathbf{0} \\
&\quad Dx\,(s+t) := Dxs + Dxt \\
&\quad Dx\,(s \cdot t) := \text{IF } \ulcorner [] \vdash s \urcorner \text{ THEN } Dxs \cdot t + Dxt \text{ ELSE } Dxs \cdot t \\
&\quad Dx\,(s^*) := Dxs \cdot s^*
\end{aligned}
$$

It remains to show that D is a derivation function. For this proof we need a strengthened inversion lemma for star expressions.

Lemma 28.5.3 (Eager star inversion)
$\forall xAs.\ x :: A \vdash s^* \to \Sigma A_1 A_2.\ A = A_1 +\!\!\!+ A_2 \times x :: A_1 \vdash s \times A_2 \vdash s^*$.

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 28.5.4 (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 28.5.3. ∎

Corollary 28.5.5 $\forall As.\ \mathcal{D}(A \vdash s)$.

Proof Follows with Fact 28.5.2 and Theorem 28.5.4. ∎

28.6 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

$$\frac{(A, B) \in C}{\mathsf{post}\ C\, A\, B} \qquad\qquad \frac{(A, B) \in C \qquad \mathsf{post}\ C\, A'\, B'}{\mathsf{post}\ C\ (A + A')\ (B + B')}$$

Note that $\mathsf{post}\, CAB$ 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

$$\forall C.\ \mathcal{D}(\exists A.\ \mathsf{post}\, CAA) \tag{28.9}$$

Since Coq's type theory can only define computable functions, we can conclude that no function of type (28.9) is definable.

29 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.

29.1 Basic Definitions

We define inductive equality as an inductive predicate with two parameters and one index:

$$\begin{aligned}&\mathsf{eq}\,(X:\mathbb{T},\ x:X):X\to\mathbb{P}\ ::=\\&\mid \mathsf{Q}:\ \mathsf{eq}\ X\ x\ x\end{aligned}$$

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

$$\begin{aligned}&C:\ \forall X^{\mathbb{T}}\ \forall x^X\ \forall p^{X\to\mathbb{T}}\ \forall y.\ x=y\to px\to py\\&C\,Xxp\,\_\,(\mathsf{Q}\_)\,a\ :=\ a \qquad\qquad : px\end{aligned}$$

$$\begin{aligned}&\mathcal{J}:\ \forall X^{\mathbb{T}}\ \forall x^X\ \forall p^{\forall y.\ x=y\to\mathbb{T}}.\ px(\mathsf{Q}x)\to\forall ye.\ pye\\&\mathcal{J}\,Xxpa\,\_\,(\mathsf{Q}\_)\ :=\ a \qquad\qquad : px(\mathsf{Q}x)\end{aligned}$$

called **cast operator** and **full eliminator**. 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 $C_p ea$ 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

$$\forall p^{X\to\mathbb{T}}\, \forall e^{x=y} \forall a^{px}.\;\; C_p ea \;\approx\; \mathcal{J}(\lambda y\_.py)aye$$

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:

$$\begin{aligned}
&\sigma :\; x = y \to y = x\\
&\sigma e \;:=\; C_{(\lambda y.y=x)}\, e\,(\mathsf{Q}x)\\[1ex]
&\tau :\; x = y \to y = z \to x = z\\
&\tau e \;:=\; C_{(\lambda y.y=z\to x=z)}\, e\,(\lambda e.e)\\[1ex]
&\varphi :\; x = y \to fx = fy\\
&\varphi e \;:=\; C_{(\lambda y.fx=fy)}\, e\,(\mathsf{Q}(fx))
\end{aligned}$$

It also turns out that these operators satisfy familiar looking algebraic laws.

Exercise 29.1.1 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 29.1.2 (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 29.1.3 Show that $\mathcal{J}$ is more general that C by defining C with $\mathcal{J}$.

Exercise 29.1.4 Prove $(\mathbf{T} = \mathbf{F}) \to \forall X^{\mathbb{T}}.\ X$ not using falsity elimination.

Exercise 29.1.5 (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 (§5.5).

29.2 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 [16] 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 [14] (§29.3) 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]

[1]We acknowledge the help of Gaëtan Gilbert, (Coq Club, November 13, 2020).

Fact 29.2.1 $\mathsf{UIP}(X) \to \mathsf{UIP}'(X)$.

Proof Instantiate $\mathsf{UIP}(X)$ with $y := x$ and $e' := \mathsf{Q}x$. ∎

Fact 29.2.2 $\mathsf{UIP}'(X) \to \mathsf{K}(X)$.

Proof Instantiate $\mathsf{UIP}'(X)$ with e from $\mathsf{K}(X)$ and rewrite. ∎

Fact 29.2.3 $\mathsf{K}(X) \to \mathsf{CD}(X)$.

Proof Apply $\mathsf{K}(X)$ to $\forall e^{x=x}.\ Cea = a$. ∎

Fact 29.2.4 $\mathsf{CD}(X) \to \mathsf{DPI}(X)$.

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 29.2.5 $\mathsf{DPI}(X) \to \mathsf{UIP}'(X)$.

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 29.2.6 $\mathsf{UIP}'(X) \to \mathsf{UIP}(X)$.

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 29.2.7 $\mathsf{UIP}(X)$, $\mathsf{UIP}'(X)$, $\mathsf{K}(X)$, $\mathsf{CD}(X)$, and $\mathsf{DPI}(X)$ are equivalent.

Proof Immediate by the preceding facts. ∎

Exercise 29.2.8 Verify the above proofs with a proof assistant to appreciate the subtleties.

Exercise 29.2.9 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 29.2.10 Prove that dependent pair types are discrete if their component types are discrete: $\forall X\, \forall p^{X\to\mathbb{T}}.\ \mathcal{E}(X) \to (\forall x.\ \mathcal{E}(px)) \to \mathcal{E}(\mathsf{sig}\,p)$.

29.3 Hedberg's Theorem

We will now prove Hedberg's theorem [14]. 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 29.3.1 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 29.3.2 (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 29.1.2.

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 29.3.3 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 29.3.4 (Hedberg) Every discrete type satisfies UIP.

Proof Lemma 29.3.3 and Lemma 29.3.2. ■

Corollary 29.3.5 Every discrete type satisfies DPI.

Proof Theorems 29.3.4 and 29.2.7. ■

Exercise 29.3.6 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 29.3.7 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 29.3.8 Assume FE and show that $\mathsf{N} \to \mathsf{B}$ satisfies UIP.
Hint: Use Exercises 29.3.7 and 12.4.12.

29.4 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

$$\begin{aligned}
\forall xy\,\forall a^{\mathsf{L}xy}.\ &\text{MATCH } y \text{ RETURN } \mathsf{L}\,xy \to \mathbb{T} \\
&[\,0 \Rightarrow \lambda a.\ \bot \\
&\mid \mathsf{S}y' \Rightarrow \lambda a^{\mathsf{L}x(\mathsf{S}y')}.\ (\Sigma e^{y'=x}.\ Cea = \mathsf{L}_1 x) + (\Sigma a'.\ a = \mathsf{L}_2 xy'a') \\
&]\,a
\end{aligned}$$

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:

$$\mathsf{L}\,xx \to \bot \tag{29.1}$$

This fact follows from a more semantic fact

$$\mathsf{L}\,xy \to x < y \tag{29.2}$$

which follows by induction on $\mathsf{L}\,xy$. We don't have a direct proof of (29.1).

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 (29.1). 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

$$\forall xy\ \forall a^{\mathsf{L}\,xy}.\ \text{MATCH}\ y\ [\,0 \Rightarrow \bot \mid \mathsf{S}y' \Rightarrow x \neq y' \to \mathsf{L}\,xy'\,]$$

by discriminating on a.

Exercise 29.4.1 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.

29.5 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:

$$\forall a^{\mathsf{K}xz}\,\forall b^{\mathsf{K}zy}.\quad \mathsf{K}_2xzyab = \mathsf{K}_2xzya'b' \to (a,b) = (a',b') \tag{29.3}$$

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 (29.3) by applying $\mathsf{K}_{\mathsf{inv}}$ using feq to both sides of the assumed equation of (29.3), 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 29.5.1 Prove injectivity of the constructors for sum using feq.

Exercise 29.5.2 Prove injectivity of K_2 using a customized inversion operator employing an option type rather than a sum type.

Exercise 29.5.3 Prove injectivity of K_2 with the dependent elimination tactic of Coq's Equations package.

Exercise 29.5.4 Define the full inversion operator for K.

Exercise 29.5.5 Prove $\mathsf{K}xy \Leftrightarrow x < y$.

Exercise 29.5.6 Prove that there is no function $\forall xy.\,\mathsf{K}xy \to \Sigma z.\,\mathsf{K}xz \times \mathsf{K}zy$.

29.6 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 29.6.1 $\mathsf{UIP}\,X\to\forall xy^X\,\forall ab^{\mathsf{id}\,xy}.\ \mathsf{id}\,ab.$

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 29.6.2 Prove the converse direction of Fact 29.6.1.

Exercise 29.6.3 Prove Hedberg's theorem for general inductive equality. Do not make use of propositional types.

Exercise 29.6.4 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.

29.7 Notes

The dependently typed algebra of identity proofs identified by Hofmann and Streicher [16] plays an important role in homotopy type theory [26], 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. [18]. That basic type theory cannot prove UIP was discovered by Hofmann and Streicher [16] in 1994 based on a so-called groupoid interpretation.

30 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.

30.1 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

$$\begin{aligned}
&\forall X^{\mathbb{T}}\ \forall p^{\forall n.\ \mathcal{V}_n(X)\to\mathbb{T}}. \\
&p\,0\,\mathsf{Nil} \to \\
&(\forall n x v.\ p n v \to p(\mathsf{S}n)(\mathsf{Cons}\,x v)) \to \\
&\forall n v.\ p n v
\end{aligned}$$

is routine (recursion on the vector v). We will use the universal eliminator for inductive proofs.

We also define an **inversion operator** for vectors

$$\begin{aligned}
&\forall X\,\forall n \forall v^{\mathcal{V}_n(X)}.\\
&\text{MATCH } n \text{ RETURN } \mathcal{V}_n(X) \to \mathbb{T}\\
&[\,0 \Rightarrow \lambda v.\, v = \mathsf{Nil}\\
&\mid \mathsf{S}n' \Rightarrow \lambda v.\, \Sigma x v'.\, v = \mathsf{Cons}\, n'\, x\, v'\\
&]\, v
\end{aligned}$$

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 $\mathcal{V}_{\mathsf{S}n}(X)$.

30.2 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

$$\mathsf{Cons}\ nxv = \mathsf{Cons}\ nx'v' \ \to \ x = x' \land v = v'$$

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:

$$\begin{aligned}
&\mathsf{last} : \forall n.\ \mathcal{V}_{\mathsf{S}n}(X) \to X \\
&\quad\mathsf{last}\ 0 := \mathsf{hd}\,0 && : \mathcal{V}_{\mathsf{S}0}(X) \to X \\
&\mathsf{last}\ (\mathsf{S}n) := \lambda v.\,\mathsf{last}\,n\,(\mathsf{tl}\,(\mathsf{S}n)\,v) && : \mathcal{V}_{\mathsf{SS}n}(X) \to X
\end{aligned}$$

Note that last recurses on n rather than on the vector v, making a direct definition possible.

Another interesting operation on vectors is

$$\begin{aligned}
&\mathsf{sub} : \forall n.\ \mathcal{N}(n) \to \mathcal{V}_n(X) \to X \\
&\quad\mathsf{sub}\,_\,(\mathsf{Z}\,n) := \mathsf{hd}\,n && : \mathcal{V}_{\mathsf{S}n}(X) \to X \\
&\mathsf{sub}\,_\,(\mathsf{U}\,n\,a) := \lambda v.\,\mathsf{sub}\,n\,a\,(\mathsf{tl}\,n\,a) && : \mathcal{V}_{\mathsf{S}n}(X) \to X
\end{aligned}$$

As with lists, $\mathsf{sub}\,nav$ 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 30.2.1 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 30.2.2 (Generalized head and tail) There is a direct realization of a generalized head operation incorporating ideas from the inversion operator:

$$\begin{aligned}
&\mathsf{hd} : \forall n\,\forall v^{\mathcal{V}_n(X)}.\ \text{MATCH}\ n\ [\,0 \Rightarrow \top \mid \mathsf{S}n' \Rightarrow X\,] \\
&\qquad\mathsf{hd}\,_\,\mathsf{Nil} := \mathsf{I} && : \top \\
&\mathsf{hd}\,_\,(\mathsf{Cons}\,n\,x\,v) := x && : X
\end{aligned}$$

Define a generalized tail operation using this idea.

Exercise 30.2.3 (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 30.2.4 (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 30.2.5 Define a map function for vectors and prove its basic properties.

30.3 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:

$$\forall nv.\ \mathsf{len}\,(Lnv) = n \tag{30.1}$$

We can also show that L is injective:

$$\forall n\,\forall v_1 v_2^{\mathcal{V}_n(X)}.\ Lnv_1 = Lnv_2 \ \to\ v_1 = v_2 \tag{30.2}$$

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:

$$\forall A^{\mathcal{L}(X)}\, \Sigma v^{\mathcal{V}_{\mathsf{len}\,A}(X)}.\ L(\mathsf{len}\,A)v = A \tag{30.3}$$

We construct the function (30.3) by induction on A. Function (30.3) 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

$$\forall n\, \forall v^{\mathcal{V}_n(X)}.\, V(Lnv) = v \tag{30.4}$$

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 (30.1). The type checking problem goes away for concrete equations since conversion is strong enough if there are no variables. For instance, the concrete equation

$$L\,3\,(V[1,2,3]) = [1,2,3] \tag{30.5}$$

type checks (since $\mathsf{len}(L3(V[1,2,3])) \approx 3$) and holds by computational equality.

Exercise 30.3.1 Check the above claims with the proof assistant.

Part V

Higher Order Recursion

31 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.

31.1 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**

$$W' : \ \forall p^{X \to \mathbb{T}}.\ (\forall x.\ (\forall y.\ Ryx \to py) \to px) \to \forall x.\ \mathcal{A}_R x \to px$$
$$W' pFx(\mathsf{C}\,\varphi) \ := \ Fx(\lambda yr.\ W' pFy(\varphi yr))$$

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 31.1.1 (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 31.1.2 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 31.1.3 Prove $\mathcal{A}_R(x) \to \neg Rxx$.
Hint: Use well-founded induction with W'.

Exercise 31.1.4 Prove $Rxy \to Ryx \to \neg \mathcal{A}_R(x)$.

Exercise 31.1.5 Show that well-founded relations disallow infinite descend:
$\mathcal{A}_R(x) \to px \to \neg \forall x.\, px \to \exists y.\, py \wedge Ryx$.

Exercise 31.1.6 Assume we narrow the elimination restriction of the underlying type theory such that recursion types are the only propositional types allowing for computational elimination. We can still express an empty propositional type with computational elimination:

$$V : \mathbb{P} \ ::= \ \mathcal{A}_{(\lambda ab^{\top}.\top)}(\mathsf{I})$$

Define a function $V \to \forall X^{\mathbb{T}}.\, X$.

31.2 Well-founded Relations

Fact 31.2.1 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 31.2.2 (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 31.2.3 (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 31.2.4 (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 §17.1 as a special case of the well-founded size recursion operator.

Corollary 31.2.5 (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 31.2.4 and Fact 31.2.1. ∎

There is a story here. We came up with retracts to have an elegant construction of the wellfounded size recursion operator appearing in Corollary 31.2.4. Note that conversion plays an important role in type checking the construction. The proof that retracts of well-founded relations are well-founded (Fact 31.2.3) 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 31.2.6 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 31.1.1).

Exercise 31.2.7 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.

31.3 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 31.3.1 (Uniqueness of recursion types)
Under FE, all recursion types are unique: $\mathsf{FE} \to \forall x\, \forall ab^{\mathcal{A}_R(x)}.\, a = b$.

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 31.3.2 (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 31.3.1. ∎

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 31.3.2. We now prove that all functions satisfying a step function agree if FE is assumed and R is well-founded.

Fact 31.3.3 (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 31.3.4 Note that the proof of Lemma 31.3.1 doubles the quantification of a. Verify that this is justified by the general law $(\forall a.\forall a.pa) \to \forall a.pa$.

$$
\begin{aligned}
&g : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\\
&\qquad g\,0\,y \;=\; y\\
&\quad g\,(\mathsf{S}x)\,0 \;=\; \mathsf{S}x\\
&g\,(\mathsf{S}x)\,(\mathsf{S}y) \;=\; \begin{cases} g\,(\mathsf{S}x)\,(y-x) & \text{if } x \le y\\ g\,(x-y)\,(\mathsf{S}y) & \text{if } x > y\end{cases}
\end{aligned}
$$

guard conditions

$$
\begin{aligned}
x \le y \;&\to\; \mathsf{S}x + (y - x) < \mathsf{S}x + \mathsf{S}y\\
x > y \;&\to\; (x - y) + \mathsf{S}y < \mathsf{S}x + \mathsf{S}y
\end{aligned}
$$

Figure 31.1: Recursive specification of a gcd function

31.4 Example: GCDs

Our second example for the use of well-founded recursion and the unfolding equation is the construction of a function computing GCDs (§17.3). We start with the procedural specification in Figure 31.1. 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

$$
\begin{aligned}
&\sigma :\; \mathsf{N} \times \mathsf{N} \to \mathsf{N}\\
&\sigma(x, y) := x + y
\end{aligned}
$$

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 31.1, 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 31.2.1) and retracts (Fact 31.2.3). Each of the three specifying equations in Figure 31.1 can now be obtained as an instance of the unfolding equation (Fact 31.3.2).

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 31.4.1 Construct a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfying the Ackermann equations (§1.10) using well-founded recursion for the lexical product $<_\mathsf{N} \times <_\mathsf{N}$.

31.5 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 31.5.1 $\mathsf{ext}(F) \to \forall xaa'.\ W'Fxa = W'Fxa'$.

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 31.5.2 (Unfolding equation)
Let R be well-founded. Then $\mathsf{ext}(F) \to \forall x.\ WFx = Fx(\lambda yr.\ WFy)$.

Proof We assume $\mathsf{ext}(F)$ 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 31.5.1. ■

Exercise 31.5.3 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 (§31.3).

b) Prove that the step function for GCDs is extensional (§31.4).

c) Prove that the step function for the Ackermann equations is extensional (Exercise 31.4.1).

Exercise 31.5.4 Show that all functions satisfying an extensional step function for a well-founded relation agree.

31.6 Witness Operator

There is an elegant and instructive construction of an existential witness operator (Chapter 18) 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 31.6.1 $p(x+y) \to \mathcal{A}_R(y)$.

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 31.6.2 $\mathcal{A}_R(x) \to \mathsf{sig}(p)$.

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 31.6.3 (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 31.6.2 it suffices to show $\mathcal{A}_R(0)$. We can now obtain a witness x for p. The claim follows with Lemma 31.6.2. ∎

We may see the construction of an existential witness operator in Chapter 18 as a specialization of the construction shown here where the general recursion types used here are replaced with special purpose linear search types.

Exercise 31.6.4 Prove $\mathcal{A}_R(n) \longleftrightarrow T(n)$.

Exercise 31.6.5 Prove that $\mathcal{A}_R$ yields the elimination lemma for linear search types:

$$\forall q^{\mathsf{N}\to\mathbb{T}}.\ (\forall n.\ (\neg pn \to q(\mathsf{S}n)) \to qn) \to \forall n.\ \mathcal{A}_R(n) \to qn$$

Do the proof without using linear search types.

31.7 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 31.1 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 §31.5.

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.

31.8 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:

$$a \approx Cx(\lambda yr.\, a')$$
$$a \approx Cx(\lambda yr. Cy(\lambda y'r'.\, a''))$$

We formulate the idea with two functions

$$D : \forall x.\, \mathcal{A}_R(x) \to \forall y.\, Ryx \to \mathcal{A}_R(y)$$
$$D\,xa \;:=\; \text{MATCH}\; a\;[\,C\,\varphi \Rightarrow \varphi\,]$$

$$P : \mathsf{N} \to \forall x.\, \mathcal{A}_R(x) \to \mathcal{A}_R(x)$$
$$P\,0xa \;:=\; a$$
$$P\,(\mathsf{S}n)a \;:=\; Cx(\lambda yr.\, Pny(Dxayr))$$

and refer to P as **padding function**. We have, for instance,

$$P(1+n)xa \approx Cx(\lambda y_1 r_1.\, Pny_1(Dxay_1r_1))$$
$$P(2+n)xa \approx Cx(\lambda y_1 r_1.\, Cy_1(\lambda y_2 r_2.\, Pny_2(Dy_1(Dxay_1r_1)y_2r_2)))$$

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 31.8.1 Write a padding function for linear search types (§18.2)

31.9 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 31.9.1 (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:

$$\mathsf{pro}_R(p^{X \to \mathbb{P}}) := \forall x.\ px \to \exists y.\ py \land Ryx$$
$$\mathsf{pro}_R(x^X) := \exists p.\ px \land \mathsf{pro}_R(p)$$

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 31.9.2 (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 31.9.3 (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 31.9.4 (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 31.9.2.

For direction $\leftarrow$ we assume $\neg\exists x.\ \mathsf{pro}_R(x)$ and prove $\mathcal{A}_R(x)$. By Fact 31.9.3 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 31.9.5 $\mathsf{XM} \to (\mathsf{pro}_R(p) \longleftrightarrow \neg\exists x.\ \mathrm{min}_{R,p}(x))$.

Proof For direction $\rightarrow$, we derive a contradiction from the assumptions $\mathsf{pro}_R(p)$, px, and $\forall y.\ py \rightarrow \neg Ryx$. Straightforward.

For direction $\leftarrow$, using XM, we derive a contradiction from the assumptions $\neg\exists x.\ \mathrm{min}_{R,p}(x)$, px, and $H : \neg\exists y.\ py \wedge Ryx$. 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 31.9.6 $\mathsf{XM} \rightarrow (\neg(\exists x.\ \mathsf{pro}_R(x)) \longleftrightarrow \forall p.\ (\exists x.\ px) \rightarrow \exists x.\ \mathrm{min}_{R,p}(x))$.

Proof For direction $\rightarrow$, 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 31.9.5 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 31.9.5 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 31.9.7 (Characterization by existence of minimal elements)
$\mathsf{XM} \rightarrow (\mathsf{wf}(R) \longleftrightarrow \forall p^{X\rightarrow\mathbb{P}}.\ (\exists x.\ px) \rightarrow \exists x.\ \mathrm{min}_{R,p}(x))$.

Proof Facts 31.9.4 and 31.9.6. ∎

The above proofs gives us ample opportunity to contemplate about the role of XM in proofs. An interesting example is Fact 31.9.3, where XM is used to show that an element is either well-founded or progressive.

31.10 Transitive Closure

The **transitive closure** R^+ of a relation $R^{X\rightarrow X\rightarrow\mathbb{P}}$ 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 31.10.1 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 31.10.2 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^+.

31.11 Notes

The inductive definition of the well-founded points of a relation appears in Aczel [1] in a set-theoretic setting. Nordström [23] 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.

32 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.

32.1 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 $\mathsf{T}Xf$ 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 32.1.1 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:

$$\mathcal{L}(X:\mathbb{T}_i):\mathbb{T}_i \;::=\; \mathsf{nil} \mid \mathsf{cons}\,(X, \mathcal{L}(X))$$

$$\begin{array}{rlll}
\mathcal{L}: & \mathbb{T}_i \to \mathbb{T}_i & :\mathbb{T}_{i+1} & \\
\mathsf{nil}: & \mathcal{L}(X) & :\mathbb{T}_i & (X:\mathbb{T}_i) \\
\mathsf{cons}: & X \to \mathcal{L}(X) \to \mathcal{L}(X) & :\mathbb{T}_i & (X:\mathbb{T}_i) \\
\mathsf{nil}: & \forall X^{\mathbb{T}_i}.\, \mathcal{L}(X) & :\mathbb{T}_{i+1} & \\
\mathsf{cons}: & \forall X^{\mathbb{T}_i}.\, X \to \mathcal{L}(X) \to \mathcal{L}(X) & :\mathbb{T}_{i+1} &
\end{array}$$

Write down an analogous table for pairs and options.

32.2 Propositional Aczel Trees

We now note that the definition

$$\mathcal{T}_\mathsf{p}:\mathbb{P} \;::=\; \mathsf{T}_\mathsf{p}\,(X:\mathbb{P},\; X \to \mathcal{T}_\mathsf{p})$$

of the type of **propositional Aczel trees** satisfies the dominance condition since the type of the constructor 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

$$u_\mathsf{p} := \;\mathsf{T}_\mathsf{p}\,\mathcal{T}_\mathsf{p}\,(\lambda s.s)$$

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 $\mathsf{T}_\mathsf{p}\,\mathcal{T}_\mathsf{p}\,(\lambda s.s)$ 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 (§18.2). It seems that the values of computational propositions are always wellfounded.

32.3 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 32.3.1 (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 32.3.2 (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 32.3.3 (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 32.3.4 $\neg\exists R^{\mathcal{T}_p\to\mathcal{T}_p\to\mathbb{P}}.\ \forall sXf.\ Rs(\mathsf{T_p}\,Xf) \longleftrightarrow \exists x.\ fx = s.$

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 32.3.2). ∎

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 32.3.5 Suppose you are allowed exactly one violation of the elimination restriction. Give a proof of falsity.

32.4 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 [9] 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 32.4.1 (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 32.3.4 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 32.4.2 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 32.4.3 Show $\forall P^{\mathbb{P}}.\ P \neq \mathbb{P}$.

32.5 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 32.5.2).

Theorem 32.5.1 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 32.5.2) 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 32.5.2 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 32.5.1.

32.6 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 32.6.1 Every type embeds into itself: $\forall X^{\mathbb{T}} :\ \mathcal{E}XX$.

Fact 32.6.2 $\forall XY^{\mathbb{T}} :\ \neg\mathcal{E}XY \to X \neq Y$.

Fact 32.6.3 $\mathbb{P}$ embeds into no proposition: $\forall P^{\mathbb{P}}.\ \neg\mathcal{E}\mathbb{P}P$.

Proof Follows with Coquand's theorem 32.4.1. ∎

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 32.6.4 (Irreflexivity) $\forall s^{\mathcal{T}}.\, s \notin s$.

Proof Analogous to the proof of Fact 32.3.2. ∎

Recall that we have to construct a contradiction. We embark on a little detour before we construct a universal tree. By Fact 32.6.1 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 32.6.1 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 32.6.5 (Hierachy) $\forall X^U.\ \neg\mathcal{E}UX$.

Exercise 32.6.6 Show $\forall X^U.\ X \neq U$ for all universes U.

Exercise 32.6.7 Let $i \neq j$. Show $\mathbb{T}_i \neq \mathbb{T}_j$.

Exercise 32.6.8 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.

Part VI

Appendices

Appendix: Typing Rules

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

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

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

$$\frac{}{x : u \vdash x : u}$$

$$\frac{}{\vdash \mathbb{P} : \mathbb{T}_2} \qquad\qquad \frac{}{\vdash \mathbb{T}_i : \mathbb{T}_{i+1}} \qquad\qquad \frac{\vdash s : u \qquad \vdash u \subset u'}{\vdash s : u'}$$

$$\frac{}{\vdash \mathbb{P} \subset \mathbb{T}_i} \qquad\qquad \frac{i < j}{\vdash \mathbb{T}_i \subset \mathbb{T}_j} \qquad\qquad \frac{\vdash u : \mathbb{T}_i \qquad \vdash v \subset v'}{\vdash \forall x^u.v \subset \forall x^u.v'}$$

- The constants $\mathbb{P} \subset \mathbb{T}_1 \subset \mathbb{T}_2 \subset \cdots$ are called universes. There is no $\mathbb{T}_0$.
- Computational equality $s \approx t$ is defined with reduction and α- and η-equivalence.
- v^x_t is capture-free substitution.
- Assumptions ($x : u$ before $\vdash$) must be introduced by the rules for $\forall$ and λ.
- Simple function types $u \to v$ are notation for dependent function types $\forall x : u.\, v$ where x does not occur in v.
- $\mathbb{T}_1$ is called Set in Coq.
- Functions whose type ends with the universe $\mathbb{P}$ are called *predicates*.
- Functions whose type ends with a universe are called *type functions* or *type families*.

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 inductive functions. 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 a type obtained with the type constructor. The first $n \geq 0$ arguments of the type constructor may be declared as **parameters**. The remaining arguments of a type constructor are called **indices**.

Parameter condition: Each value constructor must take the parameters of the type constructor as leading arguments and must target the type constructor applied to these arguments. We speak of the **parametric arguments** and the **proper arguments** of a value constructor.

Strict positivity condition: If a value constructor uses the type constructor in an argument type, the 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 proper arguments of the value constructors must be in $\mathbb{T}_i$.

Inductive Function Definitions

An inductive function definition introduces a constant called an **inductive function** together with a system of **defining equations** serving as computation rules. An inductive function must be defined with a functional type, a number of *required arguments*, and a distinguished required argument called the **discriminating argument**. The type of an inductive function 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:

- $cs_1 \dots s_n y_1 \dots y_m$ types the discriminating argument.
- c is a type constructor with $n \geq 0$ parameters and $m \geq 0$ indices.
- **Index condition**: The **index variables** $y_1, \dots, y_m$ must be distinct and must not occur in $s_1, \dots, s_n$.
- **Elimination restriction**: t must be a proposition if c is not computational. A type constructor c is computational if in case it targets $\mathbb{P}$ it has at most one proof constructor d and all proper arguments of d have propositional types.

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 inductive function, the position of discriminating argument, and the number of arguments succeeding the discriminating argument. Each pattern contains exactly two constants, the inductive function and a value constructor in the position of the discriminating argument. Patterns must be linear (no variable appears twice) and must give the index arguments of the inductive function as underlines. The patterns for constructors must omit the parametric arguments of the constructor.

Every defining equation must satisfy the **guard condition**, which constrains the recursion of the inductive function to be structural on the discriminating 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.

The format of inductive function definitions is such that for every inductive type a universal inductive function (a **universal eliminator**) can be obtained taking as arguments continuations for the value constructors of the type. A particular inductive function for the type can then be obtained by providing the particular continuations. If a constructor is recursive, its continuation takes the results of the recursive calls as arguments. Eager recursion is fine since computation terminates. Universal eliminators usually employ return type functions.

Remarks

1. The format for inductive functions is such that **universal eliminators** can be defined that can express all other inductive functions. Inductive functions may also be called *eliminators*.
2. The special case of zero value constructors is redundant. A proposition $\bot$ with an eliminator $\bot \to \forall X^{\mathbb{T}}. X$ can be defined with a single proof constructor $\bot \to \bot$.
3. Assuming type definitions at the computational level, accommodating type definitions also at the propositional level is responsible for the elimination restriction.

4. The dominance condition is vacuously satisfied for propositional type definitions.
5. Defining equations with a secondary case analysis (e.g., subtraction) come as syntactic convenience. They can be expressed with auxiliary functions defined as inductive functions.
6. 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 a recursive abstraction (i.e., fix) is directly followed by a match on the discriminating argument. This excludes a direct definition of Euclidean division. It also excludes the (redundant) eager recursion pattern sometimes used for well-founded recursion in the Coq literature.

Examples

We give for some inductive type families discussed in this text

- the type of the type constructor.
- the type of one of the value constructors.
- the type of the eliminator we have been using (prefix and target, clauses for value constructors omitted).
- The pattern of the defining equation for the eliminator and the given value constructor.

Lists

$$\begin{aligned} \mathcal{L} &: \mathbb{T} \to \mathbb{T} \\ \mathsf{cons} &: \forall X.\, X \to \mathcal{L}(X) \to \mathcal{L}(X) \\ \mathsf{E} &: \forall X. \forall p^{\mathcal{L}(X) \to \mathbb{T}}.\ \ \ldots\ \ \to \forall A.\, pA \\ \mathsf{E}\, X p \cdots (\mathsf{cons}\, x A) &:= \ e\, x A (E \cdots A) \end{aligned}$$

$\mathcal{L}(X)$ has uniform parameter X.

Linear search types

$$\begin{aligned} T &: (\mathsf{N} \to \mathbb{P}) \to \mathsf{N} \to \mathbb{T} \\ \mathsf{C} &: \forall q n.\, (\neg q n \to T q (\mathsf{S} n)) \to T q n \\ \mathsf{E} &: \forall q. \forall p^{\mathsf{N} \to \mathbb{T}}.\ \ \ldots\ \ \to \forall n.\, T q n \to p n \\ \mathsf{E}\, q p \cdots n\, (\mathsf{C} \varphi) &:= \ e\, n (\lambda a. E \cdots (\mathsf{S} n)(\varphi a)) \end{aligned}$$

Tqn has uniform parameter q and nonuniform parameter n.

Hilbert derivation types

$$
\begin{aligned}
\mathcal{H} &: \mathsf{For} \to \mathbb{T} \\
\mathsf{K} &: \forall st.\ \mathcal{H}(s \to t \to s) \\
\mathsf{E} &: \forall p^{\mathsf{For} \to \mathbb{T}}.\ \ldots \to \forall s.\, \mathcal{H}(s) \to ps \\
\mathsf{E}\, p \cdots \_\, (\mathsf{K} st) &:= e\, st
\end{aligned}
$$

$\mathcal{H}(s)$ has index s.

ND derivation types

$$
\begin{aligned}
\vdash &: \mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{T} \\
\mathsf{I}_{\to} &: \forall Ast.\ (s :: A \vdash t) \to (A \vdash (s \to t)) \\
\mathsf{E} &: \forall p^{\mathcal{L}(\mathsf{For}) \to \mathsf{For} \to \mathbb{T}}.\ \ldots \to \forall As.\ (A \vdash s) \to pAs \\
\mathsf{E}\, p \cdots A\, \_\, (\mathsf{I}_{\to} std) &:= e\, Ast(E \cdots (s :: A)td)
\end{aligned}
$$

$A \vdash s$ has nonuniform parameter A and index s.

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).

A **fixed point** of a function $f^{X \to X}$ is a value x^X such that $fx = x$.

Two types X and Y are **in bijection** if there are functions $f^{X \to Y}$ and $g^{Y \to X}$ inverting each other; that is, the **roundtrip equations** $\forall x.\, g(fx) = x$ and $\forall y.\, f(gy) = y$ are satisfied. We define:

$$\mathsf{inv}\, g\, f \;:=\; \forall x.\, g(fx) = x \qquad\qquad g \textbf{ inverts } f$$

For functions $f : \forall x^X.\, px$ we define:

$$\begin{aligned}
\mathsf{injective}\,(f) &:= \forall x x'.\, fx = fx' \to x = x' && \textbf{injectivity}\\
\mathsf{surjective}\,(f) &:= \forall y\, \exists x.\, fx = y && \textbf{surjectivity}\\
\mathsf{bijective}\,(f) &:= \mathsf{injective}\,(f) \land \mathsf{surjective}\,(f) && \textbf{bijectivity}\\
f \equiv f' &:= \forall x.\, fx = fx' && \textbf{agreement}
\end{aligned}$$

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:

$$\begin{aligned}
\mathsf{functional}\,(P) &:= \forall x y y'.\, Pxy \to Pxy' \to y = y' && \textbf{functionality}\\
\mathsf{total}\,(P) &:= \forall x\, \exists y.\, Pxy && \textbf{totality}
\end{aligned}$$

The definitions extend to predicates with $n \geq 2$ arguments as one would expect. To functional relations we may also refer as **unique relations**.

For unary predicates $P, Q : X \to \mathbb{P}$ we define:

$$\begin{aligned}
P \subseteq Q &:= \forall x.\, Px \to Qx && \textbf{respect}\\
P \equiv Q &:= \forall x.\, Px \longleftrightarrow Qx && \textbf{agreement}
\end{aligned}$$

The definitions extend to predicates with $n \geq 2$ arguments as one would expect.

For functions $f : \forall x^X.\, px$ and predicates $P : \forall x^X.\, px \to \mathbb{P}$:

$$f \subseteq P \;:=\; \forall x.\, Px(fx) \qquad\qquad \textbf{respect}$$

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. $P \subseteq Q \to \mathsf{total}\,(P) \to \mathsf{total}\,(Q)$
3. $P \subseteq Q \to \mathsf{total}\,(P) \to \mathsf{functional}\,(Q) \to P \equiv Q$
4. $f \subseteq P \to \mathsf{functional}\,(P) \to (\forall xy.\, Pxy \longleftrightarrow fx = y)$

Appendix: Favorite Problems

Here is a list of problems the author likes to discuss with students in oral exams. The problems are given in the order they appear first in the text.

1. Fibonacci function with iter, uniqueness of procedural specification
2. Russell's law
3. Leibniz symmetry
4. Leibniz equality
5. Constructor laws for numbers
6. Kaminski's equation
7. Eliminator for numbers
8. Equality decider for numbers
9. $N \neq B$
10. Cantor pairing
11. Barber theorem
12. Bijection between sum types and sigma types
13. Skolem correspondence
14. Certifying decider equivalence
15. Bijection and cardinality theorems for finite types
16. N is not finite
17. Pigeonhole theorem for finite types
18. Bijection between $(B \to B)$ and $B \times B$ under FE
19. PE implies PI, SE implies PE
20. Counterexample characterization of XM
21. Classical reasoning for stable claims
22. Antisymmetry of order on numbers from first principles
23. Euclidean division
24. Step-indexed linear search (correctness)
25. Size recursion operator
26. GCD functions and relations (step-indexed, certifying, inductive constructions)
27. Discriminating element lemmas

28. Equivalent nonrepeating lists have equal length
29. Correctness of arithmetic expression compiler
30. Predecessor and constructor laws for indexed numeral types
31. Inversion operator and equality decider for indexed numeral types
32. Decidability of regular expression matching
33. Glivenko's theorem and agreement of ND and Hilbert systems
34. Intuitionistic unprovability of double negation law
35. Inductive equality versus Leibniz equality
36. Hedberg's theorem and $\mathsf{CD}(X) \to \mathsf{DPI}(X)$

Appendix: Exercise Sheets

Below you will find the weekly exercise sheets for the course *Introduction to Computational Logic* as given at Saarland University in the summer semester 2022 (13 weeks of full teaching). The sheets tell you which topics of MPCTT we covered and how much time we spent on them.

Assignment 1

Do the following exercises on paper using mathematical notation and also with the proof assistant Coq. Follow the style of Chapter 1 and the accompanying Coq file gs.v. For each function state the type and the defining equations. Make sure you understand the definitions and proofs you give.

Exercise 1.1 Define an addition function add for numbers and prove that it is commutative.

Exercise 1.2 Define a distance function dist for numbers and prove that it is commutative. Do not use helper functions.

Exercise 1.3 Define a minimum function min for numbers and prove that it is commutative. Do not use helper functions. Prove $\mathsf{min}\, x\, (x + y) = x$.

Exercise 1.4 Define a function fib satisfying the procedural Fibonacci equations. Define the unfolding function for the equations and prove your function satisfies the unfolding equation.

Exercise 1.5 Define an iteration function computing $f^n(x)$ and prove the shift laws $f^{\mathsf{S}n}(x) = f^n(fx) = f(f^n(x))$.

Exercise 1.6 Give the types of the constructors pair and Pair for pairs and pair types. Give the inductive type definition. Define the projections fst and snd and prove the η-law. Define a swap function and prove that it is self-inverting. Do not use implicit arguments.

Want More?

You will find further exercises in Chapter 1 of MPCT. You may for instance define Ackermann functions using either a higher-order helper function or iteration and verify that your functions satisfy the procedural specification given as unfolding function.

Assignment 2

Do the exercises on paper using mathematical notation and also with the proof assistant Coq.

Exercise 2.1 Define a truncating subtraction function using a plain constant definition and a recursive abstraction.

Exercise 2.2 Assume $A := \text{FIX}\, f\, x.\, \lambda y.\, \text{MATCH}\; x\; [\, 0 \Rightarrow y \mid \mathsf{S}x \Rightarrow \mathsf{S}(f x y)\,]$.

a) Gives the types for A, f, x, and y.

b) For each of the following equations, give the normal forms of the two sides and say which reduction rules are needed. Decide whether the equation holds by computational equality.

   (i) $A\,1 = \mathsf{S}$.

   (ii) $A\,2 = \lambda y.\mathsf{SS}y$

   (iii) $(\text{LET}\; f = A\,1\; \text{IN}\; f) = \mathsf{S}$

   (iv) $A = \lambda x y.Axy$

   (v) $A = \text{FIX}\, f\, x.\, \text{MATCH}\; x\; [\, 0 \Rightarrow \lambda y.y \mid \mathsf{S}x \Rightarrow \lambda y.\, \mathsf{S}(f x y)\,]$

Exercise 2.3 Prove the following propositions (diagrams, terms, and Coq). Assume that X, Y, Z are propositions.

a) $X \to Y \to X$

b) $(X \to Y \to Z) \to (X \to Y) \to X \to Z$

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

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

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

f) $\neg\neg(\neg\neg X \to X)$

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

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

i) $(X \wedge Y \to Z) \to (X \to Y \to Z)$

j) $(X \to Y \to Z) \to (X \wedge Y \to Z)$

k) $\neg\neg(X \vee \neg X)$

l) $\neg(X \vee Y) \to \neg X \wedge \neg Y$

m) $\neg X \wedge \neg Y \to \neg(X \vee Y)$

Assignment 3

Do the exercises on paper using mathematical notation and also with the proof assistant Coq.

Exercise 3.1 (Match functions and impredicative characterizations) Give the types and the defining equations for the matching functions for $\bot$, $\wedge$ and $\vee$. Following the types of the matching functions, state the impredicative characterizations for $\bot$, $\wedge$ and $\vee$. Make sure you can prove the impredicative characterizations (proof diagram, proof term, coq script). Type the type arguments of the matching functions with $\mathbb{T}$ (rather than $\mathbb{P}$) if this is possible (elimination restriction). Explain why in the impredicative characterizations all type arguments must be typed with $\mathbb{P}$.

Exercise 3.2 (Exclusive disjunction) Exclusive disjunction $X \oplus Y$ is a logical connective satisfying the equivalence $X \oplus Y \longleftrightarrow (X \wedge \neg Y) \vee (Y \wedge \neg X)$.

a) Give an inductive definition of exclusive disjunction and prove the above equivalence.
b) Define the matching function for inductive exclusive disjunction.
c) Give and verify the impredicative characterization of exclusive disjunction.

Exercise 3.3 (Double negation law) Prove the equivalence

$$(\forall X^{\mathbb{P}}.\, X \vee \neg X) \longleftrightarrow (\forall X^{\mathbb{P}}.\, \neg\neg X \to X)$$

to show that the law of excluded middle is intuitionistically equivalent to the double negation law. Do the proof first with a diagram and then verify your reasoning with Coq.

Exercise 3.4 (Conversion rule) Prove

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

with a diagram and with Coq. Assume $X : \mathbb{T}$ and determine the types of the variables x and y.

Exercise 3.5 (Propositional equality) Assume the constants

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

for propositional equality and prove the following proposition assuming the variable types $x:X$, $y:X$, $z:X$, $f:X \to Y$, $X:\mathbb{T}$, and $Y:\mathbb{T}$:

a) $\mathsf{eq}\,xy \to \mathsf{eq}\,yx$

b) $\mathsf{eq}\,xy \to \mathsf{eq}\,yz \to \mathsf{eq}\,xz$

c) $\mathsf{eq}\,xy \to \mathsf{eq}\,(fx)\,(fy)$

d) $\neg\mathsf{eq}\,\top\,\bot$

e) $\neg\mathsf{eq}\,\mathbf{T}\,\mathbf{F}$

For each occurrence of eq determine the implicit argument.

Assignment 4

Do the exercises on paper using mathematical notation and verify your findings with the proof assistant Coq.

Exercise 4.1 Define the equational constants eq, Q, and R.

Exercise 4.2 MPCTT gives two proofs of transitivity, one using the conversion rule and one not using the conversion rule. Give each proof as a diagram and as a term and verify your findings with the proof assistant Coq.

Exercise 4.3 Define the eliminators for booleans, numbers, and pairs.

Exercise 4.4 (Truncating subtraction)
Define a truncating subtraction function using the eliminator for numbers and not using discrimination. Show that your function agrees with the standard subtraction function from Chapter 1 using the eliminator for numbers.

Exercise 4.5 (Boolean equality decider)
Define a boolean equality decider $\mathsf{eqb} : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$ using the eliminator for numbers and not using discrimination. Show that your function satisfies $eqb\,x\,y = \mathbf{T} \longleftrightarrow x = y$ using the eliminator for numbers. Use this result to show $\forall x y^{\mathsf{N}}.\ x = y \vee x \neq y$.

Exercise 4.6 (Boolean pigeonhole principle)

a) Prove the pigeonhole principle for B: $\forall x y z^{\mathsf{B}}.\ x = y \vee x = z \vee y = z$.

b) Prove Kaminski's equation based on the instance of the boolean pigeonhole principle for $f(fx)$, fx, and x.

Exercise 4.7 (Pair types)

a) Define the eliminator for pair types.

b) Prove that the pair constructor is injective using the eliminator.

c) Use the eliminator to define the projections π_1, π_2 and swap.

d) Prove the eta law using the eliminator.

e) Prove $\mathsf{swap}(\mathsf{swap}\,a) = a$.

Exercise 4.8 (Unit type $\top$)

a) Define the eliminator for $\top$ (following the scheme for B).

b) Prove the pigeonhole principle for $\top$: $\forall x y^{\top}.\ x = y$.

c) Prove $\mathsf{B} \neq \top$.

Exercise 4.9 Show $\mathsf{B} \neq \mathbb{T}$.
We remark that $\mathsf{B} = \mathbb{P}$ cannot be proved or disproved.

Assignment 5

Do the exercises on paper and verify your findings with Coq.

Exercise 5.1 Define the constants ex, E, and $M_\exists$ for existential quantification both inductively and impredicatively.

Exercise 5.2 Give and verify the impredicative characterization of existential quantification.

Exercise 5.3 Give a proof term for $(\exists x.px) \to \neg\forall x.\neg px$ using the constants for existential quantification. Do not use matches.

Exercise 5.4 Prove the following facts about existential quantification:

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

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

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

d) $\neg\neg(\exists x.px) \;\longleftrightarrow\; \neg\forall x.\neg px$

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

f) $(\exists x.px) \wedge Z \longleftrightarrow \exists x.\ px \wedge Z$

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

Exercise 5.5 (Fixed points)

a) Prove that all functions $\top \to \top$ have fixed points.

b) Prove that the successor function $\mathsf{S} : \mathsf{N} \to \mathsf{N}$ has no fixed point.

c) For each type $Y = \bot,\ \mathsf{B},\ \mathsf{B} \times \mathsf{B},\ \mathsf{N},\ \mathbb{P},\ \mathbb{T}$ give a function $Y \to Y$ that has no fixed point.

d) State and prove Lawvere's fixed point theorem.

Exercise 5.6 (Intuitionistic drinker) Using excluded middle, one can argue that in a bar populated with at least one person one can always find a person such that if this person drinks milk everyone in the bar drinks milk:

$$\forall X^{\mathbb{T}}\ \forall p^{X \to \mathbb{P}}.\ (\exists x^X.\top) \to \exists x.\ px \to \forall y.\, py$$

The fact follows intuitionistically once two double negations are inserted:

$$\forall X^{\mathbb{T}}\ \forall p^{X \to \mathbb{P}}.\ (\exists x^X.\top) \to \neg\neg\exists x.\ px \to \forall y.\neg\neg\, py$$

Prove the intuitionistic version.

Exercise 5.7 Give the procedural specification for the Fibonacci function as an unfolding function and prove that all functions satisfying the unfolding equation agree.

Exercise 5.8 (Puzzle) Give two types that satisfy and dissatisfy the predicate $\lambda X^{\mathbb{T}}.\ \forall f g^{X \to X}\ \forall x y^{X}.\ f x = y \vee g y = x$.

Assignment 6

Exercise 6.1 (Constructor laws for sum types)

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

Exercise 6.2 (Sum and sigma types)

a) Define the universal eliminator for sum types and use it to prove $\forall a^{X+Y}.\ (\Sigma x.\ a = \mathsf{L}\,x) + (\Sigma y.\ a = \mathsf{R}\,y)$.

b) Define the projections π_1 and π_2 for sigma types.

c) Write the eta law $\forall a^{\mathsf{sig}\,p}.\ a = (\pi_1 a, \pi_2\, a)$ for sigma types without notational sugar and without implicit arguments and fully quantified.

d) Define the universal eliminator for sigma types and use it to prove the eta law.

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

Exercise 6.3 (Certifying division by 2)

Define a function $\forall x^{\mathsf{N}}\,\Sigma n.\ (x = n \cdot 2) + (x = \mathsf{S}(n \cdot 2))$.

Exercise 6.4 (Certifying distance function)

Assume a function $\forall xy^{\mathsf{N}}\,\Sigma z.\ \ (x + z = y) + (y + z = x)$ and use it to define functions f as follows. Verify that your functions satisfy the specifications.

a) $fxy = x - y$

b) $fxy = \mathbf{T} \longleftrightarrow x = y$

c) $fxy = (x - y) + (y - x)$

d) $fxy = \mathbf{T} \longleftrightarrow (x - y) + (y - x) \neq 0$

Exercise 6.5 (Certifying deciders)

Define functions as follows.

a) $\forall XY^{\mathbb{T}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X + Y)$.

b) $\forall X^{\mathbb{T}}.\ (\mathcal{D}(X) \to \bot) \to \bot$.

c) $\forall X^{\mathbb{T}}\, f^{X \to \mathsf{B}}\, x^X.\ \mathcal{D}(fx = \mathbf{T})$.

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

Exercise 6.6 (Bijectivity)

a) Prove $\mathcal{B}\ \mathsf{B}\ (\top + \top)$.

b) Prove $(\mathcal{B}\ \mathsf{B}\ \top) \to \bot$.

c) Prove $\mathcal{B}\ (X \times Y)\ (\mathsf{sig}\,(\lambda x^X.Y))$.

d) Prove $\mathcal{B}\ (X + Y)\ (\mathsf{sig}\,(\lambda b^{\mathsf{B}}.\ \text{IF}\ b\ \text{THEN}\ X\ \text{ELSE}\ Y))$.

e) Find a type X for which you can prove $\mathcal{B}\,X\,(X + \top)$.

f) Assume function extensionality and prove $\mathcal{B}\,(\top \to \top)\ \top$.

g) Assume function extensionality and prove $\mathcal{B}\,(\mathsf{B} \to \mathsf{B})\ (\mathsf{B} \times \mathsf{B})$.

Assignment 7

Do the proofs with the proof assistant and explain the proof ideas on paper.

Exercise 7.1 (Option types)

a) State and prove the constructor laws for option types.

b) Give the universal eliminator for option types.

c) Prove $\mathcal{B}\,(\mathcal{O}(X))\,(X + \top)$.

d) Prove $\mathcal{E}(X) \Leftrightarrow \mathcal{E}(\mathcal{O}(X))$.

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

f) Prove $\forall f^{X \to \mathcal{O}(Y)}.\ (\forall x.\ fx \neq \emptyset) \to \forall x \Sigma y.\ fx = {}^\circ y$.

g) Prove $\forall x^{\mathcal{O}^3(\bot)}.\ x = \emptyset \lor x = {}^\circ\emptyset \lor x = {}^{\circ\circ}\emptyset$.

h) Prove $\forall f^{\mathcal{O}^3(\bot) \to \mathcal{O}^3(\bot)}\ \forall x.\ f^8(x) = f^2(x)$.

i) Find a type X and functions $f : X \to \mathcal{O}(X)$ and $g : \mathcal{O}(X) \to X$ such that you can prove $\mathsf{inv}\, g\, f$ and disprove $\mathsf{inv}\, f\, g$.

Exercise 7.2 (Finite types)

Let d be a certifying decider for $p : \mathcal{O}^n(\bot) \to \mathbb{T}$. Prove the following:

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

b) $\mathcal{D}(\Sigma x.px)$.

c) $(\Sigma x.px) + (\forall x.px \to \bot)$.

d) The type N of numbers is not finite.

Exercise 7.3 (Pigeonhole)

Prove $\forall f^{\mathcal{O}^{\mathsf{S}n}(\bot) \to \mathcal{O}^n(\bot)}.\Sigma ab.\ a \neq b \land fa = fb$.

Intuition: If $n + 1$ pigeons are in n holes, there must be a hole with at least two pigeons in it.

Exercise 7.4 (Function extensionality)

Assume function extensionality and prove the following.

a) $\forall f^{\top \to \top}.\ f = \lambda a^\top.a$.

b) $\mathcal{B}\,(\top \to \top)\,\top$.

c) $\mathsf{B} \neq (\top \to \top)$.

d) $\mathcal{E}(\mathsf{B} \to \mathsf{B})$.

Exercise 7.5 (Proof irrelevance)

a) Prove $\mathsf{PE} \to \mathsf{PI}$.

b) Suppose there is a function $f : (\top \lor \top) \to \mathsf{B}$ such that $f(\mathsf{L\,I}) = \mathbf{T}$ and $f(\mathsf{R\,I}) = \mathbf{F}$. Prove $\neg\,\mathsf{PI}$. Why can't you define f inductively?

Exercise 7.6 (Set extensionality)

We define *set extensionality* as $\mathsf{SE} := \forall X^{\mathbb{T}}\, \forall p q^{X \to \mathbb{P}}.\ (\forall x.\ px \longleftrightarrow qx) \to p = q.$ Prove the following:

a) $\mathsf{FE} \to \mathsf{PE} \to \mathsf{SE}$.

b) $\mathsf{SE} \to \mathsf{PE}$.

c) $\mathsf{SE} \to p - (q \cup r) = (p - q) \cap (p - r)$.

Assignment 8

Do the proofs with the proof assistant and explain the proof ideas on paper.

Exercise 8.1 (Arithmetic proofs from first principles)

Prove the following statements not using lemmas from the Coq library. Use the predefined definitions of addition and subtraction and define order as $(x \leq y) := (x - y = 0)$. Start from the accompanying Coq file providing the necessary definitions.

a) $x + y = x \to y = 0$
b) $x - 0 = x$
c) $x - x = 0$
d) $(x + y) - x = y$
e) $x - (x + y) = 0$
f) $x \leq y \to x + (y - x) = y$
g) $(x \leq y) + (y < x)$
h) $\neg(y \leq x) \to x < y$
i) $x \leq y \longleftrightarrow \exists z.\, x + z = y$
j) $x \leq x + y$
k) $x \leq \mathsf{S}x$
l) $x + y \leq x \to y = 0$
m) $x \leq 0 \to x = 0$
n) $x \leq x$
o) $x \leq y \to y \leq z \to x \leq z$
p) $x \leq y \to y \leq x \to x = y$
q) $x \leq y < z \to x < z$
r) $\neg(x < 0)$
s) $\neg(x + y < x)$
t) $\neg(x < x)$
u) $x \leq y \to x \leq y + z$
v) $x \leq y \to x \leq \mathsf{S}y$
w) $x < y \to x \leq y$
x) $\neg(x < y) \to \neg(y < x) \to x = y$
y) $x \leq y \leq \mathsf{S}x \to x = y \vee y = \mathsf{S}x$
z) $x + y \leq x + z \to y \leq z$

Exercise 8.2 (Arithmetic proofs with automation)

Do the problems of Exercise 1 with Coq's definition of order and the automation tactic lia.

Exercise 8.3 (Complete induction)

a) Define a certifying function $\forall xy.\ (x \leq y) + (y < x)$.

b) Prove a complete induction lemma.

c) Prove $\forall xy.\Sigma ab.\ x = a \cdot \mathsf{S}y + b \ \wedge\ b \leq y$ using complete induction and repeated subtraction.

d) Formulate the procedural specification

$$\begin{aligned} &f : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &f x y \ := \ \text{IF } \ulcorner x \leq y \urcorner \text{ THEN } x \text{ ELSE } f\,(x - \mathsf{S}y\ y)\ y \end{aligned}$$

as an unfolding function using the function from (a).

e) Prove that all functions satisfying the procedural specification agree.

f) Let f be a function satisfying the procedural specification.

   i) Prove $\forall xy.\ fxy \leq y$.

   ii) Prove $\forall xy.\ \Sigma k.\ x = k \cdot \mathsf{S}y + fxy$.

Assignment 9

Do all exercises with the proof assistant.

Exercise 9.1 (Certifying deciders with lia)

Define deciders of the following types using lia but not using induction.

a) $\forall xy.\ (x \le y) + (y < x)$

b) $\forall xy.\ (x \le y) + \neg(x \le y)$

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

d) $\forall xy.\ (x < y) + (x = y) + (y < x)$

Exercise 9.2 (Uniqueness with trichotomy)

Show the uniqueness of the predicate δ for Euclidean division using nia but not using induction.

Exercise 9.3 (Euclidean quotient)

We consider $\gamma\, xya := (a \cdot \mathsf{S}y \le x < \mathsf{S}a \cdot \mathsf{S}y)$.

a) Show that γ specifies the Euclidean quotient: $\gamma\, xya \longleftrightarrow \exists b.\ \delta xyab$.

b) Show that γ is unique: $\gamma xya \to \gamma xya' \to a = a'$.

c) Show that every function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfies

$$(\forall xy.\ \gamma\, xy(fxy)) \longleftrightarrow \forall xy.\ fxy = \text{IF } \ulcorner x \le y \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f(x - \mathsf{S}y)y)$$

d) Consider the function

$$\begin{aligned} &f : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &f0yb \;:=\; 0 \\ &f(\mathsf{S}x)yb \;:=\; \text{IF } \ulcorner b = y \urcorner \text{ THEN } \mathsf{S}(fxy0) \text{ ELSE } fxy(\mathsf{S}b) \end{aligned}$$

Show $\gamma xy(fxy0)$; that is, $fxy0$ is the Euclidean quotient of x and $\mathsf{S}y$. This requires a lemma. Hint: Prove $b \le y \to \gamma\,(x+b)\,y\,(fxyb)$.

Exercise 9.4 (Least and safe predicates)

a) Prove $\mathsf{safe}\ p(\mathsf{S}n) \longleftrightarrow \mathsf{safe}\ pn \wedge \neg pn$.

b) Prove $\mathsf{least}\,(\lambda a.\ x < \mathsf{S}a \cdot \mathsf{S}y)a \longleftrightarrow \exists b.\ x = a \cdot \mathsf{S}y + b \wedge b \le y$.

c) Prove $\mathsf{least}\,(\lambda z.\ x \le y + z)z \longleftrightarrow z = x - y$.

d) Show that the predicates in (b) and (c) are decidable using lia.

e) Prove $(\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{ex}\, p \to \mathsf{ex}\,(\mathsf{least}\, p)) \to \forall x.\ \mathsf{safe}\ p\,x \vee \mathsf{ex}\,(\mathsf{least}\, p)$.

Exercise 9.5 (Least witness search)
Let $p^{\mathsf{N}\to\mathbb{P}}$ be a decidable predicate and L and G be the functions from §17.4 of MPCTT. Prove the following:

a) $\forall n.\ \mathsf{least}\,p\,(Gn) \lor (Gn = n \land \mathsf{safe}\,pn)$

b) $\forall n.\ pn \to \mathsf{least}\,p\,(Gn)$

c) $\forall nk.\ \mathsf{safe}\,pk \to \mathsf{least}\,p\,(Lnk) \lor (Lnk = k + n \land \mathsf{safe}\,p(k + n)$

d) $\forall n.\ pn \to \mathsf{least}\,p\,(Ln0)$

Assignment 10

Do all exercises with the proof assistant.

Exercise 10.1 (Relational specification of least witness operators)

One can give a relational specification of least witness operators in the way we have seen it for division operators. Given a decidable predicate $p^{\mathsf{N}\to\mathbb{P}}$, we define

$$\delta xy \;:=\; (\mathsf{least}\,py \wedge y \le x) \vee (y = x \wedge \mathsf{safe}\,px)$$

Understand and prove the following:

a) $\forall nxy.\; pn \to n \le x \to \delta xy \to \mathsf{least}\,py$ *soundness*

b) $\forall xyy'.\; \delta xy \to \delta xy' \to y = y'$ *uniqueness*

c) $\forall x\,\Sigma y.\; \delta xy$ *satisfiability*

d) $\forall x.\; \delta x(Gx)$ *correctness of G*

e) $\forall x.\; \delta x(Lx0)$ *correctness of L*

Claim (e) needs to be generalized to Lxy for the induction to go through.

Exercise 10.2 (List basics)

Define the universal eliminator and the constructor laws for lists. First on paper using mathematical notation, then with Coq.

Exercise 10.3 (List facts)

Understand and prove the following facts about lists:

a) $x :: A \neq A$

b) $(A +\!\!+ B) +\!\!+ C = A +\!\!+ (B +\!\!+ C)$

c) $\mathsf{len}\,(A +\!\!+ B) = \mathsf{len}\,A + \mathsf{len}\,B$

d) $x \in A +\!\!+ B \;\longleftrightarrow\; x \in A \vee x \in B.$

e) $x \in f@A \;\longleftrightarrow\; \exists a.\; a \in A \wedge x = fa.$

Exercise 10.4 (Lists over discrete type)

Understand and prove the following facts about lists over a discrete type:

a) $\mathsf{rep}\,A + \mathsf{nrep}\,A$

b) $\mathsf{nrep}\,A \longleftrightarrow \neg\mathsf{rep}\,A$

c) $\mathsf{dec}\,(\mathsf{rep}\,A)$

d) $x \in A \to \Sigma B.\; \mathsf{len}\,B < \mathsf{len}\,A \wedge A \subseteq x :: B$

e) $\mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \Sigma z.\; z \in A \wedge z \notin B$

f) $\mathsf{nrep}\,A \to \mathsf{nrep}\,B \to A \equiv B \to \mathsf{len}\,A = \mathsf{len}\,B$

Exercise 10.5 (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: $\mathsf{sum}\,A > \mathsf{len}\,A \;\to\; \Sigma x.\; x \in A \wedge x \ge 2$. First define the function sum.

Exercise 10.6 (Andrej's Challenge)
Assume an increasing function $f^{\mathsf{N}\to\mathsf{N}}$ (i.e., $\forall x.\ x < fx$) and a list A of numbers satisfying $\forall x.\ x \in A \longleftrightarrow x \in f@A$. Show that A is empty.

Assignment 11

Exercise 11.1 (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 11.2 (Non-repeating lists)

Assume a discrete base type and prove the following facts. You may use the discriminating element lemma.

a) $\mathcal{D}(x \in A)$ and $\mathcal{D}(A \subseteq B)$

b) $\forall A. \Sigma B.\, B \equiv A \wedge \mathsf{nrep}\, B$

c) $A \subseteq B \;\to\; \mathsf{len}\, B < \mathsf{len}\, A \;\to\; \mathsf{rep}\, A$

d) $\mathsf{nrep}\, A \to A \subseteq B \to \mathsf{len}\, B \le \mathsf{len}\, A \to \mathsf{nrep}\, B$

e) $\mathsf{nrep}\, A \to A \subseteq B \to \mathsf{len}\, B \le \mathsf{len}\, A \to B \equiv A$

f) $\mathsf{nrep}\,(f@A) \to \mathsf{nrep}\, A$

g) $\mathsf{nrep}\, A \to \mathsf{nrep}(\mathsf{rev}\, A)$

Exercise 11.3 (Equivalent nonrepeating lists)

Show that equivalent nonrepeating lists have equal length without assuming discreteness of the base type. Hint: Show $\mathsf{nrep}\, A \to A \subseteq B \to \mathsf{len}\, A \le \mathsf{len}\, B$ by induction on A with B quantified using a deletion lemma.

Exercise 11.4 (Existential characterizations)

Give non-recursive existential characterizations for $x \in A$ and $\mathsf{rep}\, A$ and prove their correctness.

Exercise 11.5 (Existential witness operator for booleans)

Let $p^{\mathsf{B} \to \mathbb{P}}$ be a decidable predicate. Prove $\mathsf{ex}\, p \to \mathsf{sig}\, p$.

Exercise 11.6 (Search types)

Prove the following facts about search types for a decidable predicate $p^{\mathsf{N} \to \mathbb{P}}$.

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) $pn \to m \le n \to Tm$

g) $\forall Z^{\mathbb{T}}.\, ((\neg pn \to T(\mathsf{S}n)) \to Z) \to Tn \to Z$

h) $\forall q^{\mathsf{N} \to \mathbb{T}}.\, (\forall n.\, (\neg pn \to q(\mathsf{S}n)) \to qn) \to \forall n.\, Tn \to qn$

i) $Tn \;\longleftrightarrow\; \exists k.\, k \ge n \wedge pk$

Note that (h) provides an induction lemma for T useful for direction $\to$ of (i).

Exercise 11.7 (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: Assume the definition gives you the constants

$$B : \mathbb{T} \qquad C : (B \to \bot) \to B \qquad M : \forall Z.\, B \to ((B \to \bot) \to Z) \to Z$$

First define a function $f : B \to \bot$ using the matching constant M.

Assignment 12

Exercise 12.1 (Intuitionistic ND)

Assume the weakening lemma and prove the following facts with diagrams giving for each line the names of the deduction rules used:

a) $(A \vdash \neg\neg\bot) \to (A \vdash \bot)$

b) $(A \vdash \neg\neg\neg s) \to (A \vdash \neg s)$

c) $(A \vdash s) \to (A \vdash \neg\neg s)$

d) $A \vdash s \;\to\; A, s \vdash t \;\to\; A \vdash t$

e) $A \vdash \neg\neg(s \to t) \to \neg\neg s \to \neg\neg t$

f) $(\vdash s \to t \to u) \to (A \vdash s) \to (A \vdash t) \to (A \vdash u)$

g) $(A \vdash s \to t) \to (A, s \vdash t)$

h) $(A \vdash s \vee t) \;\Leftrightarrow\; \forall u.\; (A, s \vdash u) \;\to\; (A, t \vdash u) \;\to\; (A \vdash u)$

Exercise 12.2 (Classical ND)

Assume the weakening lemma and prove the following facts with diagrams giving for each line the names of the deduction rules used:

a) $(A \dot\vdash \bot) \to (A \dot\vdash s)$

b) $(A \dot\vdash \neg\neg s) \to (A \dot\vdash s)$

c) $\dot\vdash s \vee \neg s$

d) $\dot\vdash ((s \to t) \to s) \to s$

Exercise 12.3 (Glivenko)

Assume $\forall A s.\ (A \vdash s) \to (A \dot\vdash s)$ and $\forall A s.\ (A \dot\vdash s) \to (A \vdash \neg\neg s)$ and prove the following:

a) $A \dot\vdash \neg s \;\Leftrightarrow\; A \vdash \neg s$

b) $A \dot\vdash \bot \;\Leftrightarrow\; A \vdash \bot$

c) $((\vdash \bot) \to \bot) \Leftrightarrow ((\dot\vdash \bot) \to \bot)$

Exercise 12.4 (Induction)

a) $(A \vdash s) \to pAs$ can be shown by induction on the derivation of $A \vdash s$. Give the proof obligation for each of the 9 deduction rules.

b) How do the obligations change if we switch to the classical system and prove $(A \dot\vdash s) \to pAs$?

c) As an example, give the proof obligations for a proof of $(A \dot\vdash s) \to (A \vdash \neg\neg s)$.

Exercise 12.5 (Reversion, challenging)

We define a reversion function $A \cdot s$ preserving the order of assumptions:

$$
\begin{aligned}
[] \cdot s &:= s \\
(t :: A) \cdot s &:= t \to (A \cdot s)
\end{aligned}
$$

Prove $(A \vdash s) \Leftrightarrow (\vdash A \cdot s)$.

Assignment 13

Exercise 13.1 (Formulas)

We consider an inductive type for formulas $s ::= x \mid \bot \mid s \to t$ with the constructors for, Var, Bot, and Imp.

a) Give the types of the constructors.

b) Give the type of the eliminator for formulas.

c) Define a recursive predicate ground for formulas saying that a formula contains no variables.

d) Prove $\mathsf{ground}(s) \to ([] \vdash s) + ([] \vdash \neg s)$ using the eliminator from (b).

Exercise 13.2 (Hilbert Systems)

We consider formulas $s ::= x \mid \bot \mid s \to t \mid s \vee t$.

a) Give the rules for the Hilbert systems $\mathcal{H}(s)$.

b) Give the types of the constructors for the inductive type family $A \Vdash s$. Explain why A is a uniform parameter and s is an index.

c) Complete the type of the induction lemma $\forall Ap.\ \cdots\ \to \forall s.\ A \Vdash s \to ps$.

d) Prove $(A \Vdash s \to s)$.

e) Prove $(A \Vdash t) \to (A \Vdash s \to t)$.

f) Prove $(s :: A \Vdash t) \to (A \Vdash s \to t)$.

Exercise 13.3 (Heyting evaluation)

Consider the Heyting interpretation $0 < 1 < 2$.

a) Define the evaluation function $\mathcal{E}$.

b) Give an assignment such that $((x \to y) \to x) \to x$ evaluates to 1.

c) Explain how one shows $\mathcal{H}(((x \to y) \to x) \to x) \to \bot$ using (b).

d) Give a formula that evaluates under all assignments to 2 but is not intuitionistically provable.

Exercise 13.4 (Certifying solver)

Assume that $\mathcal{E}$ is the boolean evaluation function and that every refutation predicate ρ has a certifying solver $\forall A.\ (\Sigma\, \alpha.\ \forall s \in A.\ \mathcal{E}\alpha s = \mathbf{T}) + \rho A$. Show the following:

a) $\lambda A. A \dot{\vdash} \bot$ is a refutation predicate.

b) $\mathcal{D}(\dot{\vdash} s)$.

c) $\dot{\vdash} s \Leftrightarrow \forall \alpha.\ \mathcal{E}\alpha s = \mathbf{T}$.

Exercise 13.5 (Refutation system)

Consider the predicate $\rho^{\mathsf{For} \to \mathbb{P}}$ inductively defined with the following rules:

$$\frac{\bot \in A}{\rho(A)} \qquad \frac{s \in A \qquad \neg s \in A}{\rho(A)}$$

$$\frac{(s \to t) \in A \qquad \rho(\neg s :: A) \qquad \rho(t :: A)}{\rho(A)} \qquad \frac{\neg(s \to t) \in A \qquad \rho(s :: \neg t :: A)}{\rho(A)}$$

$$\frac{(s \wedge t) \in A \qquad \rho(s :: t :: A)}{\rho(A)} \qquad \frac{\neg(s \wedge t) \in A \qquad \rho(\neg s :: A) \qquad \rho(\neg t :: A)}{\rho(A)}$$

$$\frac{(s \vee t) \in A \qquad \rho(s :: A) \qquad \rho(t :: A)}{\rho(A)} \qquad \frac{\neg(s \vee t) \in A \qquad \rho(\neg s :: \neg t :: A)}{\rho(A)}$$

a) Show $\rho(\neg(((s \to t) \to s) \to s))$.

b) Show $\rho(A) \to \exists s.\ s \in A \wedge \mathcal{E}\alpha s = \mathbf{F}$.

c) Show the weakening property: $\rho(A) \to A \subseteq B \to \rho(B)$.

d) Show ρ is a refutation predicate.

Appendix: Glossary

Here is a list of technical terms used in the text but not used (much) otherwise. The technical terms are given in the order they appear first in the text.

- Discrimination
- Inductive function
- Elimination restriction
- Computational falsity elimination
- Index condition and index variables
- Reloading match

Appendix: Historical Remarks

1. The first paper discussing *indexed finite types* seems to be McBride [21] from 2004. The HoTT book [26] from 2013 doesn't mention generic finite types. Neither do generic finite types appear in Martin-Löf's papers.

Appendix: To Do and Improvements

The following remarks are for the author.

1. Coq development of the certifying boolean solver in § 23.12.
2. Indexed numeral types (Chapter 27) need work. They are in bijection with recursive numeral types (§ 27.5). Recursive numeral types should not be re-introduced from scratch.
3. Reconsider introduction of indexed inductives. In cases where weak elimination suffices (e.g., propositional deduction systems) everything is smooth.
4. Constructor patterns always omit the arguments of the type constructor.
5. Refinement types are missing. Numeral types as refinement type of N may be nice.
6. Regular expressions (§ 28.4) could appear as separate chapter.

Improvements done

1. Injections and bijections introduced early.
2. Finite types introduced early and defined with lists.

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] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.

[5] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[6] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[7] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[8] R. L. Constable. Computational type theory. *Scholarpedia*, 4(2):7618, 2009.

[9] Thierry Coquand. Metamathematical investigations of a calculus of constructions, 1989.

[10] 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.

[11] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *CPP 2019, Lisbon, Portugal*, 2019.

[12] 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.

[13] 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.

[14] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

[15] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators, an Introduction.* Cambridge University Press, 2008.

[16] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *LICS 1994*, pages 208–212, 1994.

[17] 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.

[18] 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.

[19] Edmund Landau. *Grundlagen der Analysis: With Complete German-English Vocabulary*, volume 141. American Mathematical Soc., 1965.

[20] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[21] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[22] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[23] Bengt Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619, Sep 1988.

[24] Raymond M. Smullyan and Melvin Fitting. *Set Theory and the Continuum Hypothesis.* Dover, 2010.

[25] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory.* Cambridge University Press, 2nd edition, 2000.

[26] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[27] Louis Warren, Hannes Diener, and Maarten McKubre-Jordens. The drinker paradox and its dual. *CoRR*, abs/1805.06216, 2018.